AVR og Pointere

I den følgende opgave skal vi lave lidt pointer gymnastik. Du skal derfor lave et projekt i Atmel Studio ”Pointer1”, hvor du udover koden vist herunder også skal have UART filer med, så du kan få adgang til Serial kommunikation til/fra dit Target. Når der skrives UART filer, menes der de 3 filer, I kan finde i Moodle læringsrummet i læringselementet UART- USART under titlen : C – Header filer til UART. De 3 filer er her pakket i en .zip fil. Denne .zip filer pakker I bare ud i jeres projekt direktorie og adderer de 3 filer som eksisterende filer til jeres Atmel Studio Projekt. Så burde i meget gerne være kørende mht. Seriel kommunikation. **Denne øvelse med de 3 UART filer skal I gentage for de efterfølgende projekter også**.


```
#include “Uart.h”

int main(void)
{
   RS2323Init();               // Init UART

   int *i_ptr;                // int pointer
   char *c_ptr;               // char pointer
   int tal = 0XCC33;

   i_ptr = &tal;              // Sætter ipointer = adressen på tal | 0xCC33
   c_ptr = i_ptr;             // c_ptr peger nu på samme adr som i_ptr. | 0xCC33

   // Udskriver informationer omkring i_prt.
   printf("i_ptr adr=%x, adr i_ptr peger på=%x, indhold af det i_ptr peger på=%x\n", &i_ptr, i_ptr,
*i_ptr);

   while (1);
}
```

Kør programmet og noter, hvad microen skriver over den serielle forbindelse:

i_prt’s adresse: (&i_ptr): 4f8 (tal)

---

adressen, som i_prt peger på : (i_ptr): 4fa (tal)

---

indholdet af adressen, som i_prt peger på: (*i_ptr) cc33

Prøv nu at lave endnu en printf sætning, som udskriver de samme informationer omkring c_prt og noter de samme info herfra:

c_prt's adresse: (&c_ptr): 4f8

---

adressen, som c_prt peger på: (c_ptr): 4fa

Indholdet af variablen, som c_prt peger på: (*c_ptr): 33

Som du sikkert allerede har set, er der flere forskelle på i_ptr og c_ptr. For det første har de to pointere ikke samme adresse. De to pointere ligger altså to forskellige steder i hukommelsen… Okay, det vidste vi vel egentlig godt i forvejen?

Men indholdet af de to pointere er det samme, så de peger altså samme sted hen (På første byte af den samme 16 bit Int!) Og det er her forskellen på de to pointere slår igennem, int pointeren viser nemlig hele 16 bit værdien. Mens char pointeren kun viser de 8 mindst betydende bit. Og kigger vi i vores kode, har vi jo lavet den ene som en pointer til int, mens den anden er pointer til byte (char).

