B0 Wiederholung

Die Grundlagen der Assembler-Programmierung aus dem Modul A sollen anhand einiger Programmieraufgaben wiederholt werden.

Kurze Zusammenfassung Modul A

- Es existieren **32 Arbeitsregister** `r0-r31`, die sich auf den unteren Adressen im SRAM befinden. Mit ihnen lassen sich alle Berechnungen durchführen (Akkumulatorregister). Die obersten acht (`r24-r31`) Register können als 16 Bit Doppelregister `W`[1], `X` , `Y` und `Z` angesprochen werden. Die obersten sechs (`r24-r31`) Register können auch als Adresszeiger `X`, `Y` und `Z` zur indirekten Adressierung benutzt werden
  Die unmittelbare Adressierung (`ldi`) funktioniert nur bei den oberen 16 Register (`r16-r31`). Deshalb werden meist diese Register verwendet.
- **64 Sonderfunktions-Register** (SF-Register) ermöglichen die Ein- und Ausgabe von Daten (z.B. Ausgaberegister `PORTD`), die Steuerung der Peripherie (z.B. Datenrichtungsregister `DDRD`) und die Abfrage von Status-Informationen (z.B. Flagregister `SREG`). Sie können nicht unmittelbar adressiert werden. Zum Beschreiben und Lesen des ganzen Registers dienen die Befehle "`in`" und "`out`". Ein bitweises Schreiben der unteren 32 SF-Register ist mit den Befehlen "`sbi`", "`cbi`" möglich. Ein bitweises Überprüfen der unteren 32 SF-Register kann mit den Befehlen "`sbis`", "`sbic`" durchgeführt werden.
- Um ein neues Programm zu schreiben wird die **Assemblervorlage** (`A2_template.asm`) verwendet, da diese einige Initialisierungsarbeit abnimmt und ein einheitliches Erscheinungsbild der Lösungen ermöglicht.
- Um die Ein-/Ausgabe-Ports A-D benutzen zu können müssen diese zuerst initialisiert werden. Das **Datenrichtungsregister** `DDRx` (***D**ata **D**irection **R**egister*) legt fest ob es sich um einen Eingang oder Ausgang handelt. Es ist lese- und schreibbar. Das erwünschte Bit (Pin) wird mit `DDxn` bezeichnet.
- Über das **Datenausgaberegister** `PORTx` werden die Daten ausgegeben. Es ist lese- und schreibbar.
- Über das **Dateneingaberegister** `PINx` (***P**ort **IN**put Pins*) werden die Daten eingelesen. Es ist nur lesbar.
- **Nicht benutzte Pins werden als Eingang initialisiert!** Um Strom zu sparen sollte der interne Pull-Up-Widerstand zugeschaltet werden.
- Um den **internen Pull-Up-Widerstand** zu aktivieren wird im Ausgaberegister `PORTx` der entsprechende Pin auf Eins gesetzt.

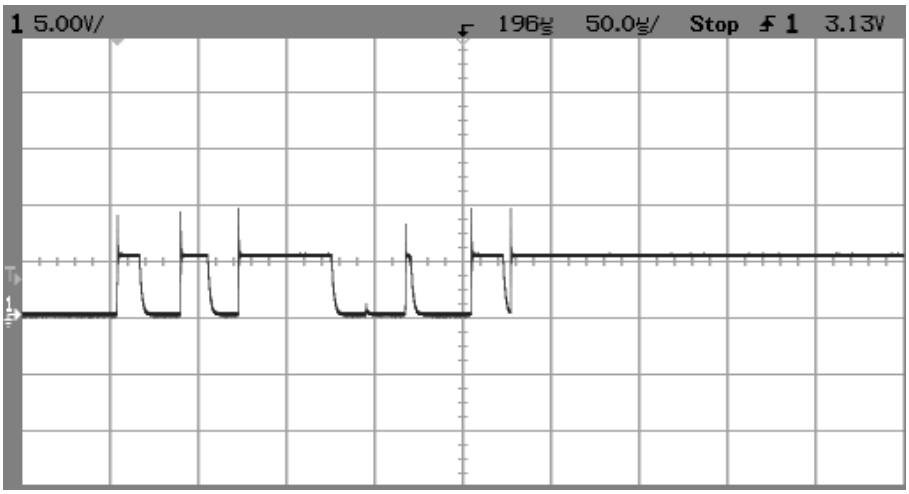
1 Das Doppelregister `W` wird in der Vorlage (A2_template.asm) definiert (`.DEF WL = r24; .DEF WH = r25`) und kann nicht als Adresszeiger verwendet werden. `X`, `Y` und `Z` sind in der AVR-Definitionsdatei (z.B.: m32def.inc) definiert. `r25` und `r24` werden auch zur Parameterübergabe bei Unterprogrammen genutzt.

