Skriptum Mikrocontroller

Teil 3 Bitmanipulation

DI Wolfgang Zukrigl, HTL Litec
DI(FH) Andreas Pötscher, HTL Litec

Inhaltsverzeichnis

Datum	Beschreibung der durchgeführten Änderung	Autor
22.02.2019	V2.2 . . . Dieser Teil wurde basierend auf Version 1.0 neu erstellt	Wolfgang Zukrigl, HTL Linz–Paul-Hahn-Straße (LiTec)
01.04.2020	V2.3 . . . direktes Programmieren der Special Function Register ausführlicher erklärt	Wolfgang Zukrigl, HTL Linz–Paul-Hahn-Straße (LiTec)
29.08.2024	V2.3 . . . übernahme im Markdown und allgemeine Überarbeitung	Andreas Pötscher, HTL Linz–Paul-Hahn-Straße (LiTec)

Notwendigkeit für Bitmanipulation

Im vorigen Teil dieser Skriptenreihe („Skriptum-GPIOs_Teil-2“) wurde gesagt, dass in der Programmiersprache C Special-Function-Register immer nur Byte-weise angesprochen werden können.

Im Falle der Register DDRn, PORTn und PINn bedeutet das, dass immer nur alle acht Pins eines Ports gemeinsam modifiziert werden können. Beispielsweise werden durch den Befehl DDRB=0xF0; alle 8 Pins PB7 bis PB0 teils zu Eingängen, teils zu Ausgängen gemacht.

Sehr oft möchte man aber nur ein oder zwei Pins eines Ports modifizieren, ohne den Zustand der anderen Pins dieses Ports zu verändern. Man möchte also einzelne Bits in einem Special-Functionregister gezielt mit einer eins oder eine null beschreiben, ohne die anderen Bits dieses Registers zu verändern. Mit folgendem Befehl wird z.B. das Bits Nr. 4 im DDRB-Register auf 1 gesetzt, wodurch der Pin PB4 ein Ausgang wird:

```
DDRB |= (0x01<<4);
```

Die anderen Bits im DDRB-Register bleiben durch den Befehl aber unverändert, wodurch die an-deren Pins von Port B (PB0 bis PB3 und PB5 bis PB7) Ein- oder Ausgänge bleiben – je nachdem was sie vor Ausführung des Befehls waren.

Bei gezielter Veränderung einzelner Bits in einer Variable oder einem Special-Function-Register (SFR), ohne Veränderung der restlichen Bits, spricht man von Bitmanipulation.

Die nötigen Operatoren der Programmiersprache C

Bitweise Verknüpfungen

Die Operatoren `&`, `|`, `^` und `~` sind bitweise Operatoren. (Im Gegensatz zu den logischen Operatoren `&&`, `||` und `!`, die bei Bedingungen in Schleifen oder if-Verzweigungen zur Anwendung kommen). Jedes Bit des einen Operanden wird mit dem entsprechenden Bit des anderen Operanden verknüpft.

Beim Verändern von einzelnen Bits einer Variable wird diese meist mit einer zweiten Variable mit einer bitweisen Verknüpfung verarbeitet. Die zweite Variable die zur Veränderung dient heißt Bitmaske oder kurz Maske.

Am besten lassen sich die Operatoren anhand eines Beispiels erklären. Die Variable `a` soll verändert werden. Die Variable `m` ist die Maske.

```
//Stelle                //7654 3210
uint8_t a = 0xA5;       //1010 0101
uint8_t m = 0x0F;       //0000 1111
```

Beim bitweisen not `~` werden alle Bits einzeln invertiert. Aus 0 wird 1 und aus 1 wird 0.

```
//a                     //1010 0101
uint8_t notA = ~a;      //0101 1010
```

Beim bitweisen und `&` werden alle Bits einzeln und verknüpft. Es müssen beide Bits 1 sein damit das Ergebnis 1 ist.

```
//a                         1010 0101
//m                         0000 1111
uint8_t aAndM = a&m;      //0000 0101
```