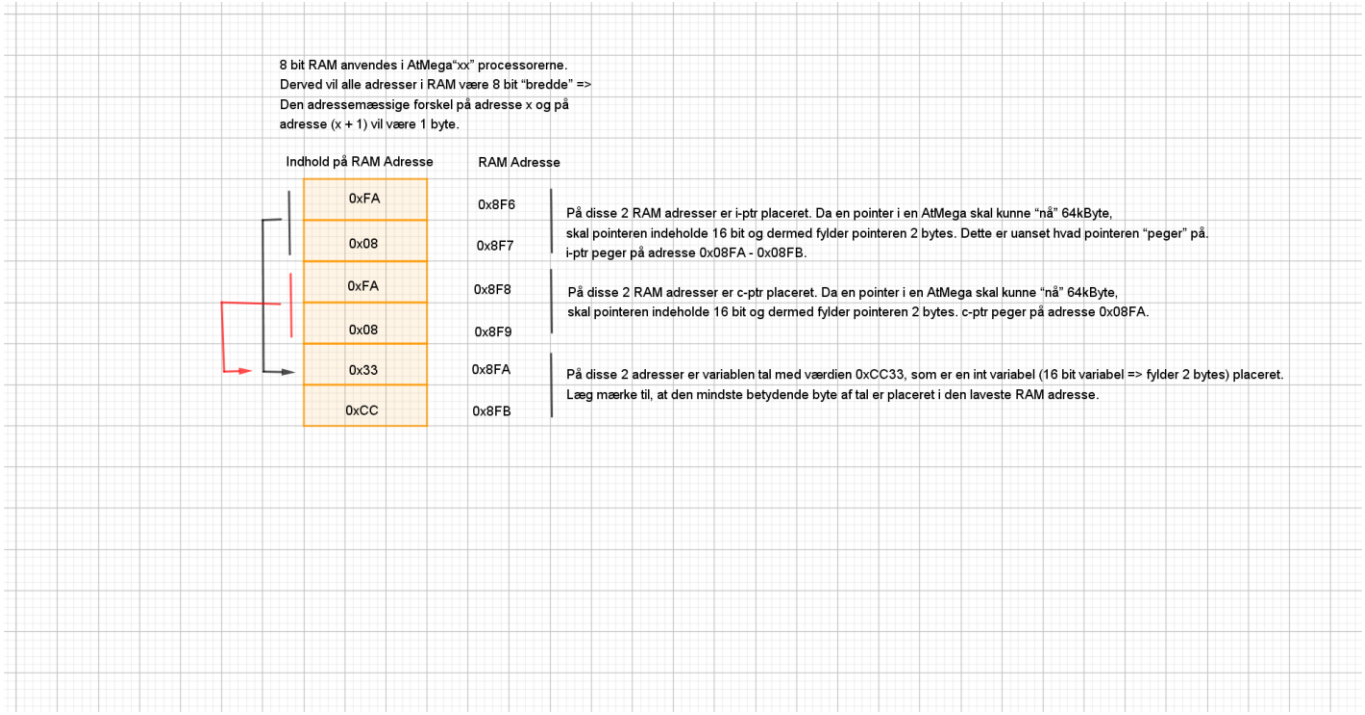
Hvad skal der til for at få char pointeren til også at vise de 8 mest betydende bit. Som vist herunder ??

```
i_ptr adr=8f6, adr i_ptr peger på=8fa, indhold af det i_ptr peger på=cc33
c_ptr adr=8f8, adr c_ptr peger på=8fa, indhold af det c_ptr peger på=33
c_ptr adr=8f8, adr c_ptr peger på=8fb, indhold af det c_ptr peger på=cc
```

Hvis man siger c_ptr++, flytter den sin adresse fra 8fa til 8fb i rammen, og kan derfor give værdien cc

Eksemplet her var mindre brugbart, men gav en indledende ide til hvordan pointere virker. Jeg har kørt eksemplet på en AtMega328P processor hvorfor jeg får adresserne vist herover. Kører man eksemplet på en anden AtMega<xx> processor, f.eks. en AtMega168 processor vil man få andre adresser.

For overskuelighedens skyld kan man også tegne sig frem til, hvordan de forskellige ting er placeret i processorens RAM. Dette er vist i figuren herunder.



Af figuren herover kan vi se, at en AtMega<xx> processor benytter sig af adressing mode, hvor den mindst betydende byte i f.eks. en int (en int er en 16 bit størrelse på en AtMega<xx> processor) er placeret på den laveste adresse i RAM.

Nu skal vi prøve at benytte pointere til funktioner. For det er her de bliver rigtig brugbare. (Og en måske også en bette smule uoverskuelige og derfor farlige)

Funktioner og pointere

Start med at lave et nyt projekt ”Pointer2” i Atmel Studio og tilføj, udover almindelig Seriel kommunikation, følgende kode:

```
#include “Uart.h”

int main(void)
{
		RS232Init();

		int tal1 = 17;
		int tal2 = 25;
		int resultat = 0;


		// Kalder Plus funktionen og kopierer automatisk de to variabler over på stakken.
		resultat = Plus(tal1, tal2);

		// viser at tal1 og tal2 stadig er de samme;
		printf("tal 1 = %d, tal2 = %d. Resultatet af Plus = %d\n", tal1, tal2, resultat);

		// Kalder pPlus funktionen og kopierer automatisk adresserne på de to variabler
		// over på stakken.
		resultat = ptrPlus(&tal1, &tal2);

		// viser at tal1 og tal2 blev modificeret i ptrPlus funktionen.
		printf("tal 1 = %d, tal2 = %d. Resultatet af pPlus = %d\n", tal1, tal2, resultat);

		while(1);
}

int Plus(int t1, int t2)
{
		// Bytter om på de to variabler (Kopierne)
		int tmp = t1;
		t1 = t2;
		t2 = tmp;

		tmp = t1 + t2;

		return tmp;
}

int ptrPlus(int *t1, int *t2)
{
		// Bytter om på de to variabler der peges til (Originalerne!)
		int tmp = *t1;
		*t1 = *t2;
		*t2 = tmp;

		tmp = *t1 + *t2;

		return tmp;
}
```

Kort fortalt har du nu lavet en ”almindelig” funktion ”Plus”, der tager noget input og afleverer 1 variabel som output (det vi er vant til i c). Funktionen overfører argumenterne tal1 og tal2 ”by value”, hvilket betyder, at der kommer to lokale variabler (t1 og t2), som er kopier af de originale værdier tal1 og tal2. Altså kan vi ændre i disse lokale variable t1 og t2 uden at påvirke tal1 og tal2 i main funktionen.