- Die **Maskierung** mit logischen Funktionen erlaubt es gezielt einzelne Bits zu löschen, setzen oder umzuschalten (toggeln). Bei der **AND-Verknüpfung löscht** eine Null in der Maske das **Bit**. Bei der **OR-Verknüpfung setzt** eine Eins in der Maske das **`Bit`**. Bei der **XOR-Verknüpfung invertiert** eine Eins in der Maske das **Bit**.
  Die Maskierung wird besonders zur Manipulation der oberen 32 SF-Register benötigt.
- Ein Befehl besteht aus dem Opcode und den Operanden. Es gibt Befehle ohne Operand, welche wo nur ein Operand benötigt wird und Befehle mit zwei Operanden. Bei zwei Operanden steht immer **zuerst das Ziel und dann die Quelle!**
- **Zeitschleifen** sind Programme, deren Aufgabe es ist Zeit zu verbrauchen. Ändert man die Taktfrequenz des Controllers, so müssen in einem bestehenden Programm auch die Zeitschleifen verändert werden. Am einfachsten arbeitet man deshalb mit einer externen Unterprogramm-Sammlung (Zeitschleifen-Bibliothek, z.B.: **`SR_TIME_16M.asm`**) welche mit der Direktive "**`.INCLUDE`**" eingebunden wird.
- **Unterprogramme** sind Programmteile, die von einem Hauptprogramm aus (oder aus einem anderen Unterprogramm) aufgerufen werden und nach der Ausführung wieder an dieselbe Stelle des Hauptprogramms zurückspringen.
- **Unterprogramme können nur verwendet werden, wenn der Stapel vorher initialisiert wurde!** Register die ebenfalls im Hauptprogramm benötigt werden, müssen im Unterprogramm mit dem Befehl "**`push`**" gerettet werden und vor dem Verlassen des Unterprogramms mit dem Befehl "**`pop`**" wiederhergestellt werden. Ein Unterprogramm endet mit dem "**`ret`**"-Befehl.

Entprellen eines Tasters (Schalters)

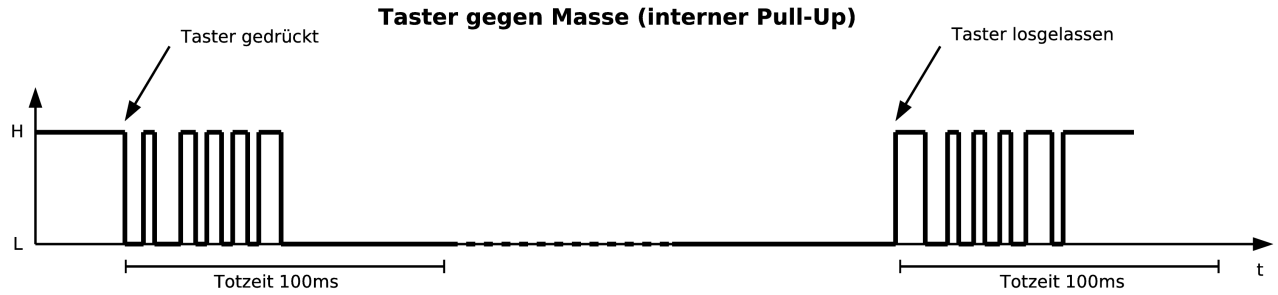
Wird ein Taster oder Schalter betätigt, so wird mechanisch ein Kontakt geschlossen. Hierbei federt der Kontakt mehrmals hin und her, das heißt der Kontakt wird mehrmals geschlossen und geöffnet bevor ein stabiler Zustand eintritt. Das Prellen ist vom Schaltertyp abhängig und dauert meist einige hundert Mikrosekunden, kann bei großen Schaltern aber bis zu 50 ms dauern! In dieser Prellzeit schließen und öffnen sich die Kontakte oft einige zehn- bis hundertmal!

