OPERATORS

By Mustafa Onur Parlak

Category	Operator	Name/Description	Example	Result
Arithmetic	+	Addition	3+2	5
	-	Subtraction	3-2	1
	*	Multiplication	3*2	6
	/	Division	10/5	2
	%	Modulus	10%5	0
	++	Increment and then return value	X=3; ++X	4
		Return value and then increment	X=3; X++	3
	--	Decrement and then return value	X=3; --X	2
		Return value and then decrement	X=3; X--	3
Logical	&&	Logical "and" evaluates to true when both operands are true	3>2 && 5>3	False
	||	Logical "or" evaluates to true when either operand is true	3>1 || 2>5	True
	!	Logical "not" evaluates to true if the operand is false	3!=2	True
Comparison	==	Equal	5==9	False
	!=	Not equal	6!=4	True
	<	Less than	3<2	False
	<=	Less than or equal	5<=2	False
	>	Greater than	4>3	True
	>=	Greater than or equal	4>=4	True
String	+	Concatenation(join two strings together)	"A"+"BC"	ABC

- **x++ ve ++x Arasındaki Fark?**
  - **x++** yazıldığında, önce **x'in değeri yazdırılır**. Daha sonra **değer 1 kez arttırılır.**
  - **x++** yazıldığında, **flash memory** byte'ları okunur. Diğer durumda ise verilerin ilk byte'ını göz ardı edersin.
  - **++x** yazıldığında, önce **x'in değeri 1 kez arttırılır.** Daha sonra **x değeri yazdırılır.**
  - **++x** yazıldığında, **eski verinin kopyası** oluşturacağından dolayı **ekstra zaman tüketilir.**

- **Byte Değişimleri**

```
// 16-bit değeri 8-bit'e çevirme

uint8_t lowByte_u8;
uint8_t highByte_u8;
uint16_t Word_u16;

// 1. Yöntem
Word_u16 = (highByte_u8*256) + lowByte_u8;

// 2. Yöntem
Word_u16 = (highByte_u8<<8) +  lowByte_u8;

// 3. Yöntem
Word_u16 = (highByte_u8<<8) |  lowByte_u8;

// 4. Yöntem
typedef union
{
uint8_t  Data_u8[2];
uint16_t Data_u16;
}DataGroup16_tu;

DataGroup16_tu DataGroup16_u;
DataGroup16_u.Data_u8[0] = lowByte_u8;
DataGroup16_u.Data_u8[1] = highByte_u8;
Word_u16 = DataGroup16_u.Data_u16;
```

Buradaki yöntemler aşağı indikçe **Gömülü C** programlama kullanımı için daha **elverişli** olmaktadır. **Dört işlem operatörleri** her ne kadar yazılımda kolaylık sağlamış olsa da, donanım tarafını göz önüne alınca ciddi anlamda **verimi düşürmektedir**.

Bu sebeple **3 ve 4. Yöntemin**, ideal gömülü yazılım içeriği oluşturduğu kanaatindeyim.

- **Bitwise / Binary Değişimleri**

Normal şartlarda register kodlaması yapmak, Decimal ve Binary kodlamadan çok daha verimli ve hızlı gerçekleşmektedir.

```
#define A 13
#define B 23

A = 0000 1101

B = 0001 0111

// Buna göre,

A&B = 0000 0100 //İkisinde olanlar

A|B = 0001 1111 // Bütün değerlikli bitler

A^B = 0001 1010 // Sadece onda olan
```

```
#define A 13 // Bitsel Karşılıkları: A = 0000 1101

// X << Y = X, kayacak değer; Y, kaydırılacak yer sayısı

// Sola Shift (<<) Operatörü

A << 1 = 0001 1010

// Sağa Shift (>>) Operatörü

A >> 1 = 1000 0110
-----------------------------------------------------------
/* & - lojik 1 biti tersleme
   | - lojik 0 biti tersleme
   ^ - lojik bitleri tersleme
   &, | ve ^ ifadesi X OLSUN
   A=247 olsun
*/

// 1. Yöntem
A = A X 0x11110111

// 2. Yöntem
A = A X 0xF7;

// 3. Yöntem
A X= ~(1<<3); // '~' işareti sadece &'de kullanılır
```

```
const a = 5;   // 00000000000000000000000000000101
const b = 2;   // 00000000000000000000000000000010

x(a << b);     // 00000000000000000000000000010100
```