# Bonusaufgaben zum C/C++-Praktikum

**Einführung in C**

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungsblatt 5

**Hinweise zur Abgabe:**

- Es ist eine Datei abzugeben: `main.c`
- Der Code muss kompilierbar sein, um bewertet zu werden

## Aufgabe 5.1: `[C]` Strings zusammenfügen (4 Punkte)

### 5.1a) Mit `sprintf()` konkatenieren (1 Punkt)

Schreibe eine Funktion `void sprintf_concatenator(char *dest, const char *a, const char *b)`, die als Parameter

- einen Pointer `char *`, und
- zwei Zeichenketten `const char *`

übergeben bekommt und mit der Funktion `sprintf(char *str, const char *format, ...)` aus der Standardbibliothek beide Zeichenketten zusammenfügt und in dem angegebenen Speicherbereich speichert.

Gehe davon aus, dass der Speicherbereich von `dest` groß genug ist, um die resultierende Zeichenkette zu speichern.

### 5.1b) Funktion `concatenate()` (1 Punkt)

Schreibe eine Funktion `char *concatenate(void (*func)(char*, const char*, const char*), const char* a, const char* b)`, die als Parameter

- zwei Zeichenketten `const char *`, und
- einen Funktionspointer auf eine Funktion mit der Signatur von `void sprintf_concatenator(char *dest, const char *a, const char *b)` aus vorheriger Teilaufgabe

übergeben bekommt. Sie alloziert einen Speicherbereich, in den die beiden Zeichenketten zusammengefügt genau[1] hineinpassen, ruft die übergebene Funktion auf, um die beiden Speicherketten zu konkatenieren, und gibt die konkatenierte Zeichenkette zurück.

Überprüfe nun, dass sie funktioniert; beispielsweise so:

[1] Alloziere exakt die benötigte Menge an Speicher

```c
int main() {
    char *str;
    str = concatenate("Hello ", "World!", sprintf_concatenator);
    puts(str);  // gibt "Hello World!" aus.
    free(str);
    return 0;
}
```

### 5.1c) Mit `strcpy()` konkatenieren (1 Punkt)

Schreibe eine Funktion `void strcpy_concatenator(char *dest, const char *a, const char *b)`, die durch Aufrufe von `strcpy(char *destination, const char *source)` aus der Standardbibliothek beide übergebenen Zeichenketten zusammenfügt und in dem angegebenen Speicherbereich speichert. Sie darf auch die Funktion `strlen(const char *str)` verwenden.

Ergänze deine `main()`-Funktion um einen Aufruf von `concatenate(void (*func)(char*, const char*, const char*), const char* a, const char* b)` mit `strcpy_concatenator(char *dest, const char *a, const char *b)`.

### 5.1d) Selber konkatenieren (1 Punkt)

Schreibe eine Funktion `void loop_concatenator(char *dest, const char *a, const char *b)`, die beide übergebenen Zeichenketten zusammenfügt und in dem angegebenen Speicherbereich speichert, ohne eine Funktion aus einer zusätzlichen Bibliothek aufzurufen.

Ergänze deine `main()`-Funktion um einen Aufruf von `concatenate(void (*func)(char*, const char*, const char*), const char* a, const char* b)` mit `loop_concatenator(char *dest, const char *a, const char *b)`.

## Aufgabe 5.2: `[C]` String Manipulation (10 Punkte)

In den folgenden Aufgaben sollt ihr den Hauptteil der Implementation selbst schreiben und nicht auf die vorhandenen Funktionen der Standardbibliothek zurückgreifen. Zur Vereinfachung dürft ihr aber `strlen()` [2] und `memcpy()` [3] nutzen. `strlen()` gibt euch die Länge einer Zeichenkette aus und mit `memcpy()` könnt ihr eine bestimmte Anzahl von Bytes von einem Speicherblock in einen anderen kopieren.