Oszillogramm eines über etwa 250µs prellenden Tasters:



(Quelle: http://de.wikipedia.org/wiki/Prellen)

Das Prellen tritt auch beim Loslassen des Tasters auf:



Da Mikrocontroller einen Eingang mehrmals innerhalb von zehn Mikrosekunde abfragen können, besteht die Möglichkeit dass fälschlicherweise mehrere Änderungen der Eingangssignale erkannt werden, obschon der Schalter manuell nur einmal betätigt wurde.

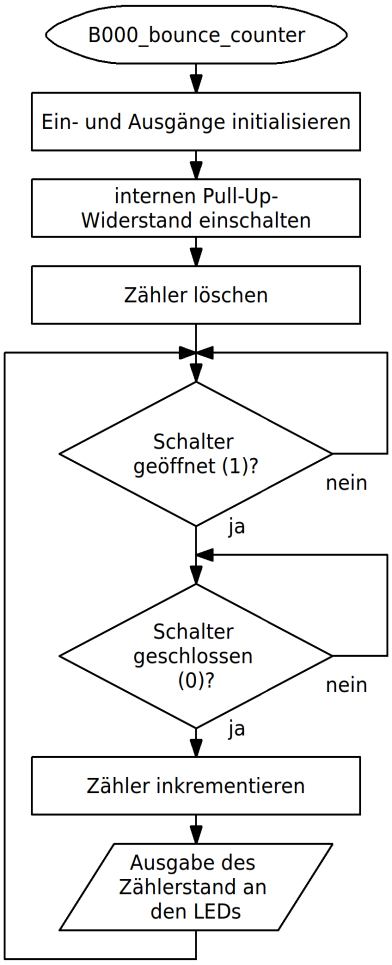
Durch Entprellung wird dieses Fehlverhalten vermieden. Die Entprellung kann hardwaremäßig oder softwaremäßig durchgeführt werden.

✎ **B000** a) Schreibe ein Programm, welches die mit einem gegen Masse geschalteten **Schalter** (Schließer, interner Pull-Up) an **PB0** erzeugten „Ein“-Schaltungen (1/0-Wechsel) beim Prellen zählt und laufend das Resultat binär an den LEDs von **PORTD** ausgibt. Das Flussdiagramm ist vorgegeben.

Nenne das Programm "**`B000_bounce_counter.asm`**".

b) Wie muss das Programm verändert werden, damit die „Aus“-Schaltungen (0/1-Wechsel) gezählt werden?

c) Teste das Programm ebenfalls mit einem hardwaremäßig entprellten Schalter.