Beim bitweisen oder `|` werden alle Bits einzeln oder verknüpft. Es muss eines der beiden Bits 1 sein damit das Ergebnis 1 ist.

```
//a                         1010 0101
//m                         0000 1111
uint8_t aOrM = a|m;       //1010 1111
```

Beim bitweisen exklusiv-oder `^` werden alle Bits einzeln exklusiv-oder verknüpft. Es müssen beide Bits ungleich sein damit das Ergebnis 1 ist.

```
//a                         1010 0101
//m                         0000 1111
uint8_t aXOrM = a^m;      //1010 1010
```

> **Wichtig!**
>
> - *`a & m` beim bitweisen und werden alle Bits die in der Maske 0 sind im Ergebnis 0. Alle anderen Bits bleiben unverändert.*
> - *`a | m` beim bitweisen oder werden alle Bits die in der Maske 1 sind im Ergebnis 1. Alle anderen Bits bleiben unverändert.*
> - *`a ^ m` beim bitweisen xor werden alle Bits die in der Maske 1 sind im Ergebnis invertiert. Alle anderen Bits bleiben unverändert.*

Schiebebefehle

Um die für die Bitmanipulation benötigte Maske in möglichst gut lesbarer Form erzeugen zu können, sind Schiebebefehle nötig. Diese können wieder anhand von Beispielen am besten erklärt werden.

Beim Linksschieben oder Shift-Left `<<` werden alle Bits um eine bestimmte Anzahl von Stellen nach links geschoben. Auf der rechten Seite wird mit 0 aufgefüllt.

```
//Stelle                  //7654 3210
uint8_t a = 0xC7;         //1100 0111
uint8_t result = a << 1; //1000 1110
```

Die Variable `a` erhält zuerst den Wert `11000111`. In der letzten Zeile wird der Variablen result der um eine Stelle nach links geschobene Inhalt von `a` zugewiesen. Dabei wird das in `a` abgespeicherte Bitmuster verschoben:

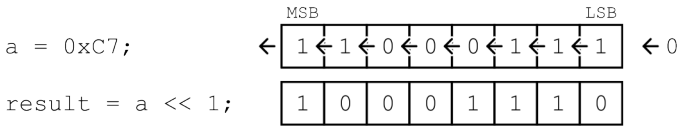

Abbildung 1: Shift Left

Jedes Bit des Bitmusters `11000111` wir um eine Stelle nach links geschoben. In das unterste Bit wird eine „0“ geschoben, das oberste Bit wird „herausgeschoben“ und geht verloren. Das Bitmuster, das schlussendlich in result abgespeichert wird, ist demnach `10001110`.

Beim Rechtssschieben oder Shift-Right `>>` werden alle Bits um eine bestimmte Anzahl von Stellen nach rechts geschoben. Auf der linken Seite wird mit 0 aufgefüllt.

```
//Stelle                  //7654 3210
uint8_t a = 0xC7;         //1100 0111
uint8_t n = 2;
uint8_t result = a >> n; //0011 0001
```

Hier wird der Variablen `a` ebenfalls zuerst der Wert `0xC7`, also das Bitmuster `11000111`, zugewiesen, und der Variablen number der Wert 2. Das Ergebnis von `a >> number` ist das um 2 Stellen nach rechts verschobene Bitmuster von `a`. An höchster Stelle werden dabei 0-Bits „hereingeschoben“, die an unterster Stelle „herausgeschobenen“ Bits gehen verloren. Dadurch erhält die Variable `result` das Bitmuster `00110001`:

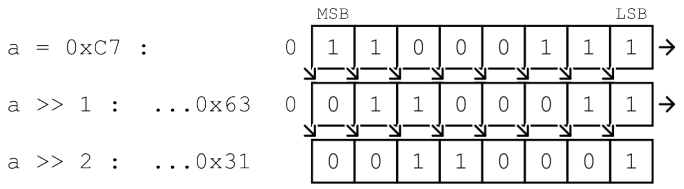

Abbildung 2: Shift Right

ACHTUNG: Wäre a nicht unsigned, sondern signed, so würden an höchster Stelle keine „0“-Bits hereingeschoben, und das Ergebnis wäre ein anderes (Arithmetisches Schieben). Verwenden Sie daher für Variablen, die ein Bitmuster darstellen (und keine Zahl) immer den Datentyp-Spezifizierer unsigned! Bei Verwendung der Datentypen aus dem Header stdint.h verwenden Sie also die mit „u“ beginnenden Datentypen, z.B: uint8_t. Wenn Sie stattdessen die Standard-C-Datentypen verwenden wollen, müssen die Variablen vom Typ unsigned char, unsigned int oder unsigned long sein.