I funktionshovedet er det angivet, at funktionen returnerer en int (**int Plus**) og den aktuelle int værdi returneres med sætningen : **return tmp**; .

Derudover har du lavet en funktion (**ptrPlus**), der ved hjælp af pointere, kan modtage nogle input variable, som den kan modificere ved kode inde i selve funktionen (overførte variabler ”by reference”. Det som vi kender som ref variable i C#). Derudover har den som normalt den ene variabel som output => koden i funktionen har en return sætning (**return tmp**) og funktionshovedet for funktionen angiver, at den returnerede værdi er en int (**int ptrPlus**).

Altså er vi i funktionen **ptrPlus** ikke længere begrænset af en enkelt variabel som output fra vores funktioner. For vi kan jo reelt modificere de to variable ”T1 og T2 der er overført via pointer referencerne.

Inden du kører programmet, kan du jo lige overveje hvad forskellen på de to udskrifter bliver:

Forskellen ville være, at man i Plus funktionen kopierer de originale variabler, bytter om på værdierne **på kopierne** og returnerer den samlet værdi af kopierne. Altså de originale variablers værdier forbliver værdierne de var før, Plus funktionen kørte.

Dog, i ptrPlus funktionen, referer vi til de originale værdier. Når vi så bytter om på værdierne i ptrPlus funktionen, bytter vi hermed også værdierne i de originale variabler. Resultatet af ptrPlus retunerer det samme som Plus funktionens resultat, da x + y = y + x.

Strukturer og pointere

Okay, det var da ikke så svært, men der er stadig ikke meget at vinde i eksemplet ovenfor. Lige udover at vi nu har flere muligheder med hensyn til I/O til vores funktioner? Faktisk vil du se, at funktionen uden pointere gør programmet 4 bytes mindre, end hvis vi brugte pointer versionen til begge vores regnestykker ovenfor. Så det er ikke af plads hensyn vi gør det. Endnu…

Men nu skal vi til at lege lidt med ”komplicerede” datastrukturer. Altså variabler der kan indeholde en eller flere simple variabler. (Og andre komplicerede typer hvis nødvendigt)

Først skal vi dog lige kigge lidt på, hvad en ”kompliceret” datatype er for noget… I almindelig C er det nemlig muligt at lave og anvende en anden type variabel, end de sædvanlige int, chars og arrays af samme. Vi har mulighed for at lave en såkaldt struct. Og en struct er som navnet antyder en struktur. En struktur kan i bund og grund indeholde de normale simple typer, men også andre structs. (Og funktioner om nødvendigt men det vil vi ikke komme ind på i denne omgang) **Hvis du allerede kender til OOP og classes, så er classes principielt storebror til structs**.

Lad os prøve med et eksempel. Et punkt på en 2 dimensional flade skal indeholde 2 koordinater - X og Y.

Hvis vi skulle repræsentere disse koordinater i almindelige simple typer, kunne det se nogenlunde således ud:

```
int X = 10;
int Y = 2;
```

Hvilket jo er fint nok, hvis vi kun har denne ene koordinat. Men hvad hvis der skulle en tredje akse og evt fart med også?

```
int X = 10;
int Y = 2;
int Z = 0;
int fart = 200;
```

Lige pludselig er der mange variable at holde styr på. Specielt hvis der er flere af disse punkter i samme program?!. Nemmere er det, hvis vi indkapsler disse variabler i en struct:

```
struct Vector
{
   int X;
   int Y;
   int Z;
   int fart;
};
```

Her har vi grupperet dem i en struct ved navn Vector. I koden kan vi så efterfølgende oprette variabler af denne type. Hvilket vi vil prøve nu.