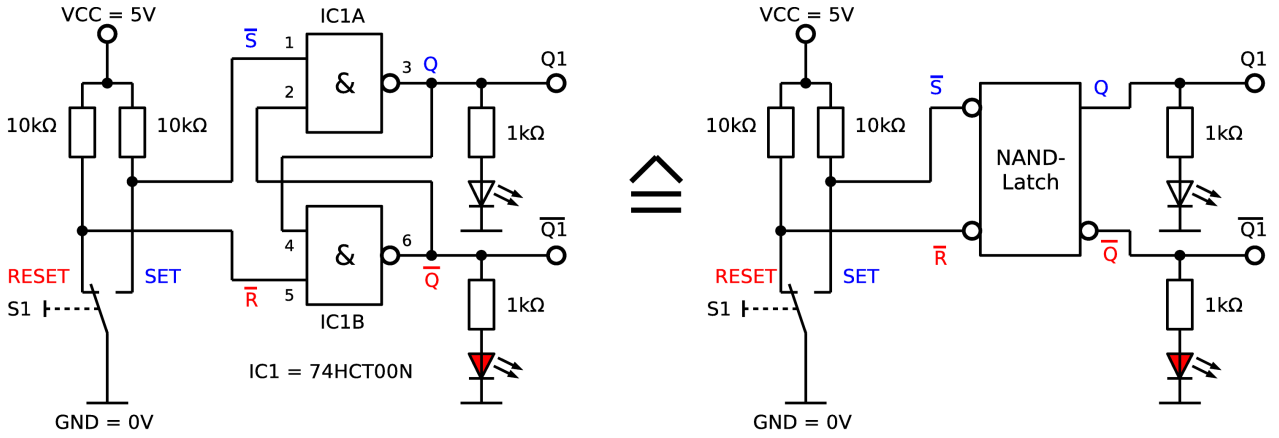
Bemerkungen: Der Zähler kann jeweils durch Drücken des RESET-Knopfes auf Null zurückgesetzt werden.
Die Taster des MICES2-Boards (im Gegensatz zum MICES-Board) prellen nur sehr wenig. Hier sollte mit unterschiedlichen externen Tastern und Schaltern experimentiert werden.

Hardwaremäßige Entprellung

Die beste Lösung um das Prellen eines Schalters zu verhindern ist eine zusätzliche Schaltung am Eingang zum Beispiel mit einem Wechseltaster oder -schalter und einem **Flip-Flop**[2], so dass der Schalterzustand gleich eindeutig ist (hardwaremäßiges Entprellen). Hier eine Schaltung mögliche Schaltung mit zwei NAND-Gliedern (NAND-FF, *NAND-Latch*).

2 Beim Flip-Flop (FF, bistabile Kippstufe) handelt es sich um eine Speicherzelle. Mit einer Eins am SET-Eingang wird ein FF gesetzt (Information gespeichert). Eine Eins am RESET-Eingang wird das FF rückgesetzt (Information gelöscht). Ein 8 Bit-Register besteht zum Beispiel aus 8 Flip-Flops.

Setzen und Rücksetzen wird hier mit einer logischen 0 erreicht. Die zwei Eingänge werden über Pull-Up-Widerstände auf definiertem Pegel gehalten wenn diese nicht mit Masse verbunden sind.



Eine Entprell-Schaltung mit einem einfachen Schließer mittels Tiefpass und Schmitt-Trigger ist auch möglich, hat allerdings den Nachteil, dass durch den Kondensator Verzögerungen beim Ein- und Ausschalten auftreten.