*Achtung: `strlen()` gibt euch die Länge des Strings ohne Terminator zurück, d.h. ein String mit strlen 10 hat im Speicher die Größe 11 Byte. Zum Beispiel: Die Zeichenkette "Hello" würde im Speicher so aussehen: 'H' + 'e' + 'l' + 'l' + 'o' + '\0'.*

### 5.2a) Substrings (3 Punkte)

Schreibe eine Funktion `int substring(char* dest, const char* src, size_t start, size_t n)`.

Die Funktion soll einen Teil der übergebenen Zeichenkette `src` in den `char`-Pointer `dest` mithilfe von `memcpy` kopieren. Der Bereich, der kopiert werden soll, wird durch die zwei `size_t` Werte gekennzeichnet. Der Parameter `start` steht für die Position des ersten Zeichen, welches kopiert werden soll. Der Parameter `n` steht für die Anzahl von Zeichen, die

[2] https://www.cplusplus.com/reference/cstring/strlen/
[3] https://www.cplusplus.com/reference/cstring/memcpy/

kopiert werden sollen. Der Rückgabewert wird benutzt, um anzuzeigen, ob ein Fehler geschehen ist. Im Normalfall soll die Funktion 0 zurückgeben. Wenn aber der zu kopierende Bereich außerhalb von `src` ist, soll -1 zurück gegeben werden. Ihr könnt davon ausgehen, dass `dest` groß genug ist.

Beispiel: Nach dem Aufruf der Funktion mit dem Parametern `src = "Hello World!"`, `start = 6` und `n = 5`, soll die Zeichenkette **"World"** in `dest` gespeichert sein.

## 5.2b) String split (3 Punkte)

Schreibe eine Funktion `size_t split(char** dest, const char* src, const char* splitStr)`.

Die Funktion soll die Zeichenkette `src` in ein Array von Zeichenketten unterteilen. Die Unterteilung soll immer dort stattfinden, wo `splitStr` in `src` vorhanden ist. Das Array von Zeichenketten soll in `dest` gespeichert werden und die Anzahl von Zeichenketten soll der Rückgabewert sein. Wichtig ist, dass `splitStr` nicht ein Zeichen ist, sondern eine Zeichenkette. Ihr könnt davon ausgehen, dass `dest` groß genug ist.

Beispiel: Nach dem Aufruf der Funktion mit den Parametern `src = "Hello;;World;;!"` und `splitStr = ";;"` soll das Array `["Hello", "World", "!"]` in `dest` abspeichern und 3 zurückgeben.

## 5.2c) String replace (3 Punkte)

Schreibe eine Funktion `void replace(char* dest, const char* src, const char* id, const char * replaceStr)`.

Die Funktion soll alle Vorkommnisse von `id` in `src` mit `replaceStr` ersetzen und die resultierende Zeichenkette in `dest` abspeichern. Ihr könnt davon ausgehen, dass `dest` groß genug ist.

Beispiel: Nach dem Aufruf der Funktion mit den Parameters `src = "He;;;;o Wor;;d!"`, `id = ";;"`, `replaceStr = "l"` soll die Zeichenkette "Hello World" in `dest` abspeichern.

*Hinweis: Hier können euch die vorherigen Aufgaben vielleicht helfen.*

## 5.2d) Implementation testen (1 Punkt)

Um eure Implementationen zu testen, sollt ihr einen "korrupten" String reparieren.
Der String lautet: "00000000{}::is::fun00000000" und soll in 3 Schritten repariert werden:

1. Nullen mithilfe von `substring(char* dest, const char* src, size_t start, size_t n)` entfernen
2. Die Zeichenkette "{}" mithilfe von `replace(char* dest, const char* src, const char* id, const char* replaceStr)` durch eine Zeichenkette eurer Wahl ersetzen
3. Mithilfen von `split(char** dest, const char* src, const char* splitStr)` soll die Zeichenkette anhand von "::" gespalten werden.

