CHAPTER 2

Arduino Cookbook, Second Edition

스케치 만들기

2.0 소개

아두이노 프로젝트에서는 아두이노 보드와 지원 하드웨어를 통합하는 작업이 큰 비중을 차지하기는 하지만, 프로젝트의 나머지 부분에서는 어떤 작업을 수행하려고 하는지를 보드에게 알려 주어야 한다. 아두이노 프로그래밍과 관련된 주요 요소를 소개하는 이 장에서는 프로그래머가 아닌 사용자를 위해 일반적인 언어 구조를 사용하는 방법에 대해 설명한 후 아두이노에서 사용하는 C나 C++에 익숙하지 않는 독자들을 위해 C 언어의 구조를 개괄적으로 소개한다.

아두이노를 사용하여 흥미로운 작업을 수행하는 예제를 제공하기 위해 이 책의 레시피에서는 보드의 실제 기능을 사용하고 있으며, 이러한 기능에 대해서는 이후에 하나하나 살펴볼 것이다. 이 장의 코드를 살펴볼 때 쉽게 이해되지 않는 부분이 있다면 해당 주제에 대해 설명하는 부분을 먼저 살펴보는 것도 좋은 방법이 될 것이다. 특히 시리얼 출력에 대한 설명은 4장, 디지털 및 아날로그 핀을 사용하는 방법에 대한 설명은 5장에서 자세히 다루고 있다. 그리고 레시피에서 중점을 두고 있는 기능이 어떻게 수행되는지 알기 위해 예제의 코드를 반드시 모두 이해해야 하는 것은 아니다. 다음은 앞으로 살펴볼 예제에서 자주 사용되는 함수 중 일부다.

`Serial.println(value);`

아두이노 IDE의 시리얼 모니터에 값을 인쇄한다. 따라서 컴퓨터에서 아두이노의 출력을 볼 수 있다(레시피 4.1 참조).

`pinMode(pin, mode);`

디지털 값을 읽거나(입력) 쓰도록(출력) 디지털 핀을 구성한다(5장. 소개 참조).

`digitalRead(pin);`

입력으로 설정된 핀의 디지털 값(`HIGH` 또는 `LOW`)을 읽는다(레시피 5.1 참조).

`digitalWrite(pin, value);`

출력으로 설정된 핀에 디지털 값(`HIGH` 또는 `LOW`)을 쓴다(레시피 5.1 참조).

2.1 아두이노 프로그램 구조화하기

과제

프로그래밍 입문자로서 아두이노 프로그램의 빌딩 블록을 이해하고 싶다.

해결책

아두이노에서는 일반적으로 프로그램을 **스케치**라고 부른다. 초기에 주로 아티스트와 디자이너가 사용했기 때문에 아이디어를 빠르고 쉽게 실현할 수 있는 방법이라는 의미로 **스케치**라는 용어가 사용되었다. 따라서 **스케치**(sketch)와 **프로그램**(program)이라는 용어는 동의어로 이해해도 무관하다. 스케치에는 보드에서 실행할 명령어, 즉 코드가 들어 있다. 애플리케이션에 맞게 보드를 설정하는 작업을 수행하는 경우처럼 한 번만 실행해야 하는 코드는 `setup` 함수에 배치해야 한다. 초기 설정 완료 후 지속적으로 실행되어야 하는 코드는 `loop` 함수에 배치해야 한다. 다음은 전형적인 스케치다.

```
const int ledPin = 13;      // LED를 디지털 핀 13에 연결한다.

// setup() 메소드는 스케치가 시작될 때 한 번만 실행된다.
void setup()
{
  pinMode(ledPin, OUTPUT); // 디지털 핀을 출력으로 초기화한다.
}
```

```
// loop() 메소드는 반복해서 실행된다.
void loop()
{
  digitalWrite(ledPin, HIGH);       // LED를 켠다.
  delay(1000);                      // 1초 동안 대기한다.
  digitalWrite(ledPin, LOW);        // LED를 끈다.
  delay(1000);                      // 1초 동안 대기한다.
}
```

아두이노 IDE에서 코드를 업로드한 후에는 보드를 켤 때마다 코드의 맨 위부터 순차적으로 실행된다. 먼저 setup 안에 있는 코드가 실행된 다음 loop 안에 있는 코드가 실행된다. loop의 끝(닫는 중괄호, })에 도달한 이후에는 loop의 시작 부분으로 다시 돌아간다.

토론

이 예제에서는 핀을 HIGH 및 LOW 출력으로 번갈아 가면서 설정하는 방법을 통해 LED를 계속해서 깜박이게 만든다. 아두이노 핀의 사용 방법에 대한 내용은 5장을 참조한다. 스케치가 시작되면 setup 함수에서 핀 모드가 설정되며, 이제부터 LED를 켤 수 있다. setup의 코드가 완료된 후에는 아두이노 보드의 전원이 끊길 때까지 loop의 코드가 반복해서 호출되면서 LED가 깜박이게 된다.

아두이노 스케치를 작성할 때 몰라도 되는 내용이기는 하지만, 숙련된 C/C++ 프로그래머라면 진입점 함수인 main() 함수가 없다는 점이 조금 낯설게 느껴질 것이다. 물론 main() 함수가 있기는 하다. 단지 아두이노 빌드 환경에 숨겨져서 보이지 않을 뿐이다. 빌드 프로세스가 진행되는 동안 스케치 코드와 다음과 같은 추가 구문이 포함된 중간 파일이 생성된다.

```
int main(void)
{
    init();
    setup();
    for (;;)
        loop();

    return 0;
}
```

main() 함수에서는 먼저 init() 함수를 호출하여 아두이노 하드웨어를 초기화한다. 그런 다음에는 스케치의 setup() 함수가 호출된다. 마지막으로 loop() 함수가 반복해서 호출된다. 이 경우에는 for 루프가 절대로 종료되지 않기 때문에 return 문이 한 번도 실행되지 않는다.

참고

레시피 1.4에서 스케치를 아두이노 보드에 업로드하는 방법에 대해 설명한다.

17장과 http://www.arduino.cc/en/Hacking/BuildProcess에서 빌드 프로세스에 대한 자세한 설명을 볼 수 있다.

2.2 간단한 기본 유형(변수) 사용하기

과제

아두이노에서는 다양한 유형의 변수를 사용해서 값을 효율적으로 표현할 수 있다. 이러한 아두이노의 다양한 데이터 유형을 선택하고 사용하는 방법을 알고 싶다.

해결책

아두이노 애플리케이션에서는 int(integer의 약어, 아두이노의 경우 16비트 값) 데이터 유형이 숫자 값에 주로 사용되기는 하지만, 표 2-1을 보고 애플리케이션에서 사용할 값 범위에 적합한 데이터 유형을 결정할 수도 있다.

표 2-1 아두이노 데이터 유형

숫자 유형	바이트	범위	용도
int	2	-32768 ~ 32767	양수 및 음수 값을 나타낸다.
unsigned int	2	0 ~ 65535	int와 유사하지만 양수 값만 나타낸다.
long	4	-2147483648 ~ 2147483647	매우 큰 범위의 양수 및 음수 값을 나타낸다.
unsigned long	4	4294967295	매우 큰 범위의 양수 값을 나타낸다.

표 2-1 (계속)

숫자 유형	바이트	범위	용도
float	4	3.4028235E+38 ~ - 3.4028235E+38	소수점이 있는 숫자를 나타내며, 실제 측정치의 근삿값을 구하는 데 사용된다.
double	4	float와 같음	아두이노에서는 double과 float가 이름만 다를 뿐 동일한 데이터 유형이다.
boolean	1	false(0) 또는 true(1)	true 및 false 값을 나타낸다.
char	1	-128 ~ 127	하나의 문자를 나타낸다. -128부터 127 사이의 부호 있는 값을 나타낼 수도 있다.
byte	1	0 ~ 255	char와 유사하지만 부호 없는 값에만 사용된다.

기타 유형	용도
String	일반적으로 텍스트를 포함할 때 사용되는 문자 배열을 나타낸다.
void	리턴되는 값이 없는 함수 선언에서만 사용된다.

토론

성능이나 메모리 효율성이 필요하지 않은 경우 값 범위(표 2-1의 첫 번째 행 참조)가 적당하고, 소수 값을 사용하지 않아도 된다면 int를 사용하여 선언된 변수를 숫자 값에 사용하는 것이 좋다. 대부분의 공식 아두이노 예제 코드에서는 숫자 변수를 int로 선언한다. 하지만 애플리케이션에 따라 구체적인 유형을 선택해야 하는 경우도 발생할 수 있다.

음수가 필요한 경우도 있고, 그렇지 않은 경우도 있을 수 있기 때문에 숫자 유형에는 signed와 unsigned라는 두 가지 변형이 있다. unsigned 값은 항상 양수다. 앞쪽에 unsigned 키워드가 없는 변수는 부호가 있는 변수이므로 음수 및 양수 값을 나타낼 수 있다. unsigned 값은 주로 signed 값의 범위가 원하는 변수 범위와 맞지 않을 때 사용된다. (부호 없는 변수의 최댓값이 부호 있는 변수의 최댓값보다 2배 더 크다.) 이외에도 음수가 절대 사용되지 않을 것으로 예상되는 부분에서 이 점을 명확하게 나타내기 위해 unsigned 유형을 사용하기도 한다.

boolean 유형에는 true와 false라는 두 가지 값을 사용할 수 있다. 이 유형의 값은 대개 스위치의 상태(눌려진 상태인지 아닌지)를 검사하는 것과 유사한 상황에서 사용된

다. 이해하기 쉬운 코드를 작성하기 위해 true와 false 대신 HIGH와 LOW를 사용할 수도 있다. 예를 들어, digitalWrite(pin, HIGH), digitalWrite(pin, true) 그리고 digitalWrite(pin,1)는 모두 LED를 켜는 구문이지만 첫 번째 구문이 그 의미를 가장 명확하게 전달해 준다. 물론 이 세 구문은 모두 스케치가 실행될 때 동일하게 처리되며, 웹에 게시된 코드에서도 세 가지 형식이 모두 사용되고 있다.

참고

아두이노 레퍼런스 사이트(http://www.arduino.cc/en/Reference/HomePage)에서 데이터 유형에 대한 자세한 설명을 볼 수 있다.

2.3 부동 소수점 숫자 사용하기

과제

부동 소수점 숫자는 소수점이 포함된 값을 표현할 때 사용되며, 분수를 표현할 때도 이 방법이 사용된다. 이제 스케치에서 이러한 값을 계산하고 비교하고 싶다.

해결책

다음 코드에서는 부동 소수점 변수를 선언하는 방법을 보여 준 후 부동 소수점 값을 비교할 때 발생할 수 있는 문제점과 함께 그 해결 방법을 보여 준다.

```
/*
 * 부동 소수점 예제
 * 이 스케치에서는 부동 소수점 값을 1.1로 초기화했다.
 * 그런 다음 0이 될 때까지 반복해서 값을 0.1씩 줄여나간다.
 */

float value = 1.1;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
```

```
value = value - 0.1; // 루프를 한 번 순환할 때마다 값이 0.1씩 줄어든다.
if(value == 0)
  Serial.println("The value is exactly zero");
else if(almostEqual(value, 0))
{
  Serial.print("The value ");
  Serial.print(value, 7);// 소수점 이하 7자리까지 인쇄한다.
  Serial.println(" is almost equal to zero");
}
else
  Serial.println(value);

delay(100);
}

// a와 b 사이의 차이가 작으면 true를 리턴한다.
// DELTA 값을 설정하여 서로 비슷한 값을 같은 값으로 간주할 때 적용되는 허용 오차를 지정한다.
boolean almostEqual(float a, float b)
{
  const float DELTA = .00001; // 거의 같은 값으로 간주할 때 적용되는 최대 허용 오차
  if (a == 0) return fabs(b) <= DELTA;
  if (b == 0) return fabs(a) <= DELTA;
  return fabs((a - b) / max(fabs(a), fabs(b))) <= DELTA;
}
```

토론

부동 소수점 계산은 정확하지 않기 때문에 리턴된 값에 매우 경미한 근사치 오류가 있을 수 있다. 이는 부동 소수점 값의 범위가 너무 넓기 때문에 발생하는 오류이며, 결국 내부적으로 표현되는 값 또한 근사치일 수밖에 없다. 이 점을 감안한다면 정확히 일치하는지 여부를 테스트하기보다는 값이 허용 가능한 범위 내에 있는지 여부를 테스트해야 한다.

이 스케치의 시리얼 모니터 출력은 다음과 같다.

```
1.00
0.90
0.80
0.70
0.60
0.50
0.40
```

```
0.30
0.20
0.10
The value -0.0000001 is almost equal to zero
-0.10
-0.20
```

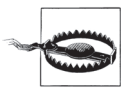
이후에도 계속해서 음수가 출력된다.

