PROGRAMOVACÍ JAZYK WIRING

REFERENČNÍ PŘÍRUČKA PROGRAMOVACÍHO JAZYKA WIRING, KTERÝ PŘEDSTAVUJE ZÁKLADNÍ RÁMEC PRO PROGRAMOVÁNÍ MIKROKONTROLÉRŮ BEZ SPECIFICKÝCH ZNALOSTÍ HARDWARU.

Programovací jazyk pro Arduino je odvozen z programovacího jazyka Wiring a je založen na klasickém programovacím jazyku C++, jehož základy položil Bjarne Stroustrup v roce 1983. Jedná se o relativně jednoduchý jazyk, jehož funkce i operátory (např. typu +, -, *, /) jsou intuitivní a snadno pochopitelné. Pro navrhování aplikací pro Arduino byl jazyk navíc dále zjednodušen vynecháním relativně komplikované syntaxe objektového programování.

STRUKTURA PROGRAMU

Základní struktura programovacího jazyka Arduino je poměrně jednoduchá a skládá se nejméně ze dvou částí, přesněji funkcí. Bloky příkazů v těle těchto dvou funkcí jsou ohraničeny složenými závorkami.

```
void setup()
{
  příkazy;
}

void loop()
{
  příkazy;
}
```

Funkce `setup()` je přípravná a provádí se jen jednou na začátku programu, funkce `loop()` je výkonná a provádí se neustále dokola. Pro správnou činnost programu je vždy nutné použít obě tyto funkce.

Funkce `setup()` by měla být volána až po deklaraci všech proměnných na začátku programu. Tato funkce se používá například k nastavení pinů Arduina na vstup nebo výstup, nastavení parametrů sériové komunikace a podobných jednorázových akcí.

Po funkci `setup()` následuje funkce `loop()`. Tělo této funkce obsahuje programový kód, který bude opakovaně prováděn v nekonečné smyčce, například čtení vstupů, nastavování

výstupů, výpočty atd. Tato funkce je jádrem všech programů Arduina a vykonává většinu činností.

SETUP()

Funkce `setup()`, jak již bylo řečeno výše, se volá pouze jednou při spuštění programu. Používá se k inicializaci režimu jednotlivých pinů, k nastavení sériové komunikace apod. Tato funkce musí být v programu obsažena vždy, i když žádné inicializační příkazy neobsahuje.

```
void setup()
{
  pinMode(pin, OUTPUT); // nastav 'pin' na výstup
}
```

LOOP()

Po dokončení funkce `setup()` se začne neustále dokola provádět funkce `loop()`, jak její název (loop =smyčka) ostatně napovídá. Příkazy, obsažené v těle této funkce, jsou určeny k provádění veškeré činnosti Arduina.

```
void loop()
{
  digitalWrite(pin, HIGH);     // nastav 'pin' na log. 1
  delay(1000);                 // čekej jednu sekundu
  digitalWrite(pin, LOW);      // nastav 'pin' na log. 0
  delay(1000);                 // čekej jednu sekundu
}
```

SYNTAXE

SLOŽENÉ ZÁVORKY {}

Složené závorky definují začátek a konec bloku kódu. Používají se ve funkcích i ve smyčkách.

```
type function()
{
  příkazy;
}
```

Za úvodní složenou závorkou [**{**] musí vždy následovat závorka uzavírací [**}**]. Proto se často uvádí, že složené závorky musí být párovány. Samostatně umístěné závorky (úvodní bez uzavírací a naopak) mohou často vést k záhadným, špatně dohledatelným chybám kompilátoru.

Programové prostředí Arduino obsahuje praktickou funkci pro kontrolu párování složených závorek. Stačí vybrat závorku nebo kliknout myší bezprostředně pod závorku a související závorka bude zvýrazněna.