Start med at lave et nyt Atmel Studio projekt ”Pointer 3” og tilføj følgende kode sammen med Seriel kommunikation:

```
// Opretter en datastruktur som kan indeholde 3 simple typer
struct Point3D
{
    int X;
    int Y;
    int Z;
};



int main(void)
{
    RS232Init();



    // Opretter en variabel af typen Point3D
    struct Point3D punkt1;

    // Tilskriver de enkelte variabler en af gangen.
    punkt1.X = 10;
    punkt1.Y = 2;
    punkt1.Z = 0;

    // her er et eksempel på hvordan du også kan oprette
    // og tilskrive en struct med det samme
    // struct Point3D newPoint = {20, 7, 10};
    // Rækkefølgen er            X, Y,  Z  -> Altså som de er listet i structen.

    //udskriver værdier inden vi kalder doStuff
    printf("X = %d, Y = %d, Z = %d\n",punkt1.X, punkt1.Y, punkt1.Z);

    //Overfører en lokal kopi af en point3D
    doStuff(punkt1);

    //udskriver værdier efter vi har kaldt doStuff
    printf("X = %d, Y = %d, Z = %d\n",punkt1.X, punkt1.Y, punkt1.Z);

    while(1);
}

void doStuff(struct Point3D punkt)
{
    // Dette er en lokal kopi, så vores ændringer gælder kun herinde...
    printf("Ændrer i den lokale kopi\n");
    punkt.X--;
```

```
        punkt.Y++;
        punkt.Z++;
    }
```

Hvis du kører programmet, vil du se, at der ikke er forskel på de to udskrifter. Altså er det præcis som tidligere. Argumentet til doStuff bliver ”passed by value”. Altså bliver hele structen sendt som en kopi til funktionen => den bliver kopieret over på stacken… (Stacken er den øvre del af de 1024Bytes RAM (AtMega168) / 2048Bytes RAM (AtMega328P), som MCU’en har. Rammen indeholder også statiske variabler, - dem i ”” til printf – samt alle andre variabler, så der er ikke meget at spilde af…) Desuden vil ændringerne på kopien ikke slå igennem på originalen.

Men det skal du nu lave op på! Lav en ptrDoStuff funktion som tager en pointer til structen i steder for som tidligere selve struct’en. I funktionen skal du lave kode, så du ændrer i den originale struct, som pointeren jo peger på.

Anvendelse af structs med pointere er lidt anderledes end det punktum vi brugte tidligere. Her bruger vi nemlig ”pile”. Altså således :

```
punkt->X = 123;

eller

int tal = punkt->X;
```

Det er dog også muligt at anvende den ”normale” pointer syntaks i stedet for -> syntaksen til at nå elementer i en struct angivet som en pointer. Vælger man at gøre dette, vil syntaksen være som følger:

```
(*Punkt).X = 123;

Eller

Int tal = (*Punkt).X;
```

Skriv hele funktionen samt kaldet til funktionen herunder og noter samtidig besparelsen du får ud af det nedenunder:

(Du kan se følgende data, i output vinduet, når du kompilerer programmet i Atmel Studio.

DATA forbrug ved at overføre structen ”by value”: 432 bytes

PROGRAM forbrug ved at overføre structen ”by value”: 2428 bytes

---

DATA forbrug ved at overføre structen ”by reference”: 444 bytes

---

PROGRAM forbrug ved at overføre structen ”by reference” 2478 bytes

---

Arrays og pointere

Hvad så med arrays. Arrays er en lille smule anderledes når det kommer til pointere. Kigger vi under motorhjælmen på arrays, vil vi nemlig se, at arrays reelt bruger pointere ganske som vi så i pointer 1 eksemplet.

Endnu et eksempel er vist på sin plads. Start med at lave et nyt projekt ”Pointer4” og tilføj igen seriel kommunikation. Tilføj derefter følgende kode:

```
#include “Uart.h”

int main(void)
{
   RS232Init();

   char cData[] = {"Pointers are cool!"};
   char *pcData = NULL;

   printf("cData har %d bytesize elementer\n", sizeof(cData));

   // cData uden [] indeholder adressen på første element i arrayet.
   pcData = cData;

   //Forløkke ud fra størrelsen på char cData
   for(int i = 0; i < sizeof(cData); i++)
   {
      printf("&cData[%d] = 0x%x, cData[%d] = %c\n", i, &cData[i],  i, *pcData++);
   }

   while(1);
  }
```

Forklar følgende spørgsmål:

Hvad returnerer &cData[i]?

Addressen på char værdien af i

Hvad returnerer *pcData++?

Den næste adresse pcData skal stå på

Forklar derudover hvad funktion de to ++ har i relation til pdData?!

Da pcData peger på en adresse, menes der, når der ++es, at pcData skal stå på den næste adresse i rammen.