> **Wichtig!**
>
> - `x << y` *liefert als Ergebnis das um y Stellen nach links geschobene Bitmuster von x zurück.*
> - `x >> y` *liefert als Ergebnis das um y Stellen nach rechts geschobene Bitmuster von x zurück.*
> - *Die Variable x sollte ein unsigned-Typ sein.*

Bitmanipulation mit der Programmiersprache C

Setzen von Bits

Setzen eines Bits bedeutet, dass ein Bit mit einer logischen eins beschrieben wird.

> **Wichtig!**
>
> *Das Setzen (auf logisch 1 setzen) von Bits erfolgt mittels bitweiser Oder-Verknüpfung mit einer Maske. In der Maske sind die zu setzenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.*

Erklärung anhand eines Beispiels: Die GPIO-Pins PB6, PB2 und PB1 sollen als Ausgänge programmiert werden. Die Richtung der anderen GPIO-Pins von Port B soll dabei nicht verändert werden. (War beispielsweise PB0 als Eingang programmiert, so soll er ein Eingang bleiben, war er als Ausgang programmiert, so soll er ein Ausgang bleiben.) Das bedeutet, dass die Bits Nr. 6, 2 und 1 vom Register DDRB auf „1“ gesetzt werden sollen, die anderen Bits (Nr. 7, 5, 4, 3, 0) aber unverändert bleiben.

Die folgenden zwei Zeilen Code bewerkstelligen das:

```
uint8_t mask = 0x46       //0100 0110
DDRB |= mask;
```

Zur Erinnerung: `DDRB |= mask` ist eine abgekürzte Schreibweise für `DDRB = DDRB | mask;`. Es ergeben sich folgende Bitmuster:

```
//DDRB                       xxxx xxxx
//mask 0x46                  0100 0110
//DDRB (nach DDRB |= mask)   x1xx x11x
```

In der Tabelle oben steht `x.` für entweder 1 oder 0. Es handelt sich um den Inhalt des jeweiligen Bits des DDRB-Registers vor Ausführen der beiden Befehlszeilen.

Die Bits 6, 2 und 1 in `mask` sind „1“. Egal ob die entsprechenden Bits in `DDRB` vorher „0“ oder „1“ waren, der neue Inhalt dieser Bits von `DDRB` ist „1“. Bei einer Oder-Verknüpfung muss nur einer der Operanden „1“ sein, damit das Ergebnis „1“ ist.

Die anderen Bits von `mask` (7, 5, 4, 3, 0) sind „0“. Die entsprechenden DDRB-Bits bleiben unverändert. Nehmen Sie an, dass Bit 7 in `DDRB` vorher „0“ war. Die Oder-Verknüpfung „0 | 0“ liefert „0“ und Bit 7 von `DDRB` ist nach der Oder-Verknüpfung somit immer noch „0“. War das Bit 7 in `DDRB` hingegen vorher „1“, so liefert die Oder-Verknüpfung „1 | 0“ eine „1“ und das Bit 7 in `DDRB` ist nach der Oder-Verknüpfung wieder gleich „1“. In beiden Fällen bleibt das Bit 7 von `DDRB` so wie es vorher war. Das gilt auch für alle anderen DDRB-Bits, bei denen das entsprechende Bit von mask gleich „0“ ist.