Gebt das resultierende Array von Zeichenketten so aus, dass in jeder Zeile eine Zeichenkette des Arrays ist.
Die Ausgabe könnte beispielsweise so aussehen:

```
C++
is
fun
```

Falls ihr die benötigten Aufgaben nicht gemacht habt, könnt ihr stattdessen leere Funktionen benutzen. Denkt daran, dass die Funktionen erwarten, dass `dest` groß genug ist.

Schreibt den Code hierfür in der `main()`-Funktion in `main.c`.

## Aufgabe 5.3: [C] Register-Manipulation (5 Punkte)

In Mikrocontrollern werden Peripheriekomponenten wie Analog-Digital-Wandler, serielle Kommunikationsschittstellen usw. meist über Register gesteuert. Diese können vom Prozessor analog zum Hauptspeicher adressiert werden. In dieser Aufgabe sollt ihr ein Pseudoregister über verschiedene Wege manipulieren. Hierbei ist es wichtig, dass nur die gewünschten Bits verändert werden und die restlichen Bits des Registers nicht verändert werden, da das sonst Einfluss auf die Hardware haben würde. Das Register, das ihr verändern sollt, ist in der Header-Datei `register.h` vorgegeben.

Das Datenblatt des Registers findet ihr am Ende des Aufgabenblatt. Keine Sorge, ihr müsst nicht das gesamte Datenblatt ausgiebig studieren. Konzentriert euch am besten auf die Dokumentation der für euch relevanten Bits.

Schreibt den Code hierfür in der `main()`-Funktion in `main.c`

### 5.3a) Bitweise Operatoren (2 Punkte)

Als erstes sollt ihr das Register über bitweise Operatoren manipulieren, z.B. mit sogenannten Bitmasken. Bei einer Bitmaske sind Zielbits 1 und allen anderen Bits 0, sodass z.B. `(REG & BIT17)` mit `BIT17 = (1 << 17)` als `true` interpretiert werden kann, wenn das 17. Bit 1 ist, ansonsten ergibt es false. In C gibt es den Typ `bool` nicht nativ, dafür wird der Wert 0 als `false` interpretiert und jede andere Zahl als `true`. Abbildung 1 zeigt ein Beispiel für beide Fälle.

```
unsigned int BIT4 = 1 << 4; // 0b00010000

unsigned int REG1 = 0b00001111;
unsigned int isBit4Set1 = REG & BIT4; // 0b00000000 = 0 = false

unsigned int REG2 = 0b11110000;
unsigned int isBit17Set2 = REG2 & BIT 4; // 0b00010000 = 16 = true
```

Abbildung 1: Beispiel für eine Bitmaske

Benutzt bitweise Operatoren, um folgende Einstellungen zu setzen:

- ADC 6 und 7 digital enabled, der Rest disabled
- $T_q$ divider zu $60 * T_{clk} = T_q$
- $AD_{REF+} = Internal\ V_{REFH}$, $AD_{REF-} = AV_{SS}$

Alle anderen Felder des Registers müssen und dürfen nicht verändert werden.
*Tipp: Benutzt defines wie* `#define BIT17 (1 << 17)`*, um euren Code übersichtlicher zu machen.*

## 5.3b) Mit Structs (2 Punkte)

Ein anderer Weg zur Registermanipulation ist das Struct. Mit dem Syntax aus Abbildung 2 können einzelne Bitfelder oder Gruppen manipuliert werden. Benutzt ein solches Struct, um die selben Änderungen wie in 5.2a vorzunehmen. Den Code hierfür sollt ihr entweder oberhalb oder unterhalb des Codes der letzten Aufgabe in der `main()`-Funktion schreiben. In der nächsten Aufgabe werden wir etwas hinzufügen, mit dem man entscheiden kann, welche der beiden Optionen benutzt werden soll.
*Achtung: Ihr sollt immer noch das selbe Register manipulieren, d.h. ihr müsst sicherstellen, dass euer Struct zur richtigen Speicherstelle zeigt.*

```
struct Register_Example {
    // Kleinstes Bit
    uint32_t 16; // Die ersten 16 Bit werden ignoriert
    uint32_t some_char : 8; // some_char ist 8 Bit groß
    uint32_t flag : 1; // flag ist ein Bit groß
    uint32_t 7; // Bits können ignoriert werden, indem der Name weggelassen wird
    // Höchstes Bit
}

unsigned int SOME_REG = 0b00000000000000000000000000000000; // Beispiel Register
struct Register_Example *reg = &SOME_REG; // Register als struct darstellen
reg->some_char = 'A'; // Zugriff auf Bit 16-23
reg->flag = 1; // Zugriff auf Bit 24
// SOME_REG = 0b00000001010000010000000000000000
//                      ^^------^
//                      |   |
//                   flag some_char
```