Udvid koden så vi også kommer til at kigge på et int array:

```
#include “Uart.h”

int main(void)
{
    RS232Init();;

    char cData[] = {"Pointers are easy as"};
    char *pcData = NULL;

    int iData[] = {1,2,3};
    int *piData = NULL;

    printf("cData har %d bytesize elementer\n", sizeof(cData));
    printf("iData har %d bytesize elementer\n", sizeof(iData));

    // cData indeholder adressen på første element i arrayet.
    pcData = cData;
    //Og nu peger pcData på samme element

    //Forløkke, af char array
    for(int i = 0; i < sizeof(cData); i++)
    {
              printf("&cData[%d] = %d, cData[%d] = %c\n", i, &cData[i],  i, *pcData++);
    }

    // pointer til test af int arrayet
    int *ip = iData;

    //forløkke, af int array
    for(int i = 0; i < sizeof(iData) / sizeof(int); i++)
    {
              piData = &iData[i];

              printf("&iData[%d] = 0x%x, iData[%d] = %d\n", i, piData,  i, *ip++);
    }

    while(1);
}
```

Kør programmet og svar på følgende spørgsmål:

Der er forskel på adresserne fra cData og iData hvorfor?

Det er fordi en int fylder 2 pladser i rammen, hvorimod en char kun fylder 1

Kan du forklarer `sizeof(iData) / sizeof(int)` sætningen?

C's måde at bruge iData.Length

At overfører arrays til funktioner er ikke meget anderledes end at overfører pointere til normale variabler. Du skal dog vide, at man fra funktioner ikke kan bruge sizeof funktionen. Her er et eksempel:

```
int main(void)
{

    RS232Init();

    char cData[] = "Pointers aren't always cool!";

    PrintSize(cData);

    while(1);
}

void PrintSize(int aTest[])
{
    printf("aTest er %d bytes stort", sizeof(aTest));
}
```

Hvis du opretter følgende kode i et nyt projekt, vil du se, at du får svaret "aTest er 2 bytes stort". Dette skyldes, at funktionen kun kan se den overførte pointer og eftersom vi befinder os i ATMega verdenen, så er adresser 2 bytes store… Dette betyder, du som programmør selv skal sørge for at sende en størrelse med til den pågældende funktion som vist her:

```
void PrintSize(int aTest[], int size)
{
   for(int i = 0; i < size, i++)
   {
      // Gør noget med arrayet
```

Eller du kan i din kode indbygge en sikkerhed, der sørger for at stoppe funktionen, når enden af arrayet er nået. Et klassisk eksempel er streng behandling:

```
int main(void)
{
   RS232Init();

   char cData[] = "Pointers aren't always cool!";

   myPrint(cData, stdout);

   while(1);
}
```

```
void myPrint(char tekst[], FILE *stream)
{
   //Udskriver samtlige tegn, indtil vi rammer nul termineringen
   for(int i = 0; tekst[i] != '\0'; i++)
   {
      uart_putch(tekst[i], stream);
   }
}
```

Input af karakterer fra Uart

Vi har set flere gange, at vi kan omdirigere **stdout** funktionaliteten (ved brug af **printf**) til at sende karakterer i en printf sætning ud på **UART'en.** Dette har vi gjort ved at sætte vores **stdout** til at "pege" på vores definerede **uart_output** i **Uart.c** filen (`static FILE uart_output = FDEV_SETUP_STREAM(uart_putch, NULL, _FDEV_SETUP_WRITE);`). Det er derfor, vi har haft sætningen : `stdout = &uart_output;` stående i bunden af vores RS232Init funktion i de forskellige øvelser.

Hvis vi kigger nærmere på static File definitionerne i toppen af vores Uart.c fil, kan vi se, at vi også har sætningen : `static FILE uart_input = FDEV_SETUP_STREAM(NULL, uart_getch, _FDEV_SETUP_READ);` til at stå her. Denne definition giver mulighed for, at vi kan læse karakterer fra UART'en ved brug af funktionen `uart_getch`. Denne funktion (`uart_getch`) er allerede implementeret i vores uart.c fil og sammenkoblingen mellem vores stdin funktionalitet og funktionen `uart_getch` er også at finde nederst i vores RS232Init funktion : `stdin = &uart_input`. Vi kan således bruge funktionen (`uart_getch`) fra andre *.c filer også ved simpelt at have en include af uart.h i *.c filer, hvor vi ønsker at benytte uart_getch funktionen. Og så "bare" kalde funktion funktionen **getchar**. Så vil funktionaliteten i stdio.h filen sørge for sammenkoblingen beskrevet her.