Softwaremäßige Entprellung

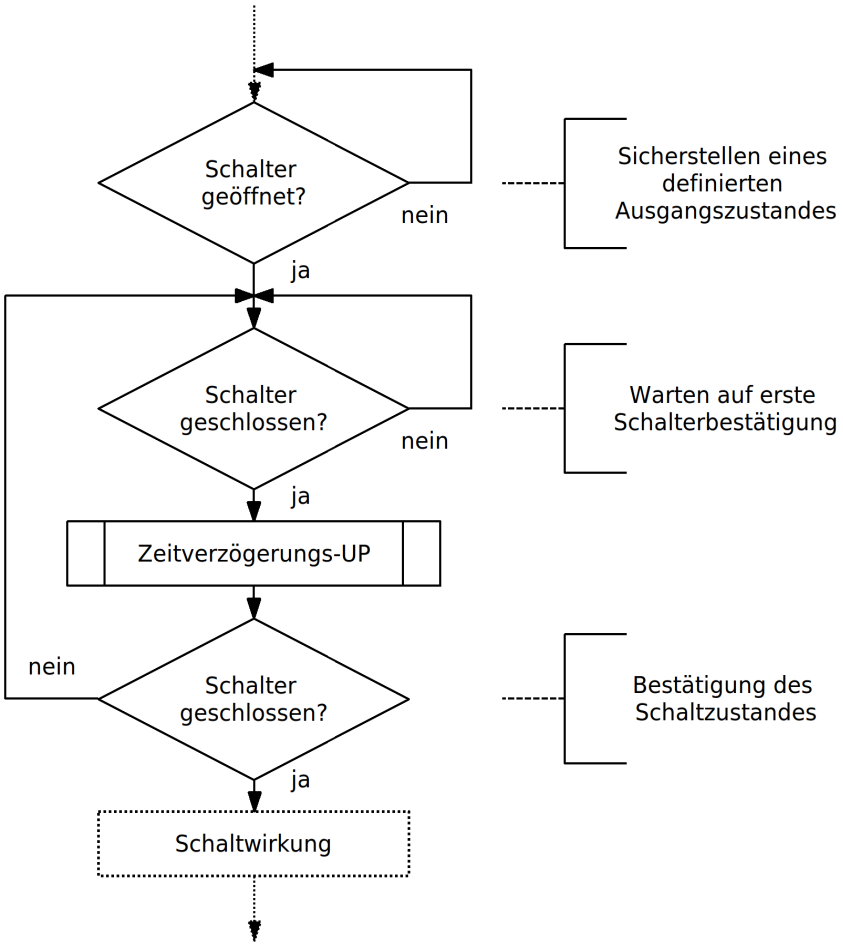
Zur softwaremäßigen Entprellung erzeugt man mit Hilfe einer Zeitschleifen eine Totzeit um das Prellen zu überbrücken. Nach der Sicherstellung eines definierten Anfangszustandes wird auf eine erste Schalterbetätigung gewartet. Ist diese erfolgt, wird nach Ablauf der Totzeit der Schalterzustand nochmals abgefragt, und bei erkanntem gültigen Schaltvorgang, fährt das Programm mit der gewünschten Schaltwirkung weiter.

Je nach Schaltertyp ergeben sich zwei verschiedene Möglichkeiten:

1. Der Schalter ist ein **Schließer:**
   Bei **Betätigung** wird der Stromkreis **geschlossen**.

2. Der Schalter ist ein **Öffner:**
   Bei **Betätigung** wird der Stromkreis **geöffnet**.

Das folgende Flussdiagramm zeigt das Prinzip der softwaremäßigen Entprellung eines Schließers S. Im Falle eines Öffners braucht nur die Fragestellung entsprechend geändert zu werden (geschlossen? → geöffnet?)



✎ **B001** a) Ändere die vorige Aufgabe (**B000**) mit Hilfe des obigen Flussdiagramms so um, dass das Prellen des Schalters softwaremäßig unterdrückt wird. Als Totzeit sollen 10 ms gewählt werden. Hierzu kann das Unterprogramm **`W10MS`** aus der Zeitschleifen-Bibliothek "**`SR_TIME_16M.asm`**" verwendet werden. Nenne das Programm "**`B001_debounced_counter_1.asm`**".

b) Ändere das unter a) erstellte Programm jetzt so um, dass das Entprellen in einem Unterprogramm erfolgt. Das Unterprogramm soll den Namen **`DEBSWC`** (*debounce switch closer*) erhalten und in der Datei "**`SR_DEBOUNCE.asm`**" abgespeichert werden. Das verwendete PIN-Register und die Pinnummer werden über die definierten Namen **`SWPinC`** und **`SwitchC`** übergeben (Mittels **`.EQU`** – Anweisung in der Bibliothek oder im Hauptprogramm definiert.
Bsp.: **`.EQU SWPinC = PINB`** oder **`.EQU SwitchC = 0`**). Dies ermöglicht es ein beliebiges Pin auszuwählen ohne das Unterprogramm verändern zu müssen.
Nenne das Programm "**`B001_debounced_counter_2.asm`**".