Abbildung 2: Struct mit welchem bestimmte Bits eines 32-Bit Register manipuliert werden können

## 5.3c) Variante zur Compile-Zeit auswählen (1 Punkt)

Wir haben jetzt zwei Varianten kennengelernt um Register zu manipulieren. Wir wollen jetzt den Präprozessor benutzen um uns zur Compile-Zeit für eine der beiden Varianten zu entscheiden. Mit den `#if`, `#elif`, `#else`, `#endif` und `#ifdef` Direktiven des Präprozessors könnt ihr definierte Konstanten überprüfen bzw. prüfen, ob eine Konstante existiert. Beim Kompilieren könnt ihr mithilfe der zusätzlichen Option `-D Name[=Wert]` der Konstante mit dem `Namen` den `Wert` zuweisen. Baut in die `main()`-Funktion folgendes Verhalten ein:
Wenn das Programm mit `-D USE_STRUCTS` kompiliert wird, dann soll die Structs Variante benutzt werden, ansonsten die Bitwise Operators Variante.

Abbildung 3 zeigt ein kleines Beispiel. Wenn das Programm mit `gcc -D F main.c` kompiliert wird, dann gibt es “Hello” aus, mit `gcc main.c` gibt es “‘Word” aus.

```
int main() {
#ifdef F
    printf("Hello");
#else
    printf("World");
#endif
}
```

Abbildung 3: Beispiel für Präprozessor Direktiven

**Register 22-3: ADCCON3: ADC Control Register 3**

| Bit Range | Bit 31/23/15/7 | Bit 30/22/14/6 | Bit 29/21/13/5 | Bit 28/20/12/4 | Bit 27/19/11/3 | Bit 26/18/10/2 | Bit 25/17/9/1 | Bit 24/16/8/0 |
|---|---|---|---|---|---|---|---|---|
| 31:24 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
| | ADCSEL<1:0> | | CONCLKDIV<5:0> | | | | | |
| 23:16 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
| | DIGEN7(5) | DIGEN6(5) | DIGEN5(5) | DIGEN4(5) | DIGEN3(5) | DIGEN2(5) | DIGEN1(5) | DIGEN0(5) |
| 15:8 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R-0, HS, HC | R/W-0 | R-0, HS, HC |
| | VREFSEL<2:0> | | | TRGSUSP | UPDIEN | UPDRDY | SAMP(1,2,3,4) | RQCNVRT |
| 7:0 | R/W-0 | R-0, HS, HC | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
| | GLSWTRG | GSWTRG | ADINSEL<5:0>(5) | | | | | |

| Legend: | HC = Hardware Set | HS = Hardware Cleared | |
|---|---|---|---|
| R = Readable bit | W = Writable bit | U = Unimplemented bit, read as '0' | |
| -n = Value at POR | '1' = Bit is set | '0' = Bit is cleared | x = Bit is unknown |

bit 31-30 **ADCSEL<1:0>:** Analog-to-Digital Clock Source (TCLK) bits
Refer to the **"12-bit High-Speed Successive Approximation Register (SAR)"** chapter in the specific device data sheet for the ADC Clock source selections.