아마도 0.1이 출력된 이후에 "The value is exactly zero"가 출력되고, 그런 다음 -0.1이 출력될 것이라고 예상했을 것이다. 하지만 value는 0에 매우 가까워지기는 하지만 결코 0과 정확히 같아지지는 않기 때문에 if (value == 0)이라는 조건문이 충족되지 못한다. 이는 메모리 효율성을 높이기 위해 정확한 값 대신 근사치를 저장하는 방법을 사용하기 때문에 발생하는 현상으로, 이렇게 하면 부동 소수점 숫자로 표현할 수 있는 값의 범위를 확대할 수 있다.

이 문제를 해결하는 방법은 이 레시피의 해결책에서 본 것처럼 변수가 원하는 값에 가까운지 여부를 검사하는 것이다.

almostEqual 함수는 변수 값과 원하는 목표 값의 차이가 0.00001 이내인지 테스트한 후 두 값의 차이가 0.0001 이내이면 true를 리턴한다. 허용되는 범위는 상수 DELTA를 사용해서 설정했다. 이 상수는 필요에 따라 더 크거나 작은 값으로 변경할 수 있다. fabs(floating-point absolute value의 약어) 함수는 부동 소수점 변수의 절댓값을 리턴하며, 주어진 두 매개변수의 차이를 테스트하는 데 사용된다.

부동 소수점 연산에서 근사치를 사용하는 이유는 아주 넓은 범위에 속한 모든 값을 32비트만을 사용하여 처리하기 위해서다. 8비트는 10의 멱수(지수)에 사용되고, 나머지 24비트가 부호와 값에 사용되는데, 24비트로는 7자리 숫자까지만 표현할 수 있다.



아두이노에서는 float와 double이 완전히 동일하지만, 다른 플랫폼에서는 일반적으로 double의 정밀도가 높은 편이다. 다른 플랫폼에서 float와 double이 포함된 코드를 가져올 경우에는 숫자의 정밀도가 자신의 애플리케이션에 적합한지 확인해야 한다.

참고

float에 대한 아두이노 레퍼런스 페이지: http://www.arduino.cc/en/Reference/Float.

2.4 여러 개의 값으로 구성된 배열 작업

과제

여러 개의 값으로 구성된 배열을 만들어서 사용하고 싶다. 배열은 간단한 목록일 수도 있고 2개 이상의 차원으로 이루어질 수도 있다. 이제 배열의 크기를 결정하는 방법과 배열의 차원에 액세스하는 방법에 대해 알고 싶다.

해결책

이 스케치에서는 배열 두 개를 만든다. 하나는 스위치에 연결된 핀과 관련된 정수 배열이고, 다른 하나는 LED에 연결된 핀으로 구성된 배열이다(그림 2-1).

```
/*
 array 스케치
 스위치 배열이 LED 배열을 제어한다.
 스위치 사용에 대한 자세한 설명은 5장을 참조한다.
 LED에 대한 자세한 설명은 7장을 참조한다.
 */

int inputPins[] = {2, 3, 4, 5}; // 스위치 입력에 사용할 핀 배열을 만든다.

int ledPins[] = {10, 11, 12, 13}; // LED 출력에 사용할 핀 배열을 만든다.

void setup()
{
  for(int index = 0; index < 4; index++)
  {
    pinMode(ledPins[index], OUTPUT);         // LED를 출력으로 선언한다.
    pinMode(inputPins[index], INPUT);        // 누름 단추를 입력으로 선언한다.
    digitalWrite(inputPins[index], HIGH);    // 풀업 저항을 설정한다.
    // (레시피 5.2 참조)
  }
}
  void loop(){
    for(int index = 0; index < 4; index++)
    {
      int val = digitalRead(inputPins[index]);       // 입력 값을 읽는다.
      if (val == LOW)                          // 스위치가 눌려 있는지 검사한다.
    {
      digitalWrite(ledPins[index], HIGH);    // 스위치가 눌려 있으면 LED를 켠다.
    }
```

```
      else
      {
        digitalWrite(ledPins[index], LOW);    // LED를 끈다.
      }
    }
  }
```

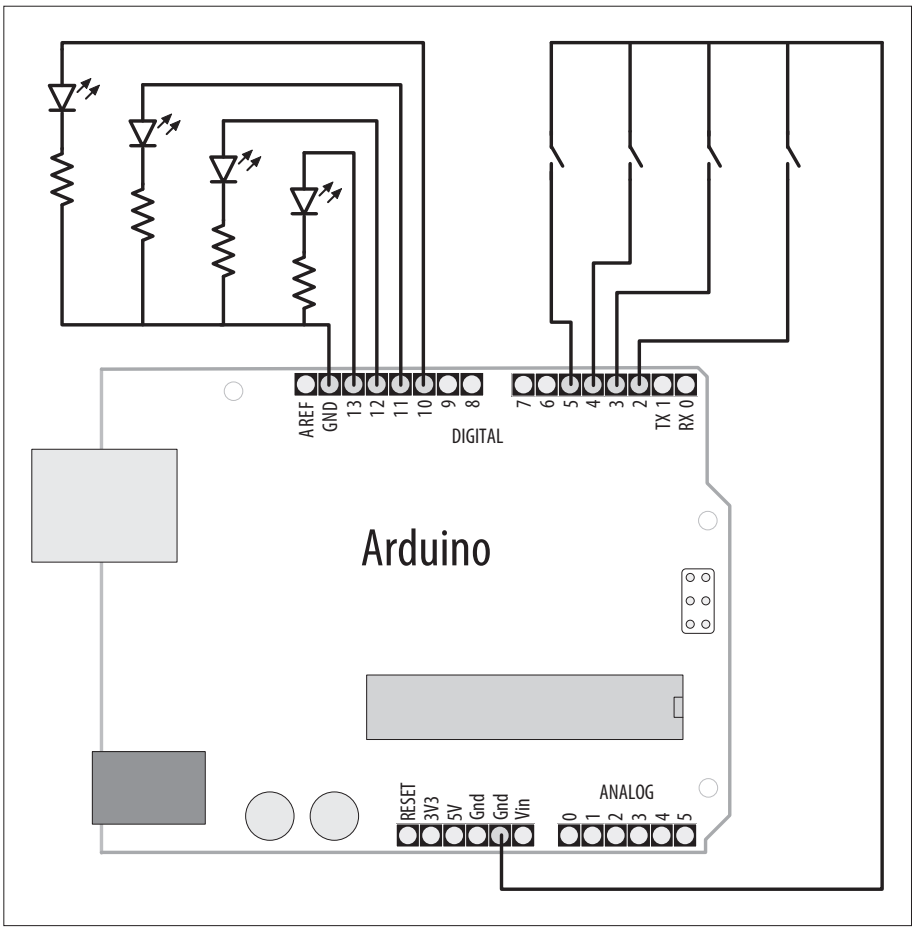


그림 2-1 LED 및 스위치 연결

토론

배열은 같은 유형의 변수를 모아 놓은 콜렉션이며, 콜렉션에 있는 각 변수를 요소(element)라고 한다. 그리고 요소의 개수를 배열의 크기(size)라고 한다.

이 레시피의 해결책에서는 핀 콜렉션을 저장하는 예제를 통해 아두이노 코드에서 일반적으로 배열을 사용하는 방법을 보여 준다. 그리고 이 예제에서 각 핀은 스위치와 LED에 연결된다(자세한 설명은 5장 참조). 이 예제에서는 배열을 선언하고 배열 요소에 액세스하는 방법이 중요하다.

아래 코드에서는 4개의 요소가 있는 정수 배열을 선언한(만든) 다음 각 요소를 초기화한다. 첫 번째 요소는 2, 두 번째 요소는 3으로 설정되며, 이와 같은 방식으로 나머지 두 요소도 설정된다.

```
int inputPins[] = {2, 3, 4, 5};
```

배열을 선언할 때 값을 초기화하지 않은 경우에는(예를 들어, 스케치가 실행 중일 때만 값을 사용할 수 있는 경우) 각 요소를 개별적으로 변경해야 한다. 배열을 선언하는 방법은 다음과 같다.

```
int inputPins[4];
```

이렇게 하면 각 요소의 초깃값이 0으로 설정된 4개의 요소로 구성된 하나의 배열이 선언된다. 대괄호([]) 안에 있는 숫자가 배열의 크기에 해당하며, 이 값에 따라 요소의 개수가 결정된다. 이 배열은 크기가 4이므로 최대 4개의 정수 값을 저장할 수 있다. 첫 번째 예제처럼 배열 선언에 이니셜라이저가 포함되어 있으면 크기를 생략할 수 있는데, 이는 컴파일러가 이니셜라이저의 개수를 계산하여 배열의 크기를 설정할 수 있기 때문이다.

배열의 첫 번째 요소는 `element[0]`이다.

```
int firstElement = inputPin[0]; // 첫 번째 요소

inputPins[0] = 2; // 이 요소의 값을 2로 설정한다.
```

마지막 요소는 전체 크기보다 하나 더 작다. 따라서 이전 예제에서 선언한 배열의 경우 크기가 4이므로 `element[3]`이 마지막 요소가 된다.

```
Int lastElement = inputPin[3]; // 마지막 요소
```

치수가 4인 배열의 마지막 요소에 액세스할 때 `array[3]`을 사용하는 것이 어색하게 느껴질 수도 있겠지만, 첫 번째 요소가 `array[0]`이라는 점을 생각하면 쉽게 이해할

수 있을 것이며, 이러한 배열의 네 요소는 다음과 같다.

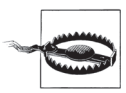
```
inputPins[0], inputPins[1], inputPins[2], inputPins[3]
```

이전 스케치에서는 다음과 같이 for 루프를 사용하여 4개의 요소에 액세스했다.

```
for(int index = 0; index < 4; index++)
{
    // 핀 배열의 각 요소에 액세스하여 핀 번호를 가져온다.
    pinMode(ledPins[index], OUTPUT);    // LED를 출력으로 선언한다.
    pinMode(inputPins[index], INPUT);   // 누름 단추를 입력으로 선언한다.
}
```

이 루프는 변수 index의 값이 0일 때부터 시작해서 3이 될 때까지 실행된다. 배열의 실제 크기를 벗어난 위치에 있는 요소에 실수로 액세스하는 경우가 종종 발생한다. 이러한 실수로 인해 다른 여러 가지 문제가 발생할 수 있으므로 이런 일이 발생하지 않도록 주의해야 한다. 이러한 실수를 방지하면서 루프를 올바르게 제어할 수 있는 방법으로 다음과 같이 상수를 사용하여 배열의 크기를 설정할 수 있다.

```
const int PIN_COUNT = 4;   // 요소의 개수를 상수로 정의한다.
int inputPins[PIN_COUNT] = {2, 3, 4, 5};

  for(int index = 0; index < PIN_COUNT; index++)
    pinMode(inputPins[index], INPUT);
```



실수로 배열의 크기를 벗어난 위치에 요소를 저장하거나 그러한 위치에 있는 요소를 읽더라도 컴파일러에서는 오류를 보고하지 않는다. 그러므로 반드시 설정한 경경 내에 있는 요소에만 액세스해야 한다. 상수를 사용하여 배열의 크기를 설정한 후 배열의 요소를 참조하게 되면 배열의 크기를 벗어나지 않는 범위 내에서 배열을 안전하게 사용할 수 있다.

배열은 일련의 텍스트 문자들을 저장할 때도 사용된다. 아두이노 코드에서는 이를 문자열(character strings)이라고 부른다. 문자열은 하나 이상의 문자로 구성되어 있으며, 맨 뒤에 문자열의 끝임을 나타내는 널 문자(값 0)가 붙는다.

문자열의 끝에 있는 널 문자는 문자 0과는 다른 문자다. 널 문자의 ASCII 값은 0이고, 문자 0의 ASCII 값은 48이다.

문자열을 사용하는 메소드에 대한 자세한 설명은 레시피 2.5와 2.6을 참조한다.

참고

레시피 5.2와 레시피 7.1을 참조한다.

2.5 아두이노의 문자열 기능 사용하기

과제

텍스트 작업을 수행하고 싶다. 텍스트를 복사하고, 추가도 하고, 문자 수도 알아내고 싶다.

해결책

앞의 레시피에서 문자 배열을 사용해서 텍스트를 저장하는 방법을 살펴보았는데, 이러한 문자 배열을 문자열이라고 부른다. 아두이노에는 텍스트를 저장하고 조작하는 등의 다양한 기능을 제공하는 `String` 기능이 있다.

아두이노에서 대문자 **S**로 시작되는 **String**이라는 단어는 아두이노 String 라이브러리에 포함되어 있는 아두이노 텍스트 기능을 의미한다. 소문자 **s**로 시작하는 **string**은 아두이노 `String` 기능이 아닌 일반적인 문자 그룹을 의미한다.

이 레시피에서는 아두이노의 `String` 기능을 사용하는 방법에 대해 설명한다.

`String` 기능은 버전 1.0보다 빠른 아두이노 0019 alpha 버전에서 도입되었다. 이전 버전을 사용하는 경우에는 Text-String 라이브러리를 사용할 수 있으며, 이 레시피의 끝에서 관련 링크를 볼 수 있다.

다음 스케치를 보드에 로드한 후 시리얼 모니터를 열고 결과를 확인한다.

```
/*
  Basic_Strings 스케치
```

```
 */

String text1 = "This string";
String text2 = " has more text";
String text3; // 스케치 내에서 지정된다.
void setup()
{
  Serial.begin(9600);

  Serial.print( text1);
  Serial.print(" is ");
  Serial.print(text1.length());
  Serial.println(" characters long.");

  Serial.print("text2 is ");
  Serial.print(text2.length());
  Serial.println(" characters long.");

  text1.concat(text2);
  Serial.println("text1 now contains: ");
  Serial.println(text1);
}

void loop()
{
}
```

토론

이 스케치에서는 text1, text2 및 text3이라는 String 유형의 변수 세 개를 생성한다. String 유형의 변수에는 텍스트를 조작할 수 있는 내장 기능이 있다. text1.length()는 text1 문자열의 길이(문자 수)를 리턴한다.

text1.concat(text2)는 문자열의 내용을 결합하는데, 이 경우에는 text2의 내용을 text1의 끝에 추가한다(concat은 concatenate의 약어).

시리얼 모니터에 다음과 같은 결과가 표시된다.

```
This string is 11 characters long.
text2 is 14 characters long.
text1 now contains:
This string has more text
```

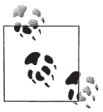
문자열을 결합하는 또 다른 방법으로 문자열 추가 연산자를 사용하는 방법도 있다. 아래 두 행을 setup 코드의 끝에 추가해 보자.

```
text3 = text1 + " and more";
Serial.println(text3);
```

새 코드를 실행하면 시리얼 모니터의 맨 밑에 아래 행이 추가된 결과가 표시된다.

```
This string has more text and more
```

indexOf 및 lastIndexOf 함수를 사용하여 문자열에 있는 특정 문자를 찾을 수 있다.



String 클래스는 아두이노에 최근에 추가된 클래스이기 때문에 기존 코드에서는 String 유형이 아닌 다른 유형의 문자 배열을 사용했었다. 아두이노 String 기능 없이 문자 배열을 사용하는 방법에 대한 자세한 설명은 레시피 2.6을 참조한다.

아래와 같은 행이 있다고 가정해 보자.

```
char oldString[] = "this is a character array";
```

이 코드에서는 C 스타일의 문자 배열을 사용하고 있다(레시피 2.6 참조). 그리고 다음과 같이 문자열을 선언했다고 가정하자.

```
String newString = "this is a string object";
```

이 코드에서는 아두이노 String을 사용하고 있다. C 스타일의 문자 배열을 아두이노 String으로 변환하는 방법은 매우 간단하다. 아래와 같이 배열의 내용을 String 오브젝트에 할당하기만 하면 된다.

```
char oldString[] = "I want this character array in a String object";
String newString = oldString;
```

표 2-2의 함수를 사용하려면 다음 예제와 같이 원하는 함수를 기존 String 오브젝트에 대해 호출해야 한다.

```
int len = myString.length();
```

표 2-2 아두이노 String 함수 개요

charAt(n)	String 오브젝트의 n번째 문자를 리턴한다.
compareTo(S2)	String 오브젝트를 지정된 String S2 오브젝트와 비교한다.
concat(S2)	String 오브젝트와 S2 오브젝트가 결합된 새 String 오브젝트를 리턴한다.
endsWith(S2)	String 오브젝트가 S2 오브젝트의 문자들로 끝나면 true를 리턴한다.
equals(S2)	String 오브젝트가 S2 오브젝트와 정확히 일치하면(대소문자 구분) true를 리턴한다.
equalsIgnoreCase(S2)	equals와 같지만 대소문자를 구분하지 않는다.
getBytes(buffer, len)	len에 해당하는 개수의 문자를 제공된 바이트 버퍼에 복사한다.
indexOf(S)	제공된 String 오브젝트(또는 문자)의 인덱스를 리턴하며, 없으면 -1을 리턴한다.
lastIndexOf(S)	indexOf와 같지만 String 오브젝트의 끝에서 시작된다는 점이 다르다.
length()	String 오브젝트의 문자 수를 리턴한다.
replace(A, B)	String 오브젝트(또는 문자) A의 모든 인스턴스를 String 오브젝트(또는 문자) B로 바꾼다.
setCharAt(index, c)	문자 c를 String 오브젝트의 지정된 위치(index)에 저장한다.
startsWith(S2)	String 오브젝트가 S2 오브젝트의 문자들로 시작되면 true를 리턴한다.
substring(index)	String 오브젝트의 지정된 위치(index)부터 마지막까지의 문자로 구성된 String 오브젝트를 리턴한다.
substring(index, to)	위 함수와 같지만 'to' 앞의 문자까지만 하위 문자열에 포함된다는 점이 다르다.
toCharArray(buffer, len)	String 오브젝트의 문자 중에서 최대 len개의 문자를 제공된 버퍼에 복사한다.
toInt()	String 오브젝트에 있는 숫자의 정수 값을 리턴한다.
toLowerCase()	모든 문자를 소문자로 변환한 String 오브젝트를 리턴한다.
toUpperCase()	모든 문자를 대문자로 변환한 String 오브젝트를 리턴한다.
trim()	선행 및 후행 공백을 모두 제거한 String 오브젝트를 리턴한다.

이 함수들의 사용법과 변형에 대한 자세한 설명은 아두이노 레퍼런스 페이지를 참조한다.

아두이노 String과 C 문자 배열 비교

아두이노의 내장 `String` 데이터 유형이 C 문자 배열보다 사용하기 쉽기는 하지만, String 라이브러리의 복잡한 코드가 실행되어야 하기 때문에 아두이노에 미치는 부하가 높을 뿐만 아니라 본질적으로 문제에 취약하다는 단점이 있다.

`String` 데이터 유형은 **동적 메모리 할당**(dynamic memory allocation)을 이용하기 때문에 유연성이 매우 높다. 즉, `String`을 만들거나 수정하면 C 라이브러리의 새 영역이 할당되며, `String`을 사용한 후에는 메모리를 해제해 주어야 한다. 이 유형은 대체로 정상적으로 작동하지만 메모리 누수를 발생시킬 수 있는 여러 가지 문제점을 가지고 있다. `String` 라이브러리의 버그로 인해 메모리의 일부 또는 전체가 제대로 리턴되지 않는 상황이 발생할 수 있다. 이런 상황이 발생하면 아두이노에서 사용할 수 있는 메모리 용량이 조금씩 줄어들게 된다(아두이노를 재부팅할 때까지). 메모리 누수가 발생하지 않았다고 하더라도 메모리 용량 부족으로 인해 `String` 요청이 실패했는지 여부를 검사하는 코드를 작성하기가 쉽지 않다. (Processing의 String 함수와 비슷하기는 하지만 아두이노에는 런타임 오류 예외 처리 기능이 없다.) 동적 메모리가 완전히 소진되었는지 여부를 추적한다는 것은 매우 어려운 일이다. 왜냐하면 메모리가 부족하여 예기치 않은 동작이 발생하기 전까지 스케치가 여러 날 또는 수 주 동안 아무 문제 없이 실행될 수 있기 때문이다.

C 문자 배열을 사용하면 메모리 사용량을 쉽게 제어할 수 있다. 이는 컴파일 타임에 정해진 용량의 메모리가 할당되기 때문이며(정적 메모리 할당), 따라서 메모리 누수가 발생하지 않는다. 이 경우 아두이노 스케치는 항상 동일한 용량의 메모리를 사용하게 된다. 그리고 사용할 수 있는 것보다 많은 용량의 메모리가 사용되는 상황이 발생하더라도 할당된 메모리 용량의 사용량을 알려 주는 도구가 있으므로 그 원인을 쉽게 찾아낼 수 있다(레시피 17.1의 `avr-objdump` 참조).

하지만 C 문자 배열을 사용할 경우에는 또 다른 문제가 있다. 즉, C에서는 배열의 범위를 벗어난 메모리를 수정하지 못하도록 차단하는 기능이 없다는 것이다. 따라서 `myString[3]`이 배열의 끝일 때 `myString[4]`라는 배열을 할당하고, `myString[4] = 'A'`를 지정하더라도 이 작업을 막을 수 있는 방법이 없다. 하지만 `myString[4]`가 참조하는 메모리에 어떤 데이터가 있는지 알 수 없기 때문에 이 메모리 위치에 `'A'`를 지정했을 때 문제가 발생할 수도 있다. 대부분의 경우 스케치에서 예기치 않은 결과가 발생한다.

아두이노의 내장 String 라이브러리는 동적 메모리를 사용하므로 사용 가능한 메모리의 용량이 줄어들 수 있는 위험이 있다. 그리고 C 문자 배열을 사용하려면 사용하는 배열의 범위를 벗어나지 않도록 주의를 기울여야 한다. 따라서 리치 텍스트 처리 기능이 필요하다면 아두이노의 내장 String 라이브러리를 사용하는 것이 효율적인데, 이 경우에는 `String`을 반복해서 만들거나 수정하지 않는 것이 좋다. 자주 반복되는 문자열을 루프에서 만들고 수정해야 하는 경우에는 용량이 큰 C 문자 배열을 할당하는 것이 좋으며, 배열의 범위를 벗어나지 않도록 주의해서 코드를 작성해야 한다.

사용할 수 있는 대부분의 RAM 또는 플래시 메모리를 사용해야 하는 대용량 스케치에서는 아두이노 `String`보다 C 문자 배열을 사용하는 것이 효과적이다. 아두이노 `String ToInt` 예제 코드에서는 C 문자 배열과 `atoi`를 사용하여 정수로 변환하는 코드에 비해 거의 2KB의 플래시 메모리가 더 많이 사용된다. 또한 아두이노 `String` 버전에는 실제 문자열 외에도 할당 정보를 저장하기 위해 약간의 추가 RAM이 필요하다.

String 라이브러리나 동적으로 할당된 메모리를 사용하는 다른 라이브러리에서 메모리 누수가 발생하고 있다는 의심이 들 경우에는 임의 시점의 여유 메모리 공간을 확인해 보는 것이 좋다(레시피 17.2 참조). 스케치를 시작할 때 RAM의 용량을 확인한 후 시간이 지나면서 메모리 용량이 줄어드는지 모니터링한다. String 라이브러리와 관련해서 문제가 있는 경우에는 알려진 버그 목록(http://code.google.com/p/arduino/issues/list)에서 "String"을 검색해 보기 바란다.

참고

아두이노 배포판에 `String` 예제 스케치가 포함되어 있다(파일 → 예제 → Strings).

String 레퍼런스 페이지(http://arduino.cc/en/Reference/StringObject)를 참조한다.

새 String 라이브러리에 대한 자습서는 http://arduino.cc/en/Tutorial/HomePage에서 볼 수 있으며, 기존 String 라이브러리에 대한 자습서는 http://www.arduino.cc/en/Tutorial/TextString에서 볼 수 있다(0019 alpha 이전 버전의 아두이노를 사용하는 경우).

2.6 C 문자열 사용하기

과제

원시 문자열을 사용하는 방법에 대해 알고 싶다. 즉, 문자열을 생성하는 방법, 그 길이를 알아내는 방법, 그리고 문자열을 비교, 복사 또는 추가하는 방법을 알고 싶다. 코어 C 언어에서는 아두이노 스타일의 `String` 기능이 지원되지 않기 때문에 원시 문자 배열을 처리할 수 있도록 작성된 다른 플랫폼의 코드를 이해할 수 있다면 많은 도움이 될 것이다.

해결책

문자로 구성된 배열을 종종 **문자열**(character strings)이라고도 한다. 레시피 2.4에서 아두이노 배열에 대해 알아보았다. 이 레시피에서는 문자열 작업에 필요한 함수에 대해 설명한다.

이제 다음과 같이 문자열을 선언한다.

```
char stringA[8]; // 최대 7개의 문자를 포함하고 널로 끝나는 문자열을 선언한다.
char stringB[8] = "Arduino";  // 위와 같은 문자열을 선언한 후 문자열을 "Arduino"로
                              // 초기화한다.
char stringC[16] = "Arduino"; // 위와 같지만 문자열에 여유 공간이 있다.
char stringD[ ] = "Arduino";  // 컴파일러가 문자열을 초기화하고 크기를 계산한다.
```

다음과 같이 `strlen`(string length의 약어)을 사용하면 종료 널 문자 앞까지의 문자 수를 알 수 있다.

```
int length = strlen(string); // 문자열의 문자 수를 리턴한다.
```

앞 코드에 있는 문자열 중에서 `stringA`의 길이는 0이고, 나머지 문자열의 길이는 7이다. 문자열의 끝을 나타내는 널 문자는 `strlen`의 계산에 포함되지 않는다.

다음과 같이 `strcpy`(string copy의 약어)를 사용하여 한 문자열을 다른 문자열로 복사할 수 있다.

```
strcpy(destination, source); // source 문자열을 destination 문자열에 복사한다.
```

strncpy를 사용하면 복사할 문자 수를 제한할 수 있으며, 결과적으로 대상 문자열의 허용 용량이 초과되는 상황을 방지할 수 있다. 레시피 2.7에서 이 함수의 용례를 볼 수 있다.

```
// 최대 6개의 문자를 source에서 destination으로 복사한다.
strncpy(destination, source, 6);
```

다음과 같이 strcat(string concatenate의 약어)을 사용하여 한 문자열을 다른 문자열의 끝에 추가할 수 있다.

```
// source 문자열을 destination 문자열의 끝에 추가한다.
strcat(destination, source);
```

문자열을 복사하거나 연결할 때는 여유 공간이 충분한지 항상 확인해야 한다. 종료 널 문자를 위한 공간도 잊지 말아야 한다.

다음과 같이 strcmp(string compare의 약어)를 사용하여 두 문자열을 비교할 수 있다. 레시피 2.7에서 이 함수의 용례를 볼 수 있다.

```
if(strcmp(str, "Arduino") == 0)
  // str 변수가 "Arduino"와 같으면 작업을 수행한다.
```

토론

아두이노 환경에서는 문자열이라는 문자 배열을 사용하여 텍스트가 표시된다. 문자열은 여러 개의 문자로 구성되어 있으며, 맨 뒤에 널 문자(값 0)가 포함되어 있다. 널 문자는 표시되는 문자는 아니지만 소프트웨어에게 문자열의 끝을 알려주는 역할을 한다.

참고

http://www.cplusplus.com/reference/clibrary/cstring/과 http://www.cppreference.com/wiki/string/c/start를 비롯한 여러 온라인 C/C++ 레퍼런스 페이지를 참조한다.

2.7 쉼표로 구분된 텍스트를 그룹으로 분리하기

과제

쉼표(또는 다른 구분 기호)로 구분된 두 가지 이상의 데이터가 포함된 문자열이 있다. 문자열의 각 부분을 개별적으로 사용하기 위해 문자열을 분리하고 싶다.

해결책

이 스케치는 각각의 쉼표 사이에 있는 텍스트를 출력한다.

```
/*
 * SplitSplit 스케치
 * 쉼표로 구분된 문자열을 분리한다.
 */

String text = "Peter,Paul,Mary";  // 예제 문자열
String message = text; // 아직 분리되지 않은 텍스트를 담고 있다.
int    commaPosition;  // 문자열에 있는 다음 쉼표의 위치

void setup()
{
  Serial.begin(9600);

  Serial.println(message);  // 소스 문자열을 표시한다.
  do
  {

    commaPosition = message.indexOf(',');
    if(commaPosition != -1)
    {
      Serial.println( message.substring(0, commaPosition));
      message = message.substring(commaPosition+1, message.length());
    }
    else
    { // 마지막 쉼표 이후
      if(message.length() > 0)
        Serial.println(message);  // 마지막 쉼표 이후에 텍스트가 있으면 해당 텍스트를
                                  // 인쇄한다.
    }
  }
  while(commaPosition >=0);
}
```

```
void loop()
{
}
```

시리얼 모니터에 다음과 같은 결과가 표시된다.

```
Peter,Paul,Mary
Peter
Paul
Mary
```

토론

이 스케치에서는 String 함수를 사용하여 쉼표 사이에 있는 텍스트를 추출한다. 다음 코드를 살펴보자.

```
commaPosition = message.indexOf(',');
```

이 코드는 commaPosition 변수를 message 문자열에 있는 첫 번째 쉼표의 위치로 설정한다(쉼표가 없으면 -1로 설정). 쉼표가 있으면 substring 함수를 사용하여 문자열의 처음부터 쉼표 앞까지의 텍스트를 인쇄한다. 그런 다음 아래 행에서는 인쇄된 텍스트와 뒤따르는 쉼표가 message에서 제거된다.

```
message = message.substring(commaPosition+1, message.length());
```

이 행에서 substring 함수는 commaPosition+1(첫 번째 쉼표 바로 뒤의 위치)부터 시작해서 메시지 길이만큼의 문자열을 리턴한다. 결과적으로 첫 번째 쉼표 이후의 텍스트만 포함된 메시지가 만들어진다. 그리고 이 과정은 쉼표가 더 이상 없을 때까지 반복된다. (commaPosition이 -1이 된다.)

숙련된 프로그래머라면 표준 C 라이브러리의 하위 레벨 함수를 사용할 수도 있을 것이다. 다음 스케치에서는 아두이노 문자열을 사용하여 앞의 스케치와 유사한 작업을 수행한다.

```
/*
 * SplitSplit 스케치
 * 쉼표로 구분된 문자열을 분리한다.
 */

const int MAX_STRING_LEN = 20;  // 앞으로 처리할 문자열 중 가장 긴 문자열로 설정한다.
```

```
char stringList[] = "Peter,Paul,Mary";  // 예제 문자열

char stringBuffer[MAX_STRING_LEN+1];   // 계산 및 출력을 위한 정적 버퍼

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  char *str;
  char *p;
  strncpy(stringBuffer, stringList, MAX_STRING_LEN); // 소스 문자열을 복사한다.
  Serial.println(stringBuffer);                      // 소스 문자열을 표시한다.

  for(str = strtok_r(stringBuffer, ", ", &p);        // 쉼표를 사용하여 분리한다.
      str;         // str이 널이 아닐 때까지 반복한다.
      str = strtok_r(NULL, ", ", &p)                          // 후속 토큰을 가
져온다.
     )
  {
    Serial.println(str);
  }
    delay(5000);
}
```

이 스케치에서 가장 중요한 부분은 `strtok_r`(아두이노 컴파일러에서 제공하는 `strtok` 함수의 이름) 함수다. 여기에서는 `strtok_r`을 처음 호출할 때 토큰을 사용하여 구분할(개별 값으로 분리할) 문자열을 매개변수로 전달한다. 하지만 `strtok_r`은 새 토큰을 찾을 때마다 문자열에 있는 문자를 덮어쓰므로 이 예제에서와 같이 문자열의 사본을 전달하는 것이 좋다. 이후 호출에서는 첫 번째 매개변수로 `NULL`을 지정하여 다음 토큰으로 이동하도록 지시한다. 이 예제에서는 각 토큰이 시리얼 포트에 인쇄된다.

토큰이 숫자로만 구성된 경우에는 레시피 4.5를 참조하기 바란다. 이 레시피에서는 문자열에서 쉼표로 구분된 숫자 값을 추출하는 방법을 보여 준다.

참고

`strtok_r`, `strcmp` 등의 C 문자열 함수에 대한 자세한 설명은 http://www.nongnu.org/avr-libc/user-manual/group__avr__string.html을 참조한다.

레시피 2.5와 C/C++ 함수 strtok_r 및 strcmp에 대한 온라인 레퍼런스를 참조한다.

2.8 숫자를 문자열로 변환하기

과제

LCD나 다른 디스플레이에 숫자를 표시하기 위해 숫자를 문자열로 변환하고 싶다.

해결책

String 변수에서는 숫자가 문자열로 자동으로 변환된다. 리터럴 값을 사용하거나 변수의 내용을 사용할 수 있다. 예를 들어, 다음 코드는 정상적으로 작동한다.

```
String myNumber = 1234;
```

아래와 같은 경우에도 정상적으로 작동한다.

```
int value = 127;
String myReadout = "The reading was ";
myReadout.concat(value);
```

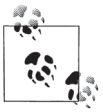
다음의 경우도 마찬가지다.

```
int value = 127;
String myReadout = "The reading was ";
myReadout += value;
```

토론

LCD나 시리얼 장치에 텍스트로 표시하기 위해 숫자를 변환할 경우에는 LCD 및 Serial 라이브러리에 있는 변환 기능을 사용하는 것이 가장 간단한 방법이다(레시피 4.2 참조). 하지만 이 기능이 지원되지 않는 장치를 사용하고 있거나(13장 참조) 스케치에서 숫자를 문자열로 조작하려면 어떻게 해야 할까?

아두이노 String 클래스는 숫자 값이 String 변수에 할당될 때 숫자 값을 문자열로 자동으로 변환한다. concat 함수나 문자열 연결 연산자를 사용하여 문자열 끝에 숫자 값을 결합(연결)할 수 있다.



+ 연산자는 숫자 유형뿐만 아니라 문자열에도 사용되지만 각기 다른 방식으로 작동한다.

다음 코드에서 number의 값은 최종적으로 13이 된다.

```
int number = 12;
number += 1;
```

다음과 같이 String을 사용하면 어떤 결과가 나올까?

```
String textNumber = "12";
textNumber += 1;
```

textNumber는 "121"이라는 텍스트 문자열이 된다.

String 클래스가 도입되기 전에는 대개 itoa 또는 ltoa 함수를 사용했다. itoa와 ltoa는 각기 "정수를 ASCII로(integer to ASCII)"와 "long 유형을 ASCII로(long to ASCII)"라는 의미를 담고 있다. 앞에서 설명한 String 버전이 사용하기 쉽기는 하지만, 레시피 2.6에서 설명한 것처럼 C 문자 배열을 사용하고 싶은 경우에는 아래와 같은 방법을 사용할 수 있다.

itoa나 ltoa에는 세 개의 매개변수가 있다. 즉, 변환할 값, 출력 문자열을 저장할 버퍼 그리고 기수(10진수의 경우 10, 16진수의 경우 16, 2진수의 경우 2)가 사용된다.

다음 스케치에서는 ltoa를 사용하여 숫자 값을 변환하는 방법을 보여 준다.

```
/*
 * NumberToString
 * 주어진 숫자를 사용하여 문자열을 만든다.
 */

void setup()
{
  Serial.begin(9600);
}

char buffer[12];  // 11개의 문자(- 부호 포함)와 종료 널을 위한 long 데이터 유형
void loop()
{
  long value = 12345;
  ltoa(value, buffer, 10);
```

```
    Serial.print( value);
    Serial.print(" has  ");
    Serial.print(strlen(buffer));
    Serial.println(" digits");
    value = 123456789;
    ltoa(value, buffer, 10);
    Serial.print( value);
    Serial.print(" has  ");
    Serial.print(strlen(buffer));
    Serial.println(" digits");
    delay(1000);
  }
```

이때 버퍼의 크기는 문자열의 최대 문자 수보다 커야 한다. 16비트 10진수 정수라면 5자리 숫자와 발생 가능한 빼기 부호, 그리고 항상 문자열의 끝을 나타내는 종료 0을 포함할 수 있도록 7개의 문자가 필요하며, 32비트 long형 정수라면 10자리 숫자, 빼기 부호, 마지막으로 종료 0을 포함할 수 있도록 12개의 문자가 필요하다. 버퍼 크기를 초과하더라도 경고가 발생하지 않는다. 하지만 이렇게 되면 오버플로우로 인해 프로그램에 사용될 다른 메모리 부분 중 일부가 손상될 수 있으며, 결과적으로 알 수 없는 유형의 다양한 오류가 발생할 수 있다. 이 문제를 해결하는 가장 쉬운 방법은 항상 12개의 문자를 담을 수 있는 버퍼와 `ltoa`를 사용하는 것이다. 이 방법을 사용하면 16비트 및 32비트 값을 모두 정상적으로 처리할 수 있다.

2.9 문자열을 숫자로 변환하기

과제

문자열을 숫자로 변환하려고 한다. 통신 회선을 통해 문자열로 된 값을 받았는데, 이 값을 정수나 부동 소수점 값으로 사용해야 한다.

해결책

이 과제는 여러 가지 방법으로 해결할 수 있다. 문자열이 시리얼 데이터로 수신되는 경우라면 각 문자가 수신될 때 그 즉시 문자를 숫자로 변환할 수 있다. 시리얼 포트를 사용하여 이 작업을 수행하는 방법에 대한 자세한 설명은 레시피 4.3을 참조한다.

또는 atoi(int형 변수의 경우)나 atol(long형 변수의 경우)이라는 C 언어 변환 함수로도 숫자를 표현하는 텍스트 문자열을 변환할 수 있다.

이 코드에서는 숫자를 수신하다가 숫자가 아닌 문자가 나타나거나 버퍼가 꽉 차면 데이터 수신을 종료한다. 이 코드를 실행하려면 시리얼 모니터에서 새 행 옵션을 활성화하거나 다른 종료 문자를 입력해야 한다.

```
/*
 * StringToNumber
 * 문자열을 사용하여 숫자를 만든다.
 */

const int ledPin = 13;  // LED가 연결되어 있는 핀

int blinkDelay=0; // 이 변수에 의해 깜박임 간격이 결정된다.
char strValue[6]; // 모든 숫자와 문자열을 종료하는 0을 저장할 수 있을 정도로 충분히
                  // 커야 한다.
int index = 0;    // 수신된 숫자를 저장할 배열의 인덱스

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);  // LED 핀을 출력으로 설정한다.
}

void loop()
{
  if( Serial.available())
  {
    char ch = Serial.read();
    if(index < 5 && isDigit(ch) ){
      strValue[index++] = ch;  // ASCII 문자를 문자열에 추가한다.
    }
    else
    {
      // 버퍼가 꽉 찼거나 숫자가 아닌 데이터가 처음으로 발생한 경우
      strValue[index] = 0;  // 0을 추가하여 문자열을 종료한다.
      blinkDelay = atoi(strValue);  // atoi를 사용하여 문자열을 정수로 변환한다.
      index = 0;
    }
  }
  blink();
}
```

```
void blink()
{
  digitalWrite(ledPin, HIGH);
  delay(blinkDelay/2);  // 깜박임 간격의 1/2시간 동안 기다린다.
  digitalWrite(ledPin, LOW);
  delay(blinkDelay/2);  // 깜박임 간격의 나머지 1/2시간 동안 기다린다.
}
```

토론

atoi(ASCII를 int로) 및 atol(ASCII를 long형으로) 함수는 문자열을 정수나 long형 정수로 변환한다. 이들 함수를 사용하려면 먼저 전체 문자열을 받아서 문자 배열에 저장해야 하며, 그런 다음에야 변환 함수를 호출할 수 있다. 위 코드에서는 최대 5자리 숫자를 저장할 수 있는 strValue라는 문자 배열을 생성한다. (이 배열은 종료 널도 포함되어야 하기 때문에 char strValue[6]로 선언되었다.) 이 배열에는 숫자가 아닌 문자가 처음 나타날 때까지 Serial.read로부터 수신된 숫자가 채워진다. 그런 다음, 배열의 끝에 널이 입력된 후 atoi 함수를 통해 문자 배열이 숫자로 변환되어 blinkRate 변수에 저장된다.

마지막으로, blinkDelay에 저장된 값을 사용하는 blink 함수가 호출된다.

레시피 2.4에서 경고한 것처럼 배열의 경계를 넘지 않도록 주의해야 한다. 그 방법이 잘 떠오르지 않는다면 레시피 2.4의 토론 섹션을 참조한다.

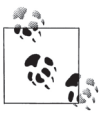
아두이노 릴리스 22에서는 문자열을 정수로 변환하는 toInt 메소드가 추가되었다.

```
String aNumber = "1234";
int value = aNumber.toInt();
```

아두이노 1.0에서는 시리얼과 이더넷(또는 Stream 클래스에서 파생된 모든 오브젝트)을 통해서도 정수 값을 가져올 수 있는 parseInt 메소드가 추가되었다. 아래 코드는 숫자 시퀀스를 숫자로 변환한다. 이 코드는 해결책 코드와 비슷하지만 버퍼를 사용하지 않는다. (그리고 숫자의 개수도 5로 제한되지 않는다.)

```
int blinkDelay=0;  // 이 변수에 의해 깜박임 간격이 결정된다.
void loop()
{
  if( Serial.available())
  {
```

```
    blinkRate = Serial.parseInt();
  }
  blink();
}
```



parseInt 등과 같은 스트림 구문 분석 메소드에서는 원하는 시간 내에 데이터가 수신되지 않았을 때 스케치에게 컨트롤을 돌려주기 위해 제한 시간을 사용한다. 기본적인 제한 시간은 1초이지만 다음과 같이 setTimeout 메소드를 호출하여 제한 시간을 변경할 수 있다.

```
Serial.setTimeout(1000 * 60);  // 최대 1분까지 기다린다.
```

parseInt(및 기타 모든 스트림 메소드)는 구분 기호가 수신되지 않으면 제한 시간 이전까지 수신된 값을 리턴한다. 리턴값은 수집된 값으로 구성되며, 숫자가 수신되지 않은 경우에는 0이 리턴된다. 아두이노 1.0에는 아직까지 구문 분석 메소드의 제한 시간 초과 여부를 확인할 수 있는 방법이 없지만 후속 릴리스에서 이 기능을 제공할 예정이다.

참고

atoi 관련 문서는 http://www.nongnu.org/avr-libc/user-manual/group__avr__stdlib.html에서 찾아볼 수 있다.

http://www.cplusplus.com/reference/clibrary/cstdlib/atoi/와 http://www.cppreference.com/wiki/string/c/atoi를 비롯한 여러 온라인 C/C++ 레퍼런스 페이지에서 이와 같은 하위 레벨 함수에 대한 설명을 볼 수 있다.

레시피 4.3과 레시피 4.5에서 parseInt와 시리얼을 함께 사용하는 방법에 대한 자세한 설명을 볼 수 있다.

2.10 코드를 기능 블록으로 구조화하기

과제

스케치에 함수를 추가하는 방법과 함수에서 어느 정도의 기능을 처리해야 하는지 그 적정 수준을 알고 싶다. 그리고 스케치의 전체 구조를 계획하는 방법에 대해서도 알고 싶다.

해결책

함수(functions)는 스케치에서 수행할 작업을 기능 블록으로 구성하는 데 사용된다. 함수는 잘 정의된 **입력**(inputs, 함수에 제공되는 정보)과 **출력**(outputs, 함수에서 제공하는 정보)으로 그 기능을 그룹화하며, 이러한 그룹화는 코드를 쉽게 구조화, 관리 및 재사용할 수 있다는 장점을 제공한다. 그리고 모든 아두이노 스케치에 사용되는 `setup`과 `loop`라는 두 가지 함수도 앞에서 자주 볼 수 있었다. 함수를 만들려면 **리턴 유형**(return type, 함수가 제공하는 정보)과 함수 이름 그리고 함수가 호출될 때 함수에 전달되는 선택적 매개변수(값)를 선언해야 한다.

함수(functions)와 **메소드**(methods)라는 용어는 프로그램 내에서 단일 항목으로 호출할 수 있는 잘 정의된 코드 블록을 의미한다. C 언어에서는 이를 함수라고 부르고 있으며, C++와 같이 클래스를 통해 기능을 제공하는 오브젝트 지향 언어에서는 메소드라는 용어를 사용한다. 아두이노에서는 두 가지 스타일을 모두 사용하고 있다. (예제 스케치에서는 주로 C 형태의 스타일이 사용되며, 라이브러리는 주로 C++ 클래스 메소드를 노출하기 위해 작성된 것이다.) 이 책에서는 코드가 클래스를 통해 노출되는 경우가 아니라면 함수라는 용어를 주로 사용한다. 두 용어를 구분하기가 쉽지 않은 경우에는 편하게 두 용어를 같은 의미로 생각해도 된다.

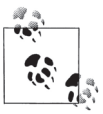
다음은 LED를 깜박이게 하는 간단한 함수다. 이 함수에는 매개변수가 없으며, 리턴하는 값도 없다. (함수 이름 앞의 `void`는 함수의 리턴값이 없음을 나타낸다.)

```
// LED를 한 번 켰다가 끈다.
void blink1()
{
   digitalWrite(13, HIGH); // LED를 켠다.
   delay(500);             // 500밀리초 동안 대기한다.
   digitalWrite(13, LOW);  // LED를 끈다.
   delay(500);             // 500밀리초 동안 대기한다.
}
```

아래 버전에서는 매개변수(`count`라는 이름의 정수)를 사용하여 LED의 깜박임 횟수를 결정한다.

```
// count 매개변수에 지정된 횟수만큼 LED를 켰다가 끄기를 반복한다.
void blink2(int count)
{
  while(count > 0 ) // count가 0보다 크지 않을 때까지 반복한다.
  {
```

```
    digitalWrite(13, HIGH);
    delay(500);
    digitalWrite(13, LOW);
    delay(500);
    count = count -1;  // count를 줄인다.
  }
}
```



숙련된 개발자라면 컴파일러가 매개변수에 사용된 값의 유형에 따라 값을 구별하기 때문에 두 함수 모두 깜박이게 된다는 것을 알고 있을 것이다. 이를 **함수 오버로딩**(function overloading)이라고 한다. 일반적인 예로, 레시피 4.2에서 설명할 아두이노의 `print` 함수가 있다. 레시피 4.6에서도 오버로딩에 대한 또 하나의 예를 볼 수 있다.

이 버전에서는 `count`의 값이 0인지 여부를 검사하여 0이 아니면 LED를 켰다가 끈 다음 `count`의 값을 1씩 줄인다. 이 과정은 `count`가 0보다 크지 않을 때까지 반복된다.

일부 문서에서는 **매개변수**(parameter)를 **인수**(argument)라고도 한다. 하지만 실제로는 두 용어를 동일한 의미로 사용할 수도 있다.

이제 매개변수를 사용하고 값을 리턴하는 예제 스케치를 살펴보자. 이 스케치에서는 매개변수를 사용하여 LED의 켜짐 시간과 꺼짐 시간을 밀리초 단위로 지정한다. 그리고 단추를 누르기 전까지 LED의 깜박임이 지속되다가 단추를 누르면 LED의 깜박임 횟수가 리턴된다.

```
/*
  blink3 스케치
  매개변수가 있는 함수를 호출하고, 이 함수에서 값을 리턴하는 방법을 보여 준다.
  레시피 5.2의 풀업 스케치와 동일한 배선을 사용한다.

  디지털 핀 2에 연결된 스위치를 누르면 프로그램이 시작되고 중지될 때 LED가 깜박인다.
  마지막으로 LED의 깜박임 횟수가 표시된다.
 */

const int ledPin = 13;      // LED 출력 핀
const int inputPin = 2;     // 스위치 입력 핀

void setup() {
  pinMode(ledPin, OUTPUT);
```

```
  pinMode(inputPin, INPUT);
  digitalWrite(inputPin, HIGH);  // 내부 풀업 저항을 사용한다(레시피 5.2).
  Serial.begin(9600);
}

void loop(){
  Serial.println("Press and hold the switch to stop blinking");
  int count = blink3(250);  // LED를 250ms 동안 켰다가 250ms 동안 끈다.
  Serial.print("The number of times the switch blinked was ");
  Serial.println(count);
}

// 지정된 지연 기간 동안 LED를 켰다가 끈다.
// LED의 깜박임 횟수를 리턴한다.
int blink3(int period)
{
  int result = 0;
  int switchVal = HIGH;    // 풀업 저항을 사용하며, 이 값은 스위치를 누르지 않은 경우
                           // HIGH가 된다.
  while(switchVal == HIGH) // 스위치를 누를 때까지 루프를 반복한다.
                           // (스위치를 누르면 LOW로 변경된다.)
  {
    digitalWrite(13, HIGH);
    delay(period);
    digitalWrite(13, LOW);
    delay(period);
    result = result + 1;  // 계수를 증가시킨다.
    switchVal = digitalRead(inputPin);  // 입력값을 읽는다.
  }
  // 스위치를 눌렀기 때문에 switchVal이 더 이상 HIGH가 아닌 경우
  return result;  // 이 값이 리턴된다.
}
```

토론

이 레시피의 해결책에서는 앞으로 자주 보게 될 세 가지 형태의 함수 호출을 보여 준다. 첫 번째 함수인 blink1에는 매개변수도 없고 리턴값도 없으며, 다음과 같은 형식을 갖는다.

```
void blink1()
{
  // 구현 코드
}
```

blink2의 경우에는 한 개의 매개변수만 있고 리턴값이 없다.

```
void blink2(int count)
{
  // 구현 코드
}
```

blink3의 경우에는 매개변수가 한 개이고, 리턴되는 값이 있다.

```
int blink3(int period)
{
  // 구현 코드
}
```

함수 이름 앞에 있는 데이터 유형은 리턴 유형을 나타낸다(void의 경우 리턴 유형이 없음). **함수를 선언**(declaring the function)할 때는(함수와 그 동작을 정의하는 코드를 작성할 때) 맨 끝에 있는 괄호 뒤에 세미콜론을 입력하지 않는다. 하지만 함수를 **사용**(use, 호출)할 때는 함수를 호출하는 행의 맨 끝에 세미콜론을 입력해야 한다.

앞으로 보게 될 대부분의 함수는 이러한 세 가지 형태를 바탕으로 하면서 약간 변형된 형태의 함수다. 예를 들어, 다음은 매개변수 한 개를 사용하고 값을 리턴하는 함수다.

```
int sensorPercent(int pin)
{
int percent;

  int val = analogRead(pin);  // 센서를 판독한다(0부터 1023까지의 범위).
  percent = map(val, 0, 1023, 0, 100);  // 백분율의 범위는 0부터 100이다.
  return percent;
}
```

이 함수의 이름은 sensorPercent다. 이 함수는 판독할 아날로그 핀 번호를 받아서 백분율 값을 리턴한다(analogRead와 map에 대한 자세한 설명은 레시피 5.7 참조). 선언부의 맨 앞에 있는 int는 정수를 리턴하는 함수임을 나타낸다. 함수를 만들 때는 수행할 작업에 적합한 리턴 유형을 선택해야 한다. 이 함수는 0부터 100 사이의 정수 값을 리턴하므로 이 함수의 리턴 유형은 int가 적합하다.

함수에는 의미 있는 이름을 사용하는 것이 효과적이며, 일반적으로 첫 번째 단어를 제외하고 각 단어의 첫 번째 문자를 대문자로 표시하는 방식으로 여러 단어를 결합해서 사용하고 있다. 하지만 어느 방법을 사용하든지 이름 지정 스타일을 일관되게만 유지하면 이해하기 쉬운 코드를 작성할 수 있다.

sensorPercent에는 pin이라는 매개변수가 있다. (함수가 호출될 때 함수에 전달된 값이 pin에 지정된다.)

함수의 본문(중괄호 내의 코드)에서는 원하는 작업을 수행한다. 이 예제의 경우에는 아날로그 입력 핀의 값을 읽은 후 백분율로 변환한다. 앞의 예제에서 백분율은 percent라는 변수에 임시로 저장된다. 그런 다음 아래 구문에서 임시 변수인 percent에 저장되어 있는 값이 호출 애플리케이션에 리턴된다.

```
return percent;
```

다음과 같이 임시 변수를 사용하지 않고도 동일한 작업을 수행할 수 있다.

```
int sensorPercent(int pin)
{
  int val = analogRead(pin);        // 센서를 판독한다(0부터 1023까지의 범위).
  return map(val, 0, 1023, 0, 100);        // 백분율의 범위는 0부터 100이다.
}
```

이제 다음과 같이 함수를 호출할 수 있다.

```
// 아날로그 핀 6개의 백분율 값을 인쇄한다.
for(int sensorPin = 0; sensorPin < 6; sensorPin++)
{
  Serial.print("Percent of sensor on pin ");
  Serial.print(sensorPin);
  Serial.print(" is ");
  int val = sensorPercent(sensorPin);
  Serial.print(val);
}
```

참고

아두이노 함수 레퍼런스 페이지: http://www.arduino.cc/en/Reference/FunctionDeclaration

2.11 하나의 함수에서 두 개 이상의 값 리턴하기

과제

하나의 함수에서 두 개 이상의 값을 리턴하고 싶다. 레시피 2.10에서는 가장 일반적인 형태의 함수, 즉 하나의 값을 리턴하거나 값을 리턴하지 않는 함수에 대한 예제를 살펴보았다. 하지만 두 개 이상의 값을 리턴해야 하는 경우도 발생할 수 있다.

해결책

이 과제는 여러 가지 방법으로 해결할 수 있다. 아래와 같이 전역 변수를 서로 바꾸는 함수를 살펴보면 쉽게 이해할 수 있을 것이다. 이 함수에서는 아무것도 리턴하지 않는다.

```
/*
 swap 스케치
 전역 변수를 사용하여 두 값을 변경한다.
 */

int x;  // x와 y는 전역 변수다.
int y;

void setup() {
  Serial.begin(9600);
}

void loop(){
  x = random(10);  // 난수를 선택한다.
  y = random(10);

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);
  swap();

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y); Serial.println();


  delay(1000);
}
```

```
// 두 개의 전역 변수를 서로 바꾼다.
void swap()
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

swap 함수는 전역 변수를 사용하여 두 값을 서로 바꾼다. 전역 변수는 어디에서나 액세스할 수 있고 자유롭게 변경할 수 있기 때문에 이해하기 쉽다는 장점이 있기는 하지만 숙련된 프로그래머들은 이 변수를 잘 사용하지 않는다. 왜냐하면 실수로 변수의 값을 수정하는 경우도 발생할 수 있고, 스케치 내에서 전역 변수의 이름이나 유형을 변경했을 경우 함수가 작동하지 않는 문제도 발생할 수 있기 때문이다.

좀 더 일반적으로 사용되는 안전한 방법은 변경하려는 값에 대한 참조를 함수에 전달한 후 함수에서 참조를 사용하여 값을 수정하는 것이다. 아래 예제에서 그 방법을 확인할 수 있다.

```
/*
 functionReferences 스케치
 참조를 전달하여 두 개 이상의 값을 리턴하는 방법을 보여 준다.
 */

void setup() {
  Serial.begin(9600);
}

void loop(){
  int x = random(10);  // 난수를 선택한다.
  int y = random(10);

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);
  swap(x, y);

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y); Serial.println();

  delay(1000);
}
```

```
// 지정된 두 값을 서로 바꾼다.
void swap(int &value1, int &value2)
{
  int temp;
  temp = value1;
  value1 = value2;
  value2 = temp;
}
```

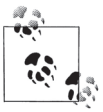
토론

이 swap 함수는 레시피 2.10에서 살펴보았던 매개변수가 있는 함수와 비슷하지만 해당 매개변수가 참조(references)임을 나타내는 앰퍼샌드(&) 기호를 사용하고 있다는 점이 다르다. 이는 곧 함수 내에서 값을 변경할 경우 함수가 호출될 때 지정되는 변수의 값도 변경된다는 것을 의미한다. 우선 이 레시피의 코드를 실행하여 매개변수가 서로 바뀌었는지 확인해 보자. 그런 다음 함수 정의 부분에 있는 두 개의 앰퍼샌드를 제거해 보자.

변경된 코드는 다음과 같다.

```
void swap(int value1, int value2)
```

위 코드를 실행하면 값이 서로 교체되지 않는다. 이는 함수 내에서 변경된 사항이 로컬 변경 사항이기 때문에 함수가 리턴될 때 사라지기 때문이다.



아두이노 릴리스 21 이전의 릴리스를 사용하고 있다면 함수 선언을 만들어서 컴파일러에게 참조를 사용하는 함수라는 사실을 알려 주어야 한다. 이 책 초판의 다운로드에 포함되어 있는 이 레시피의 스케치를 보면 함수 선언을 만드는 방법을 알 수 있다.

```
// 참조가 있는 함수는 사용하기 전에 명시적으로 선언해야 한다.
// 선언은 앞쪽에 있어야 하며, 특히 setup 및 loop 코드보다 더 먼저 나와야 한다.
// 선언의 맨 끝에 세미콜론이 있어야 한다.
void swap(int &value1, int &value2);
```

함수 선언은 **원형**(prototype)으로, 함수의 이름, 함수에 전달할 값의 유형 및 함수의 리턴 유형으로 구성되어 있다. 이러한 함수 선언은 대개 아두이노 빌드 프로세스에 의해 자동으로 작성되지만, 비표준(아두이노 21 이전 버전) 아두이노 구문을 사용할 경우에는 선언이 자동으로 작성되지 않으므로 이 예제에서 살펴본 setup 바로 앞의 행과 같이 함수 선언을 코드에 직접 추가해야 한다.

함수 정의는 함수 헤더와 함수 본문으로 구성되어 있다. 함수 헤더는 맨 끝에 세미콜론이 없다는 점만 제외하면 선언과 비슷하다. 함수 본문은 중괄호 안에 들어 있는 코드로 함수가 호출될 때 특정 작업을 수행하기 위해 실행되는 부분이다.

2.12 조건에 따라 작업 수행하기

과제

특정 조건이 충족된 경우에만 코드 블록을 실행하고 싶다. 예를 들어, 스위치를 눌렀거나 아날로그 값이 특정 임계값을 초과한 경우에만 LED를 켜고 싶다.

해결책

다음 코드에서는 레시피 5.1의 배선을 사용한다.

```
/*
  Pushbutton 스케치
  디지털 핀 2에 연결된 스위치로 핀 13에 연결된 LED를 켠다.
*/

const int ledPin = 13;              // LED용 핀을 선택한다.
const int inputPin = 2;             // 입력 핀을 선택한다(누름 단추).

void setup() {
  pinMode(ledPin, OUTPUT);          // LED 핀을 출력으로 선언한다.
  pinMode(inputPin, INPUT);         // 누름 단추 핀을 입력으로 선언한다.
}

void loop(){
  int val = digitalRead(inputPin);  // 입력값을 읽는다.
  if (val == HIGH)                  // 입력이 HIGH인지 검사한다.
  {
    digitalWrite(ledPin, HIGH);     // 스위치가 눌려져 있으면 LED를 켠다.
  }
}
```

토론

`if` 문에서는 `digitalRead`의 값을 테스트한다. `if` 문의 괄호 안에서 테스트가 수행되며, 그 결과 값은 true 또는 false일 수 있다. 이 레시피의 솔루션 예제에서는 `val == HIGH`라는 조건을 검사하며, `if` 문 뒤의 코드는 앞의 조건이 true인 경우에만 실행된다. 코드 블록이란 중괄호 안에 포함된 모든 코드를 의미하며, 중괄호가 없는 경우에는 세미콜론으로 끝나는 다음 실행 명령문이 하나의 블록이 된다.

조건이 true일 때 한 가지 작업을 수행하고, 조건이 false일 때 또 다른 작업을 수행하려는 경우에는 다음과 같이 `if...else` 문을 사용할 수 있다.

```
/*
  Pushbutton 스케치
  핀 2에 연결된 스위치로 핀 13에 연결된 LED를 켠다.
*/

const int ledPin = 13;              // LED용 핀을 선택한다.
const int inputPin = 2;             // 입력 핀을 선택한다(누름 단추).

void setup() {
  pinMode(ledPin, OUTPUT);          // LED 핀을 출력으로 선언한다.
  pinMode(inputPin, INPUT);         // 누름 단추 핀을 입력으로 선언한다.
}

void loop(){
  int val = digitalRead(inputPin);  // 입력값을 읽는다.
  if (val == HIGH)                  // 입력이 HIGH인지 검사한다.
  {
    // val이 HIGH일 때 수행되는 코드
    digitalWrite(ledPin, HIGH);     // 스위치가 눌려져 있으면 LED를 켠다.
  }
  else
  {
    // val이 HIGH가 아닐 때 수행되는 코드
    digitalWrite(ledPin, LOW);      // LED를 끈다.
  }
}
```

참고

레시피 2.2의 부울 유형에 대한 설명을 참조한다.

2.13 명령문 시퀀스 반복하기

과제

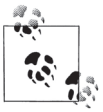
표현식이 true인 동안 명령문 블록을 반복하고 싶다.

해결책

while 루프는 표현식이 true인 동안 하나 이상의 명령문을 반복한다.

```
/*
 * Repeat 스케치
 * 조건이 true인 동안 깜박인다.
 */

const int ledPin = 13;      // LED가 연결되어 있는 디지털 핀
const int sensorPin = 0;    // 아날로그 입력 0

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);  // LED 핀을 출력으로 설정한다.
}

void loop()
{
  while(analogRead(sensorPin) > 100)
  {
    blink();  // LED를 켰다가 끄는 함수를 호출한다.
    Serial.print(".");
  }
  Serial.println(analogRead(sensorPin));     // 이 행은 while 루프가 끝날 때까지
                                             // 실행되지 않는다.
}

void blink()
{
  digitalWrite(ledPin, HIGH);
  delay(100);
  digitalWrite(ledPin, LOW);
  delay(100);
}
```

이 코드의 경우에는 analogRead의 값이 100보다 큰 동안 {}(중괄호) 안에 들어 있는 코드 블록이 실행된다. 이 코드는 어떤 값이 임계값을 초과했을 때 LED를 켜서 알려주려는 경우에 사용할 수 있다. 그런 다음 센서 값이 100 이하로 떨어지면 LED가 꺼지고, 값이 100보다 큰 경우에는 LED가 계속해서 깜박인다.



코드 블록을 정의하는 {} 기호에는 여러 가지 이름이 있지만 이 책에서는 {} 기호를 중괄호라고 부른다.

토론

중괄호는 루프 내에서 실행할 코드 블록의 범위를 정의한다. 다음과 같이 중괄호를 사용하지 않으면 코드의 첫 번째 행만 루프 내에서 반복된다.

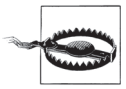
```
while(analogRead(sensorPin) > 100)
  blink();  // 루프 표현식 바로 뒤의 행만 실행된다.
  Serial.print(".");  // 이 행은 while 루프가 끝날 때까지 실행되지 않는다.
```

두 행 이상으로 구성된 코드의 경우 루프에 중괄호가 없으면 예기치 않은 결과가 발생할 수도 있다.

do...while 루프는 while 루프와 비슷하지만 코드 블록의 명령이 일단 실행된 후 조건이 검사된다는 점이 다르다. 이 루프는 표현식이 false가 되더라도 적어도 한 번은 코드를 실행해야 하는 경우에 사용한다.

```
do
{
  blink();  // LED를 켰다가 끄는 함수를 호출한다.
}
while (analogRead(sensorPin) > 100);
```

앞의 코드를 실행하면 LED가 적어도 한 번은 켜졌다가 꺼지고, 센서의 값이 100보다 크면 계속해서 깜박인다. 값이 100보다 크지 않은 경우에는 LED가 한 번만 깜박이게 된다. 이 코드는 배터리 충전 회로에서 사용할 수 있다. 예를 들어, 이 함수가 10초마다 호출된다고 가정할 경우 LED가 10초에 한 번씩만 깜박인다면 충전 중이라는 의미이고, 계속해서 깜박인다면 충전이 완료되었다는 의미가 될 것이다.



while 또는 do 루프 내의 코드만 종료 조건이 갖추어질 때까지 실행된다. 제한 시간, 센서 상태 또는 기타 입력에 대한 응답으로 루프를 종료해야 하는 경우에는 다음과 같이 break 문을 사용할 수 있다.

```
while(analogRead(sensorPin) > 100)
{
  blink();
  if(Serial.available())
    break;  // 임의의 시리얼 입력이 발생하면 while 루프가 종료된다.
}
```

참고

4장 및 5장

2.14 카운터를 사용하여 명령문 반복하기

과제

하나 이상의 명령문을 특정 횟수 동안 반복하고 싶다. 이 경우 while 루프와 유사한 for 루프를 사용할 수 있다. 하지만 for 루프를 사용하려면 시작 및 종료 조건을 자세히 제어해야 한다.

해결책

이 스케치에서는 for 루프를 사용하여 변수 i의 값을 출력하는 과정을 통해 0부터 4까지 카운트하는 방법을 보여 준다.

```
/*
   ForLoop 스케치
   for 루프의 사용법을 보여 준다.
*/

void setup() {
  Serial.begin(9600);}

void loop(){
  Serial.println("for(int i=0; i < 4; i++)");
```

```
  for(int i=0; i < 4; i++)
  {
    Serial.println(i);
  }
}
```

이 스케치의 시리얼 모니터 출력은 다음과 같다. (이 결과는 반복해서 표시된다.)

```
for(int i=0; i < 4; i++)
0
1
2
3
```

토론

for 루프는 초기화 부분, 조건 테스트 부분 그리고 반복 부분(루프를 통과할 때마다 마지막으로 실행되는 명령문) 이렇게 세 부분으로 구성되어 있다. 각 부분은 세미콜론으로 구분된다. 이 레시피의 해결책에 있는 코드에서 int i=0;은 변수 i를 0으로 초기화하고, i < 4;는 변수 i가 4보다 작은지 여부를 테스트하며, i++는 i의 값을 증가시킨다.

for 루프에서는 기존 변수를 사용할 수도 있지만 새로운 변수를 생성하여 루프 내에서만 배타적으로 사용할 수도 있다. 여기에서는 앞부분에서 생성한 변수 j의 값을 사용한다.

```
int j;

Serial.println("for(j=0; j < 4; j++ )");
for(j=0; j < 4; j++ )
{
  Serial.println(j);
}
```

이번 예제는 앞의 예제와 거의 비슷하지만 변수 j가 이미 정의되어 있기 때문에 초기화 부분에 int 키워드가 없다는 점만 다르다. 다음과 같이 이번 예제의 출력은 이전 예제의 출력과 비슷하다.

```
for(j=0; i < 4; i++)
0
```

```
1
2
3
```

루프에서 이미 정의된 변수의 값을 사용할 경우에는 초기화 부분을 완전히 생략할 수도 있다. 아래 코드의 경우 j의 값이 1인 상태에서 루프가 시작된다.

```
int j = 1;

Serial.println("for( ; j < 4; j++ )");
for( ; j < 4; j++ )
{
  Serial.println(j);
}
```

위 코드의 출력은 다음과 같다.

```
for( ; j < 4; j++ )
1
2
3
```

조건 테스트 부분에서 루프의 종료 시점을 제어할 수 있다. 앞의 예제들에서는 루프의 변수가 4보다 작은지 여부를 테스트하여 조건이 더 이상 맞지 않으면 루프가 종료되었다.

루프의 변수를 0부터 시작하면서 루프를 4번 반복하려면 조건 명령문에서 값이 4보다 작은지 여부를 테스트해야 한다. 왜냐하면 루프의 변수가 4보다 작은 네 개의 값(0, 1, 2 및 3)일 때 루프가 실행되기 때문이다.

아래 코드에서는 루프 변수의 값이 4보다 작거나 같은지 여부를 테스트하고 0부터 4까지 출력한다.

```
Serial.println("for(int i=0; i <= 4; i++)");
for(int i=0; i <= 4; i++)
{
  Serial.println(i);
}
```

for 루프의 세 번째 부분은 루프를 통과할 때마다 마지막으로 실행되는 반복 명령문

이다. 이 부분에는 모든 유효한 C/C++ 명령문을 사용할 수 있다. 아래 코드에서는 루프를 통과할 때마다 i의 값이 2씩 증가한다.

```
Serial.println("for(int i=0; i < 4; i+= 2)");
for(int i=0; i < 4; i+=2)
{
  Serial.println(i);
}
```

이 경우에는 0과 2만 출력된다.

반복기(iterator) 표현식은 아래와 같이 3부터 시작해서 0에서 끝나는 것처럼 높은 값부터 낮은 값으로 카운트를 세면서 루프를 반복할 때도 사용할 수 있다.

```
Serial.println("for(int i=3; i > = 0 ; i--)");
for(int i=3; i > = 0 ; i--)
{
  Serial.println(i);
}
```

for 루프의 다른 부분과 마찬가지로 반복기 표현식도 비워 둘 수 있다. (세 부분이 모두 비어 있더라도 두 개의 세미콜론을 사용하여 세 부분을 항상 구분해 줘야 한다.)

아래 버전에서는 입력 핀이 HIGH일 때만 i 값이 증가한다. for 루프에서는 i의 값이 변경되지 않는다. 대신 Serial.print 뒤의 if 명령문에서 i의 값이 변경된다. 여기에서는 inPin을 정의하고, pinMode()를 사용하여 inPin을 INPUT으로 설정해야 한다.

```
Serial.println("for(int i=0; i < 4; )");
for(int i=0; i < 4; )
{
  Serial.println(i);
  if(digitalRead(inPin) == HIGH) {
    i++;  // 입력이 HIGH인 경우에만 값이 증가한다.
  }
}
```

참고

for 명령문에 대한 아두이노 레퍼런스: http://www.arduino.cc/en/Reference/For

2.15 루프 종료하기

과제

테스트 중인 특정 조건에 따라 루프를 조기에 종료하고 싶다.

해결책

다음 코드를 살펴보자.

```
while(analogRead(sensorPin) > 100)
{
  if(digitalRead(switchPin) == HIGH)
  {
    break;          // 스위치를 누르면 루프가 종료된다.
  }
  flashLED();       // LED를 켰다가 끄는 함수를 호출한다.
}
```

토론

이 코드는 `while` 루프를 사용하는 코드와 유사하지만, 디지털 핀이 `HIGH`일 경우 `break` 명령문에 의해 루프가 종료된다는 점이 다르다. 예를 들어, 레시피 5.1과 같이 스위치가 핀에 연결되어 있다면 `while` 루프의 조건이 true일지라도 루프가 종료되면서 LED의 깜박임도 멈추게 된다.

참고

`break` 명령문에 대한 아두이노 레퍼런스: http://www.arduino.cc/en/Reference/Break

2.16 단일 변수를 기반으로 다양한 작업 수행하기

과제

어떤 값에 따라 다양한 작업을 수행하고 싶다. `if` 및 `else if` 명령문을 여러 번 사용할 수 있겠지만, 그럴 경우 코드가 복잡해서 이해하기도 어렵고 수정하기도 어렵다. 게다가 값 범위도 테스트하고 싶다.

해결책

switch 명령문을 사용하면 다양한 대안을 선택할 수 있다. 이 명령문은 if/else if 명령문과 비슷하기는 하지만 훨씬 간결하다는 특징이 있다.

```
/*
 * SwitchCase 스케치
 * 시리얼 포트의 문자를 전환하는 방식으로 switch 명령문을 보여 주는 예제
 *
 * 문자 1을 보내면 LED가 한 번 깜박이고, 문자 2를 보내면 2번 깜박인다.
 * +를 보내면 LED가 켜지고, -를 보내면 LED가 꺼진다.
 * 나머지 문자는 모두 시리얼 모니터에 메시지를 출력한다.
 */
const int ledPin = 13;  // LED가 연결되어 있는 핀

void setup()
{
  Serial.begin(9600);  // 9600보(baud)로 데이터를 송수신하도록 시리얼 포트를 초기화한다.
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  if ( Serial.available()) // 사용 가능한 문자가 하나 이상 있는지 검사한다.
  {
    char ch = Serial.read();
    switch(ch)
    {
    case '1':
      blink();
      break;
    case '2':
      blink();
      blink();
      break;
    case '+':
      digitalWrite(ledPin, HIGH);
      break;
    case '-':
      digitalWrite(ledPin, LOW);
      break;
    default :
      Serial.print(ch);
      Serial.println(" was received but not expected");
```

```
      break;
    }
  }
}

void blink()
{
  digitalWrite(ledPin, HIGH);
  delay(500);
  digitalWrite(ledPin, LOW);
  delay(500);
}
```

토론

switch 명령문은 시리얼 포트로부터 수신된 변수 ch를 평가한 후 변수의 값과 일치하는 레이블로 분기한다. 레이블은 숫자 상수여야 하며(case 명령문에서 문자열을 사용할 수 있음), 서로 같은 값을 가진 레이블이 있으면 안 된다. 다음과 같이 각 표현식 뒤에 break 명령문이 없으면 이후 구문이 순차적으로 실행된다.

```
case '1':
  blink();          // 다음 레이블 앞에 break 명령문이 없다.
case '2':
  blink();          // 여기에서도 case '1'이 실행된다.
  blink();
break;              // break 명령문이 있으므로 switch 표현식이 종료된다.
```

앞의 코드에서 보듯이 case '1':의 끝에 있는 break 명령문이 없으면 ch의 값이 문자 1임에도 불구하고 blink 함수가 세 번 호출된다. 실수로 break를 입력하지 않은 경우도 자주 발생하지만 편의를 위해 break를 의도적으로 입력하지 않는 경우도 종종 있다. 이 경우 다른 사람이 코드를 읽을 때 혼란스러울 수도 있으므로 되도록이면 코드에 주석을 달아서 의도를 명확히 알려 주는 것이 좋다.

switch 명령문이 의도대로 작동하지 않을 경우에는 break 명령문이 빠지지 않았는지 확인하기 바란다.

default: 레이블은 일치하는 case 레이블이 없는 값을 처리할 때 사용된다. default 레이블도 없고 switch 표현식에 일치하는 case 레이블도 없으면 아무 작업도 수행

되지 않는다.

참고

switch 및 case 명령문에 대한 아두이노 레퍼런스: http://www.arduino.cc/en/Reference/SwitchCase

2.17 문자 및 숫자 값 비교하기

과제

다양한 값의 상호 관계를 알고 싶다.

해결책

표 2-3에서는 관계 연산자를 사용하여 정수 값을 비교한다.

표 2-3 관계 및 같음 연산자

연산자	테스트 내용	예
==	같음	2 == 3 // false로 평가
!=	같지 않음	2 != 3 // true로 평가
>	보다 큼	2 > 3 // false로 평가
<	보다 작음	2 < 3 // true로 평가
>=	크거나 같음	2 >= 3 // false로 평가
<=	작거나 같음	2 <= 3 // true로 평가

다음 스케치에서는 비교 연산자의 실행 결과를 보여 준다.

```
/*
 * RelationalExpressions 스케치
 * 값을 비교한다.
 */

int i = 1;  // 일부 값을 시작할 때 지정한다.
int j = 2;
```

```
void setup() {
  Serial.begin(9600);
}

void loop(){
  Serial.print("i = ");
  Serial.print(i);
  Serial.print(" and j = ");
  Serial.println(j);

  if(i < j)
    Serial.println(" i is less than j");
  if(i <= j)
    Serial.println(" i is less than or equal to j");
  if(i != j)
    Serial.println(" i is not equal to j");
  if(i == j)
    Serial.println(" i is equal to j");
  if(i >= j)
    Serial.println(" i is greater than or equal to j");
  if(i > j)
    Serial.println(" i is greater than j");

  Serial.println();
  i = i + 1;
  if(i > j + 1)
    delay(10000);  // i가 더 이상 j와 가까워지지 않으면 지연 시간이 길어진다.
}
```

다음은 출력 결과다.

```
i = 1 and j = 2
 i is less than j
 i is less than or equal to j
 i is not equal to j

i = 2 and j = 2
 i is less than or equal to j
 i is equal to j
 i is greater than or equal to j

i = 2 and j = 3
 i is not equal to j
 i is greater than or equal to j
 i is greater than j
```

토론

같음 연산자는 이중 등호(==)라는 점에 주의해야 한다. 같음 연산자와 단일 등호를 사용하는 할당 연산자를 혼동해서 실수를 범하게 되는 경우가 상당히 많다.

i의 값과 3을 비교하는 다음 표현식을 보자. 이 표현식이 프로그래머가 원래 의도했던 것이다.

```
if(i == 3)  // i가 3과 같은지 테스트한다.
```

하지만 스케치에 실제로 입력한 표현식은 다음과 같다.

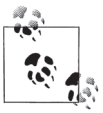
```
if(i = 3) // 실수로 하나의 등호만 입력했다.
```

이 표현식은 항상 true를 리턴한다. 왜냐하면 i가 3으로 설정될 것이고, if 조건문 안에 값이 존재하기 때문이다.

다음과 같이 상수를 표현식의 왼쪽에 놓으면 이러한 실수를 걱정할 필요 없이 변수와 상수(고정 값)를 비교할 수 있다.

```
if(3 = i) // 실수로 하나의 등호만 입력했다.
```

이렇게 하면 상수에 다른 값을 할당할 수 없기 때문에 컴파일 타임에 오류가 표시된다.



이 경우 "value required as left operand of assignment"라는 조금 낯선 오류 메시지가 표시된다. 이 메시지가 표시되면 변경할 수 없는 항목에 값을 할당하려고 시도 중이라는 의미로 받아들이면 된다.

참고

조건 및 비교 연산자에 대한 아두이노 레퍼런스: http://www.arduino.cc/en/Reference/If

2.18 문자열 비교하기

과제

문자열 두 개가 서로 같은지 확인하고 싶다.

해결책

strcmp(string compare의 약어)라는 함수를 이용해서 문자열을 비교할 수 있다. 다음은 이 함수를 사용하는 예이다.

```
char string1[ ] = "left";
char string2[ ] = "right";

if(strcmp(string1, string2) == 0)
   Serial.print("strings are equal)
```

토론

strcmp는 문자열이 서로 같으면 0을 리턴한다. 그리고 문자열이 일치하지 않을 경우에는 두 문자열에서 일치하지 않는 첫 번째 문자를 비교하여 첫 번째 문자열의 문자가 크면 0보다 큰 값을 리턴하고, 두 번째 문자열의 문자가 크면 0보다 작은 값을 리턴한다. 같은지 여부만 확인하는 경우가 대부분이기 때문에 0을 테스트하는 것이 처음에는 어색하게 느껴질 수도 있겠지만 곧 익숙해질 것이다.

길이가 서로 다른 문자열의 경우에는 짧은 문자열이 긴 문자열에 포함되어 있다고 하더라도 같은 문자열로 간주되지 않는다. 다음 예를 살펴보자.

```
strcmp("left", "leftcenter") == 0) // false로 평가된다.
```

strncmp 함수를 사용하면 비교할 최대 문자 수를 지정하여 문자열을 비교할 수 있다. 다음과 같이 strncmp에 최대 문자 수를 지정하여 비교하면 많은 문자가 남아 있어도 지정된 개수의 문자만 비교하고 함수가 종료된다.

```
strncmp("left", "leftcenter", 4) == 0) // true로 평가된다.
```

문자열과는 달리 아두이노 String은 다음과 같이 직접 비교할 수 있다.

```
String stringOne = String("this");
```

```
if (stringOne == "this")
  Serial.println("this will be true");
if (stringOne == "that")
  Serial.println("this will be false");
```

http://arduino.cc/en/Tutorial/StringComparisonOperators에서 아두이노 `String` 비교에 대한 자습서를 볼 수 있다.

참고

http://www.cplusplus.com/reference/clibrary/cstring/strcmp/에서 `strcmp`에 대한 자세한 정보를 볼 수 있다.

레시피 2.5에서 아두이노 `String`에 대한 설명을 볼 수 있다.

2.19 논리 비교 수행하기

과제

두 개 이상의 표현식 간의 논리 관계를 평가하고 싶다. 예를 들어, `if` 명령문의 조건에 따라 다양한 작업을 수행하고 싶다.

해결책

표 2-4의 논리 연산자를 사용한다.

표 2-4 논리 연산자

기호	기능	설명
&&	논리곱	&& 연산자 양쪽의 조건이 true이면 true로 평가된다.
¦¦	논리합	¦¦ 연산자 양쪽에 있는 조건 중 적어도 한 조건이 true이면 true로 평가된다.
!	부정	표현식이 false면 true로 평가되고, 표현식이 true이면 false로 평가된다.

토론

논리 연산자는 논리 관계에 따라 true 또는 false 값을 리턴한다. 앞으로 살펴볼 예

제들에서는 5장에서 설명한 것처럼 디지털 핀 2와 3에 센서가 연결되어 있는 것으로 가정한다.

논리곱 연산자(&&)는 두 개의 피연산자가 모두 true인 경우에만 true를 리턴하고, 그렇지 않은 경우에는 false를 리턴한다.

```
if( digitalRead(2) && digitalRead(3) )
  blink();  // 두 핀이 모두 HIGH이면 깜박인다.
```

논리합 연산자(||)는 두 개의 피연산자 중 true인 피연산자가 하나라도 있으면 true를 리턴하고, 둘 다 false이면 false를 리턴한다.

```
if( digitalRead(2) || digitalRead(3) )
blink();  // 두 핀 중 하나 이상이 HIGH이면 깜박인다.
```

부정 연산자(!)의 피연산자는 하나뿐이며, 값이 역으로 변경된다. 즉, 피연산자가 true이면 false를 리턴하고, false이면 true를 리턴한다.

```
if( !digitalRead(2) )
blink();  // 핀이 HIGH가 아니면 깜박인다.
```

2.20 비트 연산 수행하기

과제

값을 구성하고 있는 특정 비트를 설정하거나 지우고 싶다.

해결책

표 2-5의 비트 연산자를 사용한다.

표 2-5 비트 연산자

기호	기능	결과	예
&	비트곱	두 비트가 모두 1이면 해당 위치의 비트가 1로 설정되며, 그렇지 않으면 0으로 설정된다.	3 & 1은 1이다. (11 & 01은 01이다.)
|	비트합	두 비트 중 하나라도 1이면 해당 위치의 비트가 1로 설정된다.	3 | 1은 3이다. (11 | 01은 11이다.)

표 2-5 (계속)

기호	기능	결과	예
^	배타적 비트합	두 비트 중 단 하나의 비트가 1인 경우에만 해당 위치의 비트가 1로 설정된다.	3 | 1은 2다. (11 ^ 01은 10이다.)
~	비트 부정	각 비트의 값을 반전시킨다. 데이터 유형의 비트 수에 따라 결과가 달라진다.	~1은 254다. (~00000001은 11111110이다.)

아래 스케치에서는 표 2-5의 예제 값을 보여 준다.

```
/*
 * bits 스케치
 * 비트 연산자의 사용법을 보여 준다.
 */

void setup() {
  Serial.begin(9600);
}

void loop(){
  Serial.print("3 & 1 equals ");  // 3과 1의 비트곱
  Serial.print(3 & 1);  // 결과를 인쇄한다.
  Serial.print(" decimal, or in binary: ");
  Serial.println(3 & 1, BIN);  // 결과의 2진 표현을 인쇄한다.

  Serial.print("3 | 1 equals ");  // 3과 1의 비트합
  Serial.print(3 | 1 );
  Serial.print(" decimal, or in binary: ");
  Serial.println(3 | 1, BIN);  // 결과의 2진 표현을 인쇄한다.

  Serial.print("3 ^ 1 equals ");  // 3과 1의 배타적 비트합
  Serial.print(3 ^ 1);
  Serial.print(" decimal, or in binary: ");
  Serial.println(3 ^ 1, BIN);  // 결과의 2진 표현을 인쇄한다.


  byte byteVal = 1;
  int intVal = 1;

  byteVal = ~byteVal;  // 비트 부정을 수행한다.
  intVal = ~intVal;
```

```
  Serial.print("~byteVal (1) equals ");  // 8비트 값에 대한 비트 부정
  Serial.println(byteVal, BIN);  // 결과의 2진 표현을 인쇄한다.
  Serial.print("~intVal (1) equals ");  // 16비트 값에 대한 비트 부정
  Serial.println(intVal, BIN);  // 결과의 2진 표현을 인쇄한다.

  delay(10000);
}
```

다음과 같은 결과가 시리얼 모니터에 표시된다.

```
3 & 1 equals 1 decimal, or in binary: 1
3 | 1 equals 3 decimal, or in binary: 11
3 ^ 1 equals 2 decimal, or in binary: 10
~byteVal (1) equals 11111110
~intVal (1) equals 11111111111111111111111111111110
```

토론

비트 연산자는 비트를 설정하거나 테스트할 때 사용된다. 두 값에 대해 "And" 또는 "Or" 연산을 수행할 경우 연산자는 각각의 개별 비트에 대해 작동한다. 이는 값의 2진 표현을 살펴보면 쉽게 이해할 수 있다.

10진수 3은 2진수로 00000011이며, 10진수 1은 00000001이다. 이 두 수에 대해 비트곱 연산을 수행한다. 두 수 모두 가장 오른쪽 비트가 1이므로 비트곱 연산의 결과도 1이다. 왼쪽으로 옮겨서 다음 비트는 1과 0이므로 비트곱 연산의 결과는 0이 된다. 나머지 비트는 모두 0이므로 이들에 대한 비트곱 연산의 결과는 0이 된다. 다시 말해서 양쪽 모두 1인 비트 위치에서만 결과가 1이 되고, 그렇지 않은 경우에는 결과가 0이 된다. 그러므로 `11 & 01`은 1이다.

표 2-5, 2-6 및 2-7을 보면 비트곱, 비트합 및 배타적 비트합 값을 좀 더 쉽게 이해할 수 있을 것이다.

표 2-6 비트곱

비트 1	비트 2	비트곱 연산 결과
0	0	0
0	1	0
1	0	0
1	1	1

표 2-7 비트합

비트 1	비트 2	비트합 연산 결과
0	0	0
0	1	1
1	0	1
1	1	1

표 2-8 배타적 비트합

비트 1	비트 2	배타적 비트합 연산 결과
0	0	0
0	1	1
1	0	1
1	1	0

부정 연산자를 제외한 모든 비트 표현식은 두 개의 값에 대해 작동한다. 부정 연산자는 각 비트의 값을 다른 값으로 바꿔 준다. 즉, 0은 1이 되고, 1은 0이 된다. 예를 들어, 바이트 값(8비트) 00000001은 11111110이 되며, 정수 값(16비트) 0000000000000001은 1111111111111110이 된다.

참고

비트곱, 비트합 및 배타적 비트합 연산자에 대한 아두이노 레퍼런스: http://www.arduino.cc/en/Reference/Bitwise

2.21 연산과 할당 결합하기

과제

복합 연산자를 이해하고 사용하고 싶다. 공개된 코드 중에서는 하나의 명령문에서 두 가지 이상의 작업을 수행하는 표현식이 포함된 코드를 보기가 어렵다. `a += b`, `a >>= b`, `a &= b` 등에 대해 알고 싶다.

해결책

표 2-9에서는 복합 할당 연산자와 그에 상응하는 전체 표현식을 보여 준다.

표 2-9 복합 연산자

연산자	예	상응하는 표현식
+=	`value += 5;`	`value = value + 5;  // value에 5를 더한다.`
-=	`value -= 4;`	`value = value - 4;  // value에서 4를 뺀다.`
*=	`value *= 3;`	`value = value * 3;  // value에 3을 곱한다.`
/=	`value /= 2;`	`value = value / 2;  // value를 2로 나눈다.`
>>=	`value >>= 2;`	`value = value >> 2;  // value를 오른쪽으로 2비트 이동한다.`
<<=	`value <<= 2;`	`value = value << 2;  // value를 왼쪽으로 2비트 이동한다.`
&=	`mask &= 2;`	`mask = mask & 2;  // mask와 2에 대해 비트곱 연산을 수행한다.`
¦=	`mask ¦= 2;`	`mask = mask & 2;  // mask와 2에 대해 비트합 연산을 수행한다.`

토론

이러한 복합 명령문은 상응하는 전체 표현식에 비해 런타임 효율이 낮다. 게다가 프로그래밍 입문자라면 전체 표현식을 사용하는 것이 이해하기도 쉽고 효과적이다. 숙련된 프로그래머의 경우 약식을 좋아하기도 하므로 복합 연산자를 알고 있으면 나중에 우연히 보게 될 때 코드를 이해하는 데 도움이 될 것이다.

참고

복합 연산자에 대한 레퍼런스 페이지의 인덱스를 보려면 http://www.arduino.cc/en/Reference/HomePage를 참조한다.