Um es Menschen, die den C-Sourcecode lesen, einfacher zu machen, werden Bitmasken meist in einer anderen Form angegeben, in der die „1“-Bits der Maske direkt ablesbar sind. Die folgenden beiden Zeilen Code sind völlig identisch mit dem oben besprochenen Code:

```
uint8_t mask = (0x01<<6) | (0x01<<2) | (0x01<<1);
DDRB |= mask;
```

Die Maske ist nun eine Oder-Verknüpfung von drei Operanden `(0x01<<6)`, `(0x01<<2)` sowie `(0x01<<1)`. Es wird also jeweils das Bitmuster `00000001` (`0x01`, dezimal 1) um 6, 2 bzw. 1 Stelle nach links geschoben, so dass daraus die Bitmuster `01000000`, `00000100` sowie `00000010` werden. Nach der Oder-Verknüpfung dieser drei Bitmuster ist der Inhalt von mask also `01000110` oder `0x46`.

```
//0x01                      0000 0001
//0x01 << 6                 0100 0000
//0x01 << 2                 0000 0100
//0x01 << 1                 0000 0010
//0x01<<6|0x01<<2|0x01<<1   0100 0110
```

In der Praxis werden die obigen beiden Programmzeilen zu einer einzelnen Zeile zusammengefasst:

```
DDRB |= (0x01<<6) | (0x01<<2) | (0x01<<1);
```

Löschen von Bits

Löschen von Bits bedeutet, in diese Bits eine logische null zu schreiben.

> **Wichtig!**
>
> *Das Löschen (Rücksetzen, auf logisch 0 setzen) von Bits erfolgt mittels bitweiser Und-Verknüpfung mit dem Einser-Komplement (Negation, Not-Verknüfung, Invertierung) einer Maske. In der Maske sind die zu löschenden Bits gleich 1, die nicht zu verändernden Bits gleich 0.*

Erklärung anhand eines Beispiels: Die GPIO-Pins PB6, PB2 und PB1 sollen als Eingänge programmiert werden. Die Richtung der anderen GPIO-Pins von Port B soll dabei nicht verändert werden. Das bedeutet, dass die Bits 6, 2 und 1 vom Register DDRB auf „0“ gesetzt werden sollen, die anderen Bits (7, 5, 4, 3, 0) aber unverändert bleiben. Mit den folgenden beiden Zeilen Code kann das erreicht werden:

```
uint8_t mask = (0x01<<6)|(0x01<<2)|(0x01<<1);
DDRB &= ~(mask);
```

Die Maske `mask` ist wieder gleich wie beim vorigen Beispiel, also `01000110` oder `0x46`. Durch das Bilden des Einser-Komplements mittels des Operators „~“ und der Und-Verknüfung ergibt sich das gewünschte Ergebnis:

```
//mask 0x46                 0100 0110
//~mask 0xB9                1011 1001
//DDRB                      xxxx xxxx
//DDRB(nach DDRB |= ~mask)  x0xx x00x
```

Der Einser-Komplementoperator (Not-Verknüpfung, Invertierung, Operator „~“) erzeugt aus mask das Bitmuster `10111001`. Die Bits 6, 2 und 1 von ~mask sind gleich „0“. Das Ergebnis der Und-Verknüpfung `DDRB & ~mask` liefert daher in den Bits 6, 2 und 1 eine „0“, da bei einer &-Verknüpfung nur einer der beiden Operanden „0“ sein muss, damit das Ergebnis „0“ ist.

Die anderen Bits (7, 5, 4, 3, 0) von `mask` sind gleich „0“ und somit jene von `~mask` gleich „1“. War zum Beispiel dieser Bit 3 in `DDRB` vorher „0“, so ist das Bit 3 in `DDRB & ~mask` ebenfalls gleich „0“, weil 0 & 1 = 0 ist. War Bit 3 in `DDRB` hingegen vorher „1“, so ist es auch nachher „1“, weil das Ergebnis von `DDRB & ~mask` im Bit 3 gleich „1 & 1 = 1“ ergibt. Bei allen Bits, bei denen mask gleich „0“ und somit `~mask` gleich „1“ ist, ist der Inhalt von DDRB nach der Operation `DDRB &= ~mask`; also gleich dem Inhalt von DDRB vor dieser Operation. Diese Bits bleiben unverändert.

