PROGC U01 Bit Operatoren

Inhalt

1 Einführung

In dieser Übung verwenden Sie die Bit Operatoren um einzelne Bits eines Wertes zu setzen, zu löschen oder zu invertieren.

2 Lernziele

- Sie kennen die Funktionsweise der vier logischen Bit Operatoren
  - NOT (~), AND (&), Inklusiv-OR (| , auch OR genannt), Exklusiv-OR (^ , auch XOR genannt)
- Sie kennen die Funktionsweise der zwei Bit Shift Operatoren
  - Left-Shift (<<), Right-Shift (>>)
- Sie können die Bit Operatoren anwenden um
  - gezielt spezifische Bit Werte abzufragen
  - individuelle Bits zu setzten, zu löschen, zu invertieren während andere Bits unverändert bleiben
  - initiale Bit Muster einfach zu setzen

3 Übungen

3.1 Was macht folgende Code Sequenz

Bitte füllen Sie die betreffenden Bit Werte in der Tabelle ein für den Aufruf

```
uint16_t reg = 0xAAAA;
reg = SetBitsAt(reg, 0x5, 6, 4);
```

```
uint16_t SetBitsAt(
  uint16_t value,
  uint16_t bitPattern,
  size_t nBits,
  size_t position)
{
  uint16_t clr = 1u << nBits;
  clr -= 1;
  bitPattern &= clr ;
  clr <<= position;
  clr = ~clr;
  bitPattern <<= position;
  value &= clr;
  value |= bitPattern;
  return value;
}
```

value:	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
bitPattern:	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
clr:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
clr:	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
bitPattern :	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
clr:	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0
clr:	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1
bitPattern:	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
value:	1	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0
value:	1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	0
value: (hex)	0xA85A															

Was macht diese Funktion?

Sie ersetzt im Wert **value** die Anzahl **nBits** Bits ab Bit Position **position** mit dem Bit Muster **bitPattern**, ohne die übrigen Bits zu verändern. Der modifizierte Wert **value** wird zurückgegeben.

3.2 Bit Muster Initialisierung

Wie definiert man am einfachsten einen spezifischen Bit Wert?

Geben Sie den Initialwert auf verschiedene Arten an für einen Wert bei dem Bit 3 = 1 und Bit 13 = 1 ist und die übrigen Bits = 0 sind.

```
uint16_t value = 8200; // 2^13 + 2^3 = 2^3 x 2^10 + 2^3 = 8 x 1024 + 8 = 8192 + 8 = 8200
```

```
uint16_t value = 0x2008; // 0010'0000'0000'1000
```

```
uint16_t value = 020010; // 0'010'000'000'001'000
```

```
uint16_t value = (1u << 3) | (1u << 13);
```

Welche Art ist für eine solche Problemstellung die am wenigsten fehleranfällige?

Für meinen Geschmack ist die Variante mit Shift und Or hier am Einfachsten, gefolgt von der Hexadezimalen Darstellung (aber Achtung: Die Bits genau abzählen!). Octal ist ähnlich schwierig zu Hex, Dezimal ist eindeutig am Schwierigsten.

3.3 Abfragen von Bits

Ergänzen Sie folgende Code Sequenzen.

```
// returns 1 if bit at position is 1, otherwise returns 0
int IsBitSet(uint32_t value, size_t position)
{

  uint32_t mask = 1u << position;


  return (value & mask) == mask;


}
```

```
// returns 1 if bit at position is 0, otherwise returns 0
int IsBitClear(uint32_t value, size_t position)
{

  uint32_t mask = 1u << position;


  return (value & mask) == 0;


}
```

```
// returns 1 if the signed number is odd, else returns 0
// note: works for positive and negative values
int IsOddNumber(int32_t value)
{
  // to implement: use bit operations only (i.e. no arithmetic operations allowed)

  // odd positive as well as odd negative number have the least significant bit set
  // e.g. 8 bit values: 1 = binary: 00000001, -1 = binary: 11111111

  return (value & 1) == 1;

  // BTW: working here on a signed variable does not matter since we do not use the >> operator
}
```

3.4 Modifizieren von Bit-Gruppen

Ergänzen Sie folgende Code Sequenz.

```
uint16_t value = ... // any arbitrary number given
// set bits 2..7 of value to the bit pattern 101010 without modifying any other bits

uint16_t clr = (1u << 6) - 1; // produce binary mask 111111
clr <<= 2;                    // shift mask to bit 2
clr = ~clr;                   // invert so it can be used to clear the 6 bits at bit 2

value &= clr;                 // clear the 6 bits at bit 2
value |= 0x2A << 2;           // set the binary value 101010 = 0x2A at bit 2

// invert bits 10..15 of value without modifying any other bits

uint16_t mask = (1u<<10)|(1u<<11)|(1u<<12)|(1u<<13)|(1u<<14)|(1u<<15); // one possible solution
value ^= mask; // toggle the 6 most significant bits




}
```

4 Theorie

4.1 Speicher Grösse eines Wertes – Anzahl Bytes

Zahlenwerte wie z.B. die Zahl **`123`**, werden in einem oder mehreren Bytes im Speicher abgelegt. Die Anzahl der benötigten und verwendeten Bytes hängt vom Typ ab. So speichert eine Variable vom Typ **`char`** eine Zahl in einem Byte. Ein Zahl vom Typ **`int`** wird in mehreren Bytes gespeichert (z.B. vier Bytes auf einem 32 Bit System).