bit 29-24 **CONCLKDIV<5:0>:** Analog-to-Digital Control Clock (TQ) Divider bits
111111 = 126 * TCLK = TQ
.
.
.
000011 = 6 * TCLK = TQ
000010 = 4 * TCLK = TQ
000001 = 2 * TCLK = TQ
000000 = TCLK = TQ

bit 23 **DIGEN7:** ADC7 Digital Enable bit(5)
1 = ADC7 is digital enabled
0 = ADC7 is digital disabled

bit 22 **DIGEN6:** ADC6 Digital Enable bit(5)
1 = ADC6 is digital enabled
0 = ADC6 is digital disabled

bit 21 **DIGEN5:** ADC5 Digital Enable bit(5)
1 = ADC5 is digital enabled
0 = ADC5 is digital disabled

bit 20 **DIGEN4:** ADC4 Digital Enable bit(5)
1 = ADC4 is digital enabled
0 = ADC4 is digital disabled

**Note 1:** The SAMP bit has the highest priority and setting this bit will keep the S&H circuit in Sample mode until the bit is cleared. Also, usage of the SAMP bit will cause settings of the SAMC<9:0> bits (ADCCON2<25:16>) to be ignored.

**2:** The SAMP bit only connects Class 2 and Class 3 analog inputs to the shared ADC. All Class 1 analog inputs are not affected by the SAMP bit.

**3:** The SAMP bit is not a self-clearing bit and it is the responsibility of application software to first clear this bit and only after setting the RQCNVRT bit to start the analog-to-digital conversion.

**4:** Normally, when the SAMP and RQCNVRT bits are used by software routines, all TRGSRCx<4:0> bits and STRGSRC<4:0> bits should be set to '00000' to disable all external hardware triggers and prevent them from interfering with the software-controlled sampling command signal SAMP and with the software-controlled trigger RQCNVRT.

**5:** Depending on the device, the function will vary. Refer to the **"ADC"** chapter in the specific device data sheet to determine the function that is available for your device.

**Register 22-3: ADCCON3: ADC Control Register 3 (Continued)**

bit 19 **DIGEN3:** ADC3 Digital Enable bit(5)
1 = ADC3 is digital enabled
0 = ADC3 is digital disabled

bit 18 **DIGEN2:** ADC2 Digital Enable bit(5)
1 = ADC2 is digital enabled
0 = ADC2 is digital disabled

bit 17 **DIGEN1:** ADC1 Digital Enable bit(5)
1 = ADC1 is digital enabled
0 = ADC1 is digital disabled

bit 16 **DIGEN0:** ADC0 Digital Enable bit(5)
1 = ADC0 is digital enabled
0 = ADC0 is digital disabled

bit 15-13 **VREFSEL<2:0>:** Voltage Reference (VREF) Input Selection bits

| VREFSEL<2:0> | ADREF+ | ADREF- |
|---|---|---|
| 111 | AVDD | Internal VREFL |
| 110 | Internal VREFH | AVSS |
| 101 | Internal VREFH | External VREFL |
| 100 | Internal VREFH | Internal VREFL |
| 011 | External VREFH | External VREFL |
| 010 | AVDD | External VREFL |
| 001 | External VREFH | AVss |
| 000 | AVDD | AVss |

bit 12 **TRGSUSP:** Trigger Suspend bit
1 = Triggers are blocked from starting a new analog-to-digital conversion, but the ADC module is not disabled
0 = Triggers are not blocked

bit 11 **UPDIEN:** Update Ready Interrupt Enable bit
1 = Interrupt will be generated when the UPDRDY bit is set by hardware
0 = No interrupt is generated

bit 10 **UPDRDY:** ADC Update Ready Status bit
1 = ADC SFRs can be updated
0 = ADC SFRs cannot be updated

**Note:** This bit is only active while the TRGSUSP bit is set and there are no more running conversions of any ADC modules.

bit 9 **SAMP:** Class 2 and Class 3 Analog Input Sampling Enable bit(1,2,3,4)
1 = The ADC S&H amplifier is sampling
0 = The ADC S&H amplifier is holding

**Note 1:** The SAMP bit has the highest priority and setting this bit will keep the S&H circuit in Sample mode until the bit is cleared. Also, usage of the SAMP bit will cause settings of the SAMC<9:0> bits (ADCCON2<25:16>) to be ignored.