Die beiden Zeilen Code lassen sich wieder zu einer einzigen zusammenfassen:

```
DDRB &= ~((0x01<<6) | (0x01<<2) | (0x01<<1));
```

Dabei sind die Klammern um die gesamte Bitmaske äußerst wichtig! Werden sie weggelassen, so wird das Einserkomplement nur von (0x01«6) gebildet, und erst danach die Oder-Verknüpfungen mit (0x01«2) und (0<01«1) durchgeführt.

Toggeln von Bits

Unter Toggeln eines Bits versteht man, das Bit zu ändern. Ist das Bit vor dem Toggeln „0“ so ist es nach dem Toggeln „1“. Ist das Bit vor dem Toggeln hingegen „1“, so ist es nachher „0“.

Im ersten Ansatz würde man den aktuellen Zustand eines Bits einlesen, und entsprechend dem Ergebnis mit einer if-Abfrage das Bit entweder setzen oder löschen. Diese Vorgehensweise müsste man für alle zu toggelnden Bits eines Registers wiederholen. Mit der Exklusiv-Oder-Verknüpfung funktioniert das Toggeln aber viel einfacher.

> **Wichtig!**
>
> *Das Toggeln von Bits erfolgt mittels bitweiser XOR-Verknüpfung mit einer Maske. In der Maske sind die zu toggelnden Bits gleich 1, die nicht zu verändernden Bits gleich 0.*

Erklärung anhand eines Beispiels: Die GPIO-Pins PB6, PB2 und PB1 sind bereits als Ausgänge programmiert worden und geben bestimmte Spannungspegel aus. Diese drei Ausgänge sollen gleichzeitig getoggelt werden. Wenn ein Ausgang vorher einen Low-Pegel ausgegeben hat, so soll er einen High-Pegel ausgeben. Wenn er vorher einen High-Pegel ausgegeben hat, so soll er einen Low-Pegel ausgeben. Das bedeutet, dass die Bits 6, 2 und 1 vom Register `PORTB` getoggelt werden sollen. Die anderen Bits (7, 5, 4, 3, 0) von `PORTB` sollen dabei unverändert bleiben. Mit den folgenden beiden Zeilen Code kann das erreicht werden:

```
uint8_t mask = (0x01 << 6) | (0x01 << 2) | (0x01 << 1);
PORTB ^= mask;
```

Die Maske mask ist wieder gleich wie beim vorigen Beispiel, also `01000110` oder `0x46`. Die Exklusiv-Oder-Verknüpfung führt zum gewünschten Ergebnis:

```
//mask 0x46                 0100 0110
//PORTB                     xxxx xxxx
//PORTB (nach PORTB|=mask) x!xx x!!x
```

In der Tabelle bedeutet, `x` den alten Inhalt des Bits und `!` die Invertierung dieses alten Inhalts.

In der Maske `mask` ist das Bit 7 gleich „0“, daher bleibt Bit 7 in `PORTB` erhalten:

- War das Bit 7 in `PORTB` vorher „0“ so liefert die Exklusiv-Oder-Verknüpfung `PORTB ^ mask` ebenfalls „0“, da `0 ^ 0= 0`.
- War das Bit 7 in `PORTB` vorher „1“ so liefert die Exklusiv-Oder-Verknüpfung `PORTB ^ mask` ebenfalls „1“, da `1 ^ 0= 1`.

In der Maske `mask` ist das Bit 6 gleich „1“, daher wird Bit 6 in `PORTB` invertiert (getoggelt):

- War das Bit 6 in `PORTB` vorher „0“ so liefert die Exklusiv-Oder-Verknüpfung `PORTB ^ mask` „1“, da `0 ^ 1 = 1`.
- War das Bit 6 in `PORTB` vorher „1“ so liefert die Exklusiv-Oder-Verknüpfung `PORTB ^ mask` „0“, da `1 ^ 1 = 0`.

Jene Bits von PORTB, bei denen die entsprechenden Bits in mask also „0“ sind, bleiben unverändert. Jene Bits von PORTB, bei denen die entsprechenden Bits in mask hingegen „1“ sind, werden getoggelt.