Da die Integer Typen der Sprache C so ausgelegt sind dass die Grösse der Variablen auf das Ziel System optimiert ist, ist es unter Umständen angebracht, für Bit Manipulationen Typen zu verwenden bei denen es unabhängig vom Ziel System klar ist, welche Bit-Grösse sie haben. Dies ist insbesondere der Fall für hardware-nahe Programmierung.

Beispiele von Typen mit genau definierter Grösse sind im **`#include <stdint.h>`** zu finden, z.B. **`uint8_t`** (unsigned 8 Bit), **`int8_t`** (signed 8 Bit), etc.

4.2 Kodierung – Binär System

Wie genau werden Zahlenwerte wie z.B. 123 im Speicher abgelegt? Es gäbe verschiedene Möglichkeiten. Die üblich vorgegebene Art ist es, die Zahlen zur Basis 2 zu übersetzen.

Z.B. dezimal: 123 = 0x2^7 + **1**x2^6 + **1**x2^5 + **1**x2^4 + **1**x2^3 + 0x2^2 + **1**x2^1 + **1**x2^0 = binär: 0**1111011**

Somit werden die Werte in einem Positions-System zur Basis 2 gespeichert, analog zu den dezimal Zahlen welche ein Positions-System zur Basis 10 darstellen.

Zu bemerken ist, dass die Bit Nummerierung rechts bei **Null** beginnt (was identisch ist mit der 2-er Potenz der Bit Position).

Z.B. entspricht Bit 0 dem ersten Bit von rechts und steht für 2^0 = 1, Bit 5 ist das sechste Bit von rechts und steht für 2^5 = 32. Etc.

Bit-Nummer	…	7	6	5	4	3	2	1	0
2-er System	…	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Dezimalwert	…	128	64	32	16	8	4	2	1

Diese Interpretation der Bits hat sich bewährt, ist aber keineswegs zwingend. Wenn Sie einen Wert im Speicher anschauen dann wissen Sie nicht was die Bytes bedeuten solange Sie nicht wissen wie sie zu interpretieren sind.

Z.B. binär: 01000001 = dezimal: 65 = ASCII: ‘A’ = BCD: 41, etc.

4.3 Negative Zahlen – Zweier-Komplement

Um im oben erläuterten Binär System auch negative Zahlen darstellen zu können hat sich das Zweier-Komplement System bewährt. Dabei wird das Bit an der äussersten linken Position als **minus** die angegebene 2-er Potenz verrechnet.

Für einen 8 Bit Zweier-Komplement Wert heisst das:

Bit-Nummer	7	6	5	4	3	2	1	0
2-er Komplement	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Dezimalwert	-128	64	32	16	8	4	2	1

Z.B.

- binär: 00000001 = dezimal: 1
- binär: 11111111 = dezimal: -128+64+32+16+8+4+2+1 = -1

4.4 Einer-Komplement versus Zweier-Komplement

Das Einer-Komplement ist die bit-weise Invertierung aller Bits.

Z.B. das Einer-Komplement von binär: 01000001 ist binär: 10111110.

Um eine positive Zahl im Binär System in die Bit Darstellung der entsprechenden negativen Zahl umzurechnen (und umgekehrt) gibt es einen einfachen Weg:

1) Einer-Komplement der Ausgangszahl berechnen
2) +1 dazu addieren

Z.B.

- von 1 zu -1: binär: 00000001 → (1er Komplement) → 11111110 → (+1) → 11111111
- von -1 zu 1: binär: 11111111 → (1er Komplement) → 00000000 → (+1) → 00000001

Quiz: von +0 zu -0?

4.5 Bit Operatoren auf signed negative Werte

Wenn man Bit Operatoren anwendet, dann haben diese für **unsigned Typen** auf allen Systemen dieselbe garantierte Funktionalität.

Angewendet auf **signed Typen wo der Wert negativ ist**, kann die Funktionalität auf verschiedenen Ziel Systemen **unterschiedliche Resultate** liefern!

Deshalb beschränken wir uns bei den Bit Operatoren auf **unsigned Typen oder nur positive Werte**.

4.6 Logische Bit Operatoren

Die logischen Bit Operatoren arbeiten auf individuellen Bits. Die Wahrheitstabelle der vier logischen Bit Operatoren können Sie in der einschlägigen Literatur nachschlagen (oder noch besser: selber aufschreiben).

