

# Bonusaufgaben zum C/C++-Praktikum

Einführung in C



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



Übungsblatt 5

## Hinweise zur Abgabe:

verwendet als Grundlage für die Bearbeitung die im Moodle bereitgestellte Vorlage. Beachtet auch die Hinweise auf dem ersten Bonuszettel.

---

### Aufgabe 5.1: [C] Mit Strings arbeiten (5 Punkte)

---

Bitte bearbeite diese Aufgabe in `string.h` und `string.c` sowie die `main`-Funktion in `main.c`.

---

#### 5.1a) Mit `strcpy()` konkatenieren (2 Punkte)

---

Implementiere die Funktion `char *concatenate(const char *a, const char *b)`, die Speicher für die zusammengefügte Zeichenkette allokiert und durch Aufrufe von `strcpy(char *destination, const char *source)` aus der Standardbibliothek beide übergebenen Zeichenketten zusammenfügt. Sie darf auch die Funktion `strlen(const char *str)` verwenden.

Füge deiner `main()`-Funktion einen Aufruf von `char *concatenate(const char *a, const char *b)` hinzu.

---

#### 5.1b) String split (3 Punkte)

---

Implementiere die Funktion `int 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.

Teste dies Beispiel in der `main`-Funktion.

## Hinweise

- Für das Zusammenfügen zweier Strings können auch einfache Schleifen oder auch die Funktion `sprintf` aus der C Standardbibliothek verwendet werden. `sprintf` funktioniert ähnlich wie `printf`, nur dass das Ergebnis in einer Zeichenkette gespeichert wird, anstatt ausgegeben zu werden.

---

## Aufgabe 5.2: [C] Register-Manipulation (5 Punkte)

---

Bitte bearbeite diese Aufgabe in `registers.h` und `registers.c` sowie die `main`-Funktion in `main.c`.

In Mikrocontrollern werden Peripheriekomponenten wie Analog-Digital-Wandler, serielle Kommunikationsschnittstellen 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`

---

### 5.2a) 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.  $(\text{REG} \& \text{BIT17})$  mit  $\text{BIT17} = (1 \ll 17)$  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 = REG1 & BIT4; // 0b00000000 = 0 = false

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

Abbildung 1: Beispiel für eine Bitmaske

Implementiere die Funktion `void setRegisters(Register_t* reg)`, die bitweise Operatoren benutzt, um folgende Einstellungen im Register `reg` zu setzen:

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

Alle anderen Felder des Registers müssen und dürfen nicht verändert werden.

Füge der `main`-Funktion einen Aufruf von `setRegisters` hinzu, dem das vorgegebene Register `ADCCON3` übergeben wird.

*Tip: Benutzt defines wie `#define BIT17 (1 << 17)`, um euren Code übersichtlicher zu machen.*

---

### 5.2b) 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.

Implementiere die Funktion `void setRegistersStruct(Register_t* reg)`, die ein solches Struct benutzt, um die selben Änderungen wie in 5.2a im Register `reg` vorzunehmen.

Füge der `main`-Funktion auch einen Aufruf dieser Funktion hinzu. 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 = 0b00000000101000001000000000000000
//                ^^-----^
//                |   |
//                flag some_char
```

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

---

## 5.2c) 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 Bitweise 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

# PIC32 Family Reference Manual

**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 <sup>(5)</sup>	DIGEN6 <sup>(5)</sup>	DIGEN5 <sup>(5)</sup>	DIGEN4 <sup>(5)</sup>	DIGEN3 <sup>(5)</sup>	DIGEN2 <sup>(5)</sup>	DIGEN1 <sup>(5)</sup>	DIGEN0 <sup>(5)</sup>
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 <sup>(1,2,3,4)</sup>	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> <sup>(5)</sup>					

<b>Legend:</b>	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<sup>(5)</sup>

1 = ADC7 is digital enabled

0 = ADC7 is digital disabled

bit 22 **DIGEN6**: ADC6 Digital Enable bit<sup>(5)</sup>

1 = ADC6 is digital enabled

0 = ADC6 is digital disabled

bit 21 **DIGEN5**: ADC5 Digital Enable bit<sup>(5)</sup>

1 = ADC5 is digital enabled

0 = ADC5 is digital disabled

bit 20 **DIGEN4**: ADC4 Digital Enable bit<sup>(5)</sup>

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.

- 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.
- 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.
- 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.
- 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.

## Section 22. 12-bit High-Speed SAR ADC

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

bit 19 **DIGEN3**: ADC3 Digital Enable bit<sup>(5)</sup>

1 = ADC3 is digital enabled

0 = ADC3 is digital disabled

bit 18 **DIGEN2**: ADC2 Digital Enable bit<sup>(5)</sup>

1 = ADC2 is digital enabled

0 = ADC2 is digital disabled

bit 17 **DIGEN1**: ADC1 Digital Enable bit<sup>(5)</sup>

1 = ADC1 is digital enabled

0 = ADC1 is digital disabled

bit 16 **DIGEN0**: ADC0 Digital Enable bit<sup>(5)</sup>

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<sup>(1,2,3,4)</sup>

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.

---

## PIC32 Family Reference Manual

---

### 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<sup>(5)</sup>  
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.