Die beiden obigen Zeilen Code würden in einer tatsächlichen Anwendung typischerweise zu einer einzelnen Zeile zusammengefasst:

```
PORTB ^= (0x01<<6) | (0x01<<2) | (0x01<<1);
```

Testen von Bits

Anstelle von „Testen“ spricht man auch vom „Einlesen“ von Bits. Das gleichzeitige Testen mehrerer Bits wird in der Praxis selten benötigt, daher wird hier nur das Testen einzelner Bits besprochen. Meistens möchte man in Abhängigkeit davon, ob ein bestimmtes Bit in einem Special-Function-Register oder in einer Variable logisch 1 bzw. logisch 0 ist, einen Anweisungsblock ausführen, oder eben nicht ausführen. Es handelt sich also um eine if-Anweisung.

Verwenden des Bits direkt in einer Verzweigung

Das Testen erfolgt so, dass die nicht zu testenden Bits ausmaskiert (auf 0 gesetzt) werden.

Erklärung anhand eines Beispiels – Testen auf „1“: Am Pin PB4 sind ein Taster und ein externer pull-down-Widerstand angeschlossen. Der Port-Pin wurde bereits als Eingang mit deaktiviertem internen pull-up-Widerstand programmiert. Mit einer if-Abfrage soll überprüft werden, ob der Taster gedrückt ist, oder nicht. Wenn der Taster gedrückt ist, liegt am Pin PB4 ein High-Spannungspegel an, und Bit 4 des Registers `PINB` ist somit „1“. Bei offenem Taster ist der Spannungspegel an PB4 Low und Bit 4 von `PINB` ist „0“.

Der folgende Code liefert das gewünschte Ergebnis:

```
if(PINB & (0x01 << 4))
{
    //Code Abschnitt der ausgeführt wird, wenn an PIN PB4 ein High Pegel
    //anliegt und somit Bit 4 vom Register PINB gleich 1 ist.
}
else
{
    //Code Abschnitt der ausgeführt wird, wenn an PIN PB4 ein Low Pegel
    //anliegt und somit Bit 4 vom Register PINB gleich 0 ist.
}
```

Die verwendete Maske `(0x01<<4)` liefert das Bitmuster `0001000`. Das Ergebnis der UND-Verknüpfung `PINB & (1<<4)` ist somit ein Bitmuster, bei dem alle Bits außer dem vierten, sicher „0“ sind:

```
//PINB                      xxxx xxxx
//Maske (0x01 << 4)         0001 0000
//PINB & (0x01 << 4)        000x 0000
```

Ist das Bit 4 von `PINB` (also der Wert Bit 4) gleich „0“ (Taster nicht gedrückt), so liefert die Und-Verknüpfung als Ergebnis das Bitmuster `00000000`, also dezimal 0, wodurch der else-Zweig zur Ausführung kommt. Ist das Bit 4 von `PINB` hingegen „1“ (Taster gedrückt), so liefert die Und-Verknüpfung als Ergebnis das Bitmuster `00010000` (`0x10`, dezimal `16`). Jede Zahl ungleich null bedeutet innerhalb einer Bedingung ein logisches „wahr“ („true“), und somit kommt der if-Zweig zur Ausführung.

Weiteres Beispiel: Der Taster ist nun zwischen dem Pin PB4 und GND angeschlossen, es wird der interne pull-up-Widerstand verwendet. Für die Initialisierung müssen daher Bit 4 von `DDRB` auf „0“ und Bit 4 von `PORTB` auf „1“ gesetzt werden. Ein bestimmter Code-Abschnitt soll nur ausgeführt werden, wenn der Taster gedrückt ist, wenn also Bit 4 von `PINB` gleich „0“ ist. Ist der Taster nicht gedrückt, so soll dieser Code-Abschnitt einfach übersprungen werden.

```
if((PINB & (0x01 << 4 )) == 0)
{
    //Code Abschnitt, der ausgeführt wird, wenn an PIN PB4 ein Low Pegel
    //anliegt und somit Bit 4 in PINB gleich 0 ist.
}
```

Der Inhalt des inneren Klammerpaares `PINB & (0x01<<4)` ist, wie beim vorigen Beispiel, bei gedrücktem Taster gleich `00000000`=0 und bei nicht gedrücktem Taster `0001000` = `16`. Durch den Vergleich mit null, also `(PINB & (0x01<<4))==0`, wird der Code-Abschnitt ausgeführt, wenn Bit Nr 4 im PINB-Register logisch null ist. Das heißt der Code-Abschnitt wird bei gedrücktem Taster ausgeführt.