I eksemplet herunder er det vist, hvordan det i praksis foregår, når vi skal have kædet en læsefunktionalitet sammen med vores UART og vores `uart_getch` funktion ? . Kopier alt jeres kode fra "Pointer4" opgaven til en ny opgave : "Pointer5". I den nye opgave kan i main.c erstatte alt jeres kode med koden vist herunder.

```
#include <avr/io.h>

#include "uart.h"

void PrintSize(int aTest[]);
void myPrint(char tekst[], FILE *stream);
void ConvertReceivedChar(char *ReceivedChar);

#define Upper_Lower_Bit_Position 5
#define Upper_Lower_Bit_Value (1 << Upper_Lower_Bit_Position)

int main(void)
{
            char ch; // Global variabel til udveksling af data
```

```
		char cData[] = "Pointers aren't always cool!";

		RS232Init();

		printf("\n");

		PrintSize(cData);

		printf("\n");

		myPrint(cData, stdout);

		printf("\n");

		//while(1);

		/* Replace with your application code */
		while (1)
		{
			//ch = uart_getch (stdin);
			// Hvis ikke vi havde defineret stdin funktionalitet, måtte
			// vi kalde uart_getch direkte og have styr på vores stream
			// også, hvis vi vil læse karakterer fra UART'en
			 ch = getchar();
			// Nu kan vi stedet for bare nøjes med at kalde getchar funktionen
			// og så sker det hele ganske automatisk for os. Helt ligesom med
			// brugen af printf => bare den anden vej runct.
			 ConvertReceivedChar(&ch);
			 uart_putch(ch, stdout);
			 printf("\n");
		}
}



void PrintSize(int aTest[])
{
		printf("aTest er %d bytes stort", sizeof(aTest));
}


void myPrint(char tekst[], FILE *stream)
{
		//Udskriver samtlige tegn, indtil vi rammer nul termineringen
		for(int i = 0; tekst[i] != '\0'; i++)
		{
			 uart_putch(tekst[i], stream);
		}
}

void ConvertReceivedChar(char *ReceivedChar)
{
  // Den smarte måde at få konverteret små bogstaver om til store bogstaver og
  // modsat er ved brug af Xor, som vi tidligere har set. Så kan vi klare det i
  // én linje kode.
		*ReceivedChar ^= Upper_Lower_Bit_Value;

  // Vi kan også vælge at bruge den mere besværlige (og udkommenterede) metode vist
  // herunder.
```

```
            //if (*ReceivedChar & Upper_Lower_Bit_Value)
            //{
             //           *ReceivedChar = *ReceivedChar & ~Upper_Lower_Bit_Value;
             //}
             //else
             //{
             //           *ReceivedChar = *ReceivedChar | Upper_Lower_Bit_Value;
             //}
}
```

Fordelen ved at bruge stdin (getchar) og stdout (print) er, at vi ikke skal bekymre os om streams. Det hele foregår inde bagved, uden at vi behøver at bekymre os om det.

Faktisk kan vi løbende i vores program sætte stdout og stdin til at pege på andre streams end de her angivne. Hvis vi f.eks. har et Display koblet på vores embeddede system, kan vi bare sætte stdout til at "pege" på vores display skrive rutine. Og så vil efterfølgende kald af **printf** bevirke, at argumenterne ved de forskellige kald vil blive udskrevet på Displayet i stedet for på UART. Ønsker vi igen, at vores printf sætninger, skal dirigeres til UART igen, sætter vi følgelig bare vores stdout til at pege på UART funktionen igen. Det helt samme gør sig gældende med stdin funktionaliteten. Her kunne vi forestille os, at vi også havde et Tastatur koblet på vores embeddede system og så kunne vi stå og skifte mellem stdin "pegende" på vores UART rutine og vores Tastatur læse rutine. Når vi kalder **getchar**, vil systemet nedenunder bestemme hvorfra vi læser => ret smart !!!!

Læg også mærke til, at der er implementeret en funktion **ConvertReceivedChar**, som konverterer små bogstaver om til store bogstaver og vice versa. Kan du forklare funktionaliteten i funktionen. Selvom der ikke er mange linjer kode i funktionen (i den mest komprimerede form kun én linke kode), er der alligevel en hel del C funktionalitet i funktionen => pointere og bit gymnastik.