✎ **B002** Die Datei "**`SR_DEBOUNCE.asm`**" soll mit weiteren kommentierten Entprell-Unterprogrammen (dokumentiert mit entsprechenden Flussdiagrammen!) ergänzt werden um so als Bibliothek für spätere Programme genutzt werden zu können.
Soll die Bibliothek verwendet werden, so sind die verwendeten Schalter oder

Taster mit Masse zu verbinden und über Pull-Up-Widerstände an VCC anzuschließen.

a) Schreibe ein Unterprogramme zum Entprellen eines Öffners. Nenne das Unterprogramme **DEBSW0** (*debounce switch closer*). Übergabe des Pin mit **SWPin0** und **Switch0.**
b) Schreibe zwei weitere Unterprogramme zum Entprellen eines schließenden Tasters und eines öffnenden Tasters. Hier soll das Loslassen des Tasters ebenfalls berücksichtigt werden! Das obige Flussdiagramm muss also praktisch doppelt ausgeführt werden, einmal für das Betätigen und einmal für das Loslassen des Tasters (siehe Zeitdiagramm am Anfang des Kapitels)!
   Nenne die Unterprogramme **DEBPBC** (*debounce pushbutton closer*) und **DEBPB0** (Übergabe mit **PBPinC** und **PButtC** bzw. **PBPin0** und **PButt0**).
c) Teste die Unterprogramme mit einem Hauptprogramm das die Bibliothek einbindet und die Schalter-Betätigungen mit einem Zähler an den LEDs ausgibt. Nenne das Programm "**B002_debounced_counter_3.asm**".

Bemerkung: Sollen mehrere Schalter entprellt werden so wiederholt man die entsprechenden Unterprogramme (**DEBSWC1**, **DEBSWC2** ...).

Leuchtmustergenerator

Als weitere Wiederholungsaufgabe soll ein Leuchtmustergenerator programmiert werden:

✎ **B003** Zeichne ein Flussdiagramm und schreibe ein Programm, welches 4 verschiedene Leuchtmuster auf acht LEDs ausgeben kann (**PORTD**).
Nenne das Programm "**B003_christmas_1.asm**".
Die Leuchtmuster werden in vier unabhängigen Unterprogrammen verwaltet.

Zwei Schalter S3 und S2 (**PB3** und **PB2**) sind für den Zeitabstand zwischen den Ausgaben zuständig (damit Änderungen optisch sichtbar sind). Die Abfrage der Schalter geschieht im Hauptprogramm. Alle Schalter sind gegen Masse geschaltet und benötigen interne Pull-Up-Widerstände. Eine Entprellung ist in diesem Programm nicht nötig. Zur Zeitverzögerung wird die Zeitschleife **WW10MS** im Hauptprogramm verwendet. Die Anfangswerte für **W** (**r25**:**r24**) werden in vier Zeit-Unterprogrammen festgelegt werden.

```
S3 S2  Zeit    Name des Unterprogramms
-------------------------------------
 0 0   50ms         WMS50
 0 1   100ms        WMS100
 1 0   200ms        WMS200
 1 1   500ms        WMS500
```

Mit Hilfe zweier weiterer Schalter S0 und S1 (**PB0** und **PB1**) wird das erwünschte Leuchtmuster im Hauptprogramm ausgewählt (0 bedeutet hier

Schalter nicht betätigt!). Blinkmustern ändern dadurch, dass das Blinkmuster im Unterprogramm invertiert wird.

```
S1 S0  Aufgabe           Name des Unterprogramms    LEDs (* Ein, _ Aus)
-------------------------------------------------------------------
 0  0  Lauflicht Rechts          RUNR                     ** ->
 0  1  Lauflicht Links           RUNL                     <- **
 1  0  Blinkmuster 1            BLINK1              _*_*_*_* / *_*_*_*_
 1  1  Blinkmuster 2            BLINK2              ****____ / ____****
```

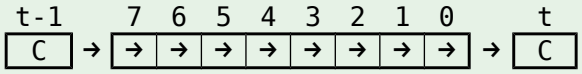
Bemerkung: Für das Lauflicht eignen sich die Befehle "**ror**" und "**rol**". Es muss aber eine Korrektur vorgenommen werden, sobald eine Eins zum Carry rausgeschoben wird.
Dies kann dadurch geschehen, dass das Flagregister **SREG** nach dem Rotieren gerettet wird und vor dem Rotieren wiederhergestellt wird, oder durch Setzen des höherwertigen Bits sobald Eins im Carry auftritt (siehe Befehle "**brcs**" oder "**brcc**").

ror Rd

Rechtsverschiebung durch Carry-Flag um eine Stelle (*rotate right through carry*).