Die Klammernsetzung im obigen Code-Abschnitt ist extrem wichtig: Der `==` Operator hat eine höhere Priorität als der `&` Operator. Beim Weglassen der Klammern, also `if ( PINB & (0x01<<4) == 0 )` würde der Compiler das als if `( PINB & ((0x01<<4) == 0) )` interpretieren, was immer zu `if (0x00)` ausgewertet wird. Der Compiler gibt in diesem Fall eine Warning aus („Suggest parenthesis around expression . . . “)

> **Wichtig!**
>
> *Das Testen von Bits erfolgt im einfachsten Fall mittels*
>
> `if ( Bitmuster & (0x01<<BitNummer) ) ...` *bzw. mit*
>
> `if ( ( Bitmuster & (0x01<<BitNummer) ) == 0 ) ...`

Speichern in einer Variablen

Um das Programm übersichtlich und wartbar zu gestalten empfiehlt sich die Zustände der GPIO Pins nach dem Einlesen direkt in eine Variable zu speichern. Nach dem vorher beschriebenen Muster sieht das so aus.

```
uint8_t bit = PINB & (0x01 << 4);
```

Die Variable bit ist je nach dem Zustand von Bit 4 von `PINB` nach diesen Zeilen entweder `00000000=0`, also logisch falsch, oder `00010000=16`, also ungleich null und somit logisch wahr.

Oft ist es im weiteren Programm praktisch nicht die Zahlenwerte 0 und 16, sondern exakt die Zahlenwerte 0 bzw. 1 abzuspeichern. Das kann man auf mehrere Arten tun: Entweder man schiebt das Ergebnis von PINB & (1«4) wieder vier Stellen nach rechts, oder man verwendet den ternären Bedingungsoperator. Als dritte Alternative könnten man eine if-else-Anweisung verwenden.

Eine sehr kurze und elegante Variante ist das verschieben des Registers, dass das gesuchte Bit an der Stelle 0 steht und dann mit `0x01` ausmaskiert wird. Damit ist bei gesetzten Bit das Ergebnis 1 und bei nicht gesetzten Bit 0.

```
uint8_t bit = PINB >> 4 & 0x01;
```

zur Erklärung. Das Bit b wird gesucht:

```
//PINB                    xxxb xxxx
//Nach PIN B >> 4         0000 xxxb
//Nach & 0x01             0000 000b
```

Das gesuchte Bit wird mit einen Schiebeoperator `>>` an die Stelle 0 geschoben. Dann wird mit `0x01 = 0000 0001` ausmaskiert. Dabei werden alle Bits außer das Bit 0 auf 0 gesetzt. Das Ergebnis ist dann dezimal entweder 0 oder 1 und lässt sich somit im Programm sehr gut weiter verarbeiten.

Zusammenfassung

Zusammenfassend ist festzuhalten, dass es 4 wichtige Code-Snippets für das bitweise verarbeiten von SFRs gibt. Das wird am Beispiel von `PORTA` und `PINA` noch mal gezeigt. Es gilt aber natürlich auch für alle anderen Register.

Setzen von Bits

```
uint8_t bitNo;
PORTA |= 0x01 << bitNo;
```

Löschen von Bits

```
uint8_t bitNo;
PORTA &= ~(0x01 << bitNo);
```

Toggeln von Bits

```
uint8_t bitNo;
PORTA ^= 0x01 << bitNo;
```

Auslesen von Bits

```
uint8_t bitNo;
uint8_t bit = PINA >> bitNo & 0x01;

if(bit)
{
    //...
}
```