; STŘEDNÍK

Středníkem musí být ukončena deklarace i jednotlivé prvky programu. Středník je také používán k oddělení prvků ve smyčce.

```
int x = 13; // deklaruje proměnnou 'x' jako datový typ integer s
            // hodnotou 13
```

Pokud zapomenete řádek programu středníkem ukončit, dojde k chybě kompilátoru. Z chybového hlášení může být zřejmé, že se jedná o zapomenutý středník, ovšem také nemusí. Pokud se objeví hlášení o záhadné nebo zdánlivě nelogické chybě kompilátoru, zkontrolujte nejprve, zda v zápisu programu nechybí středník v blízkosti místa, kde kompilátor chybu ohlásil.

/*… */ BLOKOVÉ KOMENTÁŘE

Blokové komentáře nebo víceřádkové komentáře jsou oblasti textu, které jsou programem ignorovány. Jsou používány pro obsažnější komentování kódu nebo poznámky, které pomohou pochopit ostatním význam částí programu. Blokové komentáře začínají / * a končí * / a mohou obsahovat více řádků textu.

```
/*
Toto je blokový komentář, nezapomeňte ho ukončit.
Znaky pro jeho začátek a konec musí být vždy v páru!
*/
```

Vzhledem k tomu, že komentáře jsou programem ignorovány, nezabírají žádný paměťový prostor; mohou tedy být hojně používány. Mohou být také použity k dočasnému znefunkčnění celých bloků kódu programu pro účely ladění.

Do blokového komentáře je možno vložit i jednořádkové komentáře (uvozené //), ale není možno do blokového komentáře vložit další blokový komentář.

// JEDNOŘÁDKOVÉ KOMENTÁŘE

Jednotlivé řádky komentáře musí začínat // a končí na konci řádku. Stejně jako blokové komentáře jsou jednořádkové komentáře programem ignorovány a nezabírají žádný paměťový prostor.

```
// toto je jednořádkový komentář
```

Jednořádkové komentáře jsou často používány za příkazy k vysvětlení jejich funkce nebo jako poznámka pro další použití.

PROMĚNNÉ

Proměnná je způsob pojmenování a uložení číselné hodnoty pro pozdější použití v programu. Jak jejich název naznačuje, proměnné jsou čísla, jejichž hodnota může být průběžně změněna, na rozdíl od konstant, jejichž hodnota se nikdy nemění. Proměnné musí být deklarovány a volitelně jim lze přiřadit hodnoty, které do nich mají být uloženy.

Následující kód deklaruje proměnnou názvem **inputVariable** a přiřadí jí hodnotu získanou čtením analogové hodnoty vstupního pinu 2:

```
int inputVariable = 0;           // deklaruje proměnnou
                                 // a přiřadí jí hodnotu 0
inputVariable = analogRead(2);   // přiřadí proměnné hodnotu
                                 // podle analogové hodnoty pinu 2
```

Proměnnou pojmenujeme **InputVariable**. První řádek kódu deklaruje, že proměnná bude obsahovat datový typ int (zkratka pro název integer) a nastaví její hodnotu na 0. Druhý řádek nastaví hodnotu proměnné podle hodnoty analogového 2, která je dostupná jinde v programovém kódu.

Jakmile byla proměnné přiřazena nebo změněna hodnota, můžete použít její hodnotu přímo nebo testovat, zda tato hodnota splňuje určité podmínky.

Příklad ukazuje tři užitečné operace s proměnnými. Následující kód testuje, zda hodnota proměnné **inputVariable** je menší než 100, je-li to pravda, pak proměnné **inputVariable** přiřadí hodnotu 100, a pak nastaví na základě hodnoty proměnné **inputVariable** prodlevu, která je nyní minimálně 100:

```
if(inputVariable < 100)
            // testuje, zda je hodnota proměnné méně než 100
{
  inputVariable = 100;
            // je-li to pravda, je jí přiřazena hodnota 100
}
delay(inputVariable);
            // použije hodnotu proměnné jako parametr funkce
            // delay
```

Proměnným bychom měli pro lepší orientaci v programovém kódu dávat popisné názvy. Jména proměnných, jako je **`tiltSensor`** nebo **`pushButton`** pomáhají programátorovi i komukoli jinému při čtení kódu snadněji poznat, co proměnná znamená. V našich příkladech ale naopak používáme krátká jména proměnných (například `var` nebo `value`), aby byl kód kratší a přehlednější. Proměnná může být pojmenována libovolným jménem, které nepatří mezi klíčová slova v jazyce Arduino.