Eine für Programmierzwecke praktischere Formulierung der binären Operationen ist:

AND	OR	XOR
X X & 1 0 = X 0	X X | 1 0 = 1 X	X X ^ 1 0 = ~X X
→ 1 = unverändert	→ 1 = setzen	→ 1 = invertieren
→ 0 = löschen	→ 0 = unverändert	→ 0 = unverändert

Z.B.
```
uint8_t value = 0xF0; // binary: 11110000
uint8_t mask  = 0x3C; // binary: 00111100
```

- AND: `value & mask` → binär: 00110000: löscht Bit 0,1,6,7, belässt Bit 2,3,4,5
- OR: `value | mask` → binär: 11111100: setzt Bit 2,3,4,5, belässt Bit 0,1,6,7
- XOR: `value ^ mask` → binär: 11001100: invertiert Bit 2,3,4,5, belässt Bit 0,1,6,7

4.7 Bit Shift Operatoren

Die Shift Operatoren arbeiten auf dem gesamten Wert und verschieben diesen um die angegebene Anzahl Bits nach link bzw. rechts. Die Bits welche über die Speicher-Grösse des Wer-

tes hinausgeschoben werden gehen verloren. Die auf der anderen Seite frei werdenden Bits werden mit 0 gefüllt.

Das Verschieben um N Stellen nach links entspricht im Binär System einer Multiplikation mit 2^N. Das Verschieben um N Stellen nach rechts entspricht einer Division durch 2^N.

Z.B. Left-Shift (am Beispiel von uint8_t Werten)

- 0x1 << 0 → binär: 00000001 << 0 = 00000001 = 1u x 2^0 = 1 x 1 = 1
- 0xF << 3 → binär: 00001111 << 3 = 01111000 = 15u x 2^3 = 15 x 8 = 120
- 0xF << 7 → binär: 00001111 << 7 = 10000000

  (= 15u x 2^7 = 15 x 128 = out-of-range = nicht als Multiplikation zu interpretieren)

Z.B. Right-Shift (am Beispiel von uint8_t Werten)

- 0xFF >> 7 → binär: 11111111 >> 7 = 00000001 = 255u / 2^7 = 255 / 128 = 1
- 0x15 >> 2 → binär: 00010101 >> 2 = 00000101 = 21u / 2^2 = 21 / 4 = 5
- 0xA5 >> 0 → binär: 10100101 >> 0 = 10100101 = 165u / 2^0 = 165 / 1 = 165

Zu beachten: **Das obige gilt für unsigned oder positive Werte**. Bei signed und negativen Werten ist der Right-Shift (>>) Operator sogenannt «implementation defined». D.h. ein Ziel System kann für signed Typen bei negativen Werten beim >>-Operator anstelle von 0 für die frei werdenden Bits jeweils eine 1 setzen. Das kann Sinn machen, denn dann bleibt das Vorzeichen erhalten und die Division um 2^N funktioniert dann auch – aber nur fast.

Das Problem ist die Rundung, bzw. das Abschneiden beim Dividieren. Denn nach herkömmlichen Rechenregeln ist für Integer Arithmetik 3/2 = 1 (Bruch-Teil wird abgeschnitten = Rundung gegen Null). Sinngemäss ist die Erwartung dass -3/2 auch gegen Null gerundet wird, was bei Anwendung des Divisionsoperators auch der Fall ist. Bei Anwendung der Right-Shift Operators wird aber gegen minus Unendlich gerundet, somit ergibt -3>>1 den Wert -2 und nicht -1!

Also: Vorsicht bei Right-Shift mit signed Typen und negativen Werten.

4.8 Werte setzen

Die Sprache C erlaubt es leider nicht, Werte als Binär-Muster anzugeben. Unterstützte Zahlenformate sind:

Art	Beginnt mit	Gefolgt von	Typ
Dezimal	1-9	0-9 (0..N mal)	signed int
Oktal	0	0-7 (0..N mal)	unsigned int
Hexadezimal	0x oder 0X	0-9, a-f, A-F (1..N mal)	unsigned int

Dezimal Zahlen werden als signed int interpretiert, Oktal und Hex Zahlen als unsigned int. Mit dem Suffix u kann eine Dezimal Zahl ebenfalls als unsigned int angegeben werden (das Umgekehrte gibt es aber nicht: eine unsigned Zahl als signed angeben…).

Z.B. auf einen 32 Bit Ziel System

- 123 → signed: 123 → binär: 00000000000000000000000001111011
- 123u → unsigned: 123 → binär: 00000000000000000000000001111011
- 0173 → unsigned: 0137 → binär: 00000000000000000000000001111011
- 0x7B → unsigned: 0x7B → binär: 00000000000000000000000001111011