**2:** The SAMP bit only connects Class 2 and Class 3 analog inputs to the shared ADC. All Class 1 analog inputs are not affected by the SAMP bit.

**3:** The SAMP bit is not a self-clearing bit and it is the responsibility of application software to first clear this bit and only after setting the RQCNVRT bit to start the analog-to-digital conversion.

**4:** Normally, when the SAMP and RQCNVRT bits are used by software routines, all TRGSRCx<4:0> bits and STRGSRC<4:0> bits should be set to '00000' to disable all external hardware triggers and prevent them from interfering with the software-controlled sampling command signal SAMP and with the software-controlled trigger RQCNVRT.

**5:** Depending on the device, the function will vary. Refer to the **"ADC"** chapter in the specific device data sheet to determine the function that is available for your device.

**Register 22-3: ADCCON3: ADC Control Register 3 (Continued)**

bit 8 **RQCNVRT:** Individual ADC Input Conversion Request bit
This bit and its associated ADINSEL<5:0> bits enable the user to individually request an analog-to-digital conversion of an analog input through software.
1 = Trigger the conversion of the selected ADC input as specified by the ADINSEL<5:0> bits
0 = Do not trigger the conversion
**Note:** This bit is automatically cleared in the next ADC clock cycle.

bit 7 **GLSWTRG:** Global Level Software Trigger bit
1 = Trigger conversion for ADC inputs that have selected the GLSWTRG bit as the trigger signal, either through the associated TRGSRC<4:0> bits in the ADCTRGx registers or through the STRGSRC<4:0> bits in the ADCCON1 register
0 = Do not trigger an analog-to-digital conversion

bit 6 **GSWTRG:** Global Software Trigger bit
1 = Trigger conversion for ADC inputs that have selected the GSWTRG bit as the trigger signal, either through the associated TRGSRC<4:0> bits in the ADCTRGx registers or through the STRGSRC<4:0> bits in the ADCCON1 register
0 = Do not trigger an analog-to-digital conversion
**Note:** This bit is automatically cleared in the next ADC clock cycle.

bit 5-0 **ADINSEL<5:0>:** Analog Input Select bits(5)
These bits select the analog input to be converted when the RQCNVRT bit is set, where, MAX_AN_INPUT is the maximum analog inputs available on the device.
MAX_AN_INPUT + 4 = Device dependent (see **Note 5)**
MAX_AN_INPUT + 3 = Device dependent (see **Note 5)**
MAX_AN_INPUT + 2 = Device dependent (see **Note 5)**
MAX_AN_INPUT + 1 = Device dependent (see **Note 5)**
MAX_AN_INPUT = AN[MAX_AN_INPUT]
•
•
•
000001 = AN1
000000 = AN0

**Note 1:** The SAMP bit has the highest priority and setting this bit will keep the S&H circuit in Sample mode until the bit is cleared. Also, usage of the SAMP bit will cause settings of the SAMC<9:0> bits (ADCCON2<25:16>) to be ignored.

**2:** The SAMP bit only connects Class 2 and Class 3 analog inputs to the shared ADC. All Class 1 analog inputs are not affected by the SAMP bit.

**3:** The SAMP bit is not a self-clearing bit and it is the responsibility of application software to first clear this bit and only after setting the RQCNVRT bit to start the analog-to-digital conversion.

**4:** Normally, when the SAMP and RQCNVRT bits are used by software routines, all TRGSRCx<4:0> bits and STRGSRC<4:0> bits should be set to '00000' to disable all external hardware triggers and prevent them from interfering with the software-controlled sampling command signal SAMP and with the software-controlled trigger RQCNVRT.

**5:** Depending on the device, the function will vary. Refer to the **"ADC"** chapter in the specific device data sheet to determine the function that is available for your device.