1	0	0	1	0	1	0	d	d	d	d	d	0	1	1	1

Es erfolgt eine Rechtsverschiebung aller Bits im Register Rd um eine Stelle. Das bestehende Carry-Flag wird in Bit 7 geschoben. Bit 0 wird in das Carry-Flag hinein geschoben.



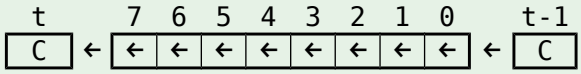
Beeinflusste Flags: S, V, N, Z, C Taktzyklen: 1

rol Rd

Linksverschiebung durch Carry-Flag um eine Stelle (*rotate left through carry*).

0	0	0	1	1	1	d	d	d	d	d	d	d	d

Es erfolgt eine Linksverschiebung aller Bits im Register Rd um eine Stelle. Das bestehende Carry-Flag wird in Bit 0 geschoben. Bit 7 wird in das Carry-Flag hinein geschoben.



Beeinflusste Flags: H, S, V, N, Z, C Taktzyklen: 1

brcc k

Bedingter relativer Sprung falls kein Übertrag (*branch if carry cleared*).(k = Adresskonstante)

1	1	1	1	0	1	k	k	k	k	k	k	0	0	0

Der Befehl wird meist gleich nach einer arithmetischen oder logischen Operation eingesetzt. **Der Sprung erfolgt falls kein Übertrag (Carry) aufgetreten ist** (**C**-Flag = 0).
Beeinflusste Flags: keine
Taktzyklen: 1 (kein Sprung), 2 (Sprung)

brcs k

Bedingter relativer Sprung falls Übertrag (*branch if carry set*).(k = Adresskonstante)

1	1	1	1	0	0	k	k	k	k	k	k	0	0	0

Der Befehl wird meist gleich nach einer arithmetischen oder logischen Operation eingesetzt. **Der Sprung erfolgt falls ein Übertrag (Carry) aufgetreten ist** (**C**-Flag = 1).
Beeinflusste Flags: keine
Taktzyklen: 1 (kein Sprung), 2 (Sprung)

Aufgaben

Zur weiteren Wiederholung können zusätzlich folgende Aufgaben gelöst werden:

✎ **B004** Schreibe ein Programm (inkl. Flussdiagramm) für einen Vorwärtszähler, dessen Schrittweite an acht Schaltern während des Zählens eingestellt werden kann und dessen Stand auf acht Leuchtdioden ausgegeben wird.

- Der Zähler soll mit einer Geschwindigkeit von einem Schritt pro halbe Sekunde zählen.
- Der Anfangswert des Zählers beträgt 0.
- Die Schalterabfrage und die Ausgabe an die LEDs soll als Unterprogramm realisiert werden.

Nenne das Programm "**`B004_up_counter.asm`**".

✎ **B005** Schreibe ein Programm (inkl. Flussdiagramm) für einen Vorwärts-/Rückwärtszähler, dessen Schrittweite während des Zählens an sieben Schaltern (S0-S6) eingestellt werden kann und dessen Stand auf die acht Leuchtdioden ausgegeben wird.

- Ein achter Schalter S7 entscheidet über die Zählrichtung und hat keinen Einfluss auf die Schrittweite.
- Wenn S7 = 0, soll der Zähler vorwärts zählen und für S7 = 1 rückwärts.
- Der Zählvorgang beginnt beim Zählerstand 0x10 und die Zählge schwindigkeit soll 1 Schritt pro 1,2 Sekunden betragen.
- Die Schalterabfrage und die Ausgabe an die LEDs soll als Unterprogramm realisiert werden.

Nenne das Programm "**`B005_up_down_counter.asm`**".