DEKLARACE PROMĚNNÝCH

Všechny proměnné musí být deklarovány ještě před jejich prvním použitím. Deklarace proměnné znamená, že definujete její typ (`int`, `long`, `float`, atd.), nastavíte její jméno, a případně přiřadíte počáteční hodnotu. Tyto deklarace postačí v programu provést jen jednou, hodnotu proměnné ale můžete v programu kdykoliv změnit.

Následující příklad deklaruje, že proměnná **`inputVariable`** je typu `int` neboli `integer`, a že její počáteční hodnota je rovna nule. To se nazývá jednoduché přiřazení.

```
int inputVariable = 0;
```

Proměnná může být deklarována kdekoli v programu a je použitelná od místa deklarace dále.

PLATNOST PROMĚNNÝCH

Proměnná může být deklarována na začátku programu, ještě před funkcí `void setup()`, lokálně uvnitř funkce, a někdy v deklaraci bloku, jako je tomu u smyček. Místo, kde je proměnná deklarována, určuje její použitelnost pro některé části programu.

Globální proměnná je ta, ke které mají přístup a kterou mohou používat všechny funkce a deklarace v programu. Taková proměnná musí být deklarována na začátku programu, ještě před funkcí `void setup()`.

Lokální proměnná je ta, která je definována uvnitř funkce nebo jako součást smyčky. Je dostupná a může být použita pouze uvnitř funkce, uvnitř které byla deklarována. Je tedy

možné mít v různých částech téhož programu umístěné dvě nebo více proměnné stejného jména, které obsahují různé hodnoty. To, že přístup ke svým lokálním proměnným má pouze daná funkce, zjednodušuje program a snižuje možnost programových chyb.

Následující příklad ukazuje, jakým způsobem můžeme deklarovat různé typy proměnných a předvádí viditelnost každé z proměnných v programu:

```
int value;        // proměnná 'value' je viditelná
                  // pro všechny funkce
void setup()      // funkce neobsahuje žádné inicializační příkazy
{

}
void loop()
{
  for(int i=0; i<20;) // proměnná 'i' je viditelná
  {                   // jen uvnitř této smyčky
    i++;
  }
  float f;        // proměnná 'f' je viditelná
}                 // jen uvnitř této smyčky
```

KONSTANTY

Jazyk Arduino má několik předdefinovaných hodnot, které se nazývají konstanty. Ty se používají k tvorbě přehlednějších programů. Konstanty jsou seřazeny do skupin.

TRUE / FALSE

Jedná se o logické konstanty, které definují logické úrovně. FALSE je jednoduše definována jako 0 (nula), TRUE je často definována jako 1, ale může nabývat každou hodnotu kromě 0. V tomto smyslu je výsledek operace Boolean -1, 2 a -200 také definován jako TRUE.

```
if(b == TRUE)
{
  příkazy;
}
```

HIGH / LOW

Tyto konstanty definují logickou úroveň pinů Arduina jako vysokou nebo nízkou a jsou používány při čtení nebo zápisu této hodnoty na digitální piny.

HIGH je definována jako logická úroveň 1, ON, nebo 5 voltů, zatímco LOW je logická úroveň 0, OFF nebo 0 voltů.

```
digitalWrite(13, HIGH);
```

INPUT / OUTPUT

Konstanty používané ve funkci `pinMode()` pro přepnutí funkce digitálního pinu na vstup (INPUT) nebo výstup (OUTPUT).

```
pinMode(13, OUTPUT);
```

DATOVÉ TYPY

BYTE

Datový typ byte ukládá hodnoty jako 8-bitové číselné hodnoty bez desetinných míst. Ukládá hodnotu v rozsahu 0-255.

```
byte someVariable = 180;      // deklaruje proměnnou 'someVariable'
                              // jako datový typ byte
```

INT

Celočíselný datový typ `integer` je nejběžnějším způsobem ukládání celých čísel. Ukládá hodnotu jako 16-bitovou v rozsahu -32 768 až 32 767.

```
int someVariable = 1500;      // deklaruje proměnnou 'someVariable'
                              // jako datový typ integer
```

Celočíselná proměnná přeteče (podteče), pokud dojde k překročení její maximální (minimální) hodnoty. Například, je-li x = 32 767 a následná operace zvětší hodnotu o 1, (například x = x + 1 nebo x ++), hodnota x je po přetečení rovna -32 768.

LONG

Datový typ `long` (integer) je určen pro velká čísla bez desetinných míst. Ukládá hodnotu jako 32-bitovou v rozsahu -2 147 483 648 až 2 147 483 647.

```
long someVariable = 90000;    // deklaruje proměnnou 'someVariable'
                              // jako datový typ long
```

FLOAT

Datový typ `float` je určen pro operace s čísly s plovoucí desetinnou čárkou. Čísla qs plovoucí desetinnou čárkou mají větší rozlišení než celá čísla a jsou uložena jako 32-bitové hodnoty s rozsahem -3,4028235E38 k 3,4028235E38.

```
float someVariable = 90000;   // deklaruje proměnnou 'someVariable'
                              // jako datový typ float
```

Čísla s plovoucí desetinnou čárkou mají omezenou přesnost, díky níž může dojít k problémům při porovnávání dvou takových čísel. Matematické operace s plovoucí desetinnou čárkou jsou také mnohem pomalejší než celočíselná aritmetika a je vhodné – pokud je to možné – se jim vyhýbat.

POLE

Polem se nazývá množina prvků, ke kterým je možno přistupovat prostřednictvím indexace. Libovolná hodnota v poli může být zpřístupněna uvedením názvu pole a indexu hodnoty. Pole jsou indexována od nuly, přičemž první hodnota v poli má index 0. Pole musí být deklarována a je jim možno přiřadit hodnoty dříve, než je můžeme použít.

```
int myArray[ ] = {hodnota0, hodnota1, hodnota2. ..}
```

Stejně tak je možné deklarovat pole tak, že mu určíme datový typ a velikost a později mu přiřadíme hodnoty na pozice dané indexem:

```
int myArray[5];  // deklaruje pole typu integer se šesti prvky
myArray[3] = 10; // přiřadí čtvrtému prvku pole hodnotu 10
```

Chceme-li načíst do proměnné hodnotu z pole, přiřadíme proměnné název pole a index pozice:

```
x=myArray[3]; // 'x' se nyní rovná 10
```

Pole jsou často používána v cyklech **for**, kde je přírůstek čítače je zároveň použit jako index pozice pro každou hodnotu pole.

V následujícím příkladu definujeme pole pro blikání LED. Použijeme smyčku, jejíž čítač začíná na 0. Hodnota indexu na pozici 0 z pole **flicker[]** se zapíše do proměnné **ledPin** (v tomto případě 180). Na pinu 10, který je nastaven do analogového módu se objeví PWM modulace s parametry, určenými proměnnou **ledPin**, běh programu se pozastaví na 200 ms. Pak se smyčka znovu opakuje, vybere se další prvek pole podle hodnoty proměnné čítače atd.

```
int ledPin = 10;        // LED na pinu 10
byte flicker[] = {180, 30, 255, 200, 10, 90, 150, 60} ;
                  // vytvořeno pole s osmi prvky různých hodnot
void setup()
{
  PinMode(ledPin, OUTPUT);    // nastav pin jako VÝSTUP
}

void loop()
{
  for(int i=0; i<8; i++)      // smyčka pokračuje, dokud se
                              // proměnná 'i' nerovná počtu prvků
                              // v poli
  {
    analogWrite(ledPin, flicker [i]) ;
                             // zapiš do proměnné hodnotu indexu
                              // z pole
    delay(200) ;              // pauza 200 ms
  }
}
```

ARITMETICKÉ OPERACE

Aritmetické operátory jsou: sčítání, odčítání, násobení a dělení.

Vrací součet, rozdíl, součin nebo podíl (v uvedeném pořadí) dvou operandů.

```
y = y + 3;
x = x - 7;
i = j * 6;
r = r / 5;
```

Výsledek aritmetické operace je stejného datového typu jako operandy. Například 9/4 poskytne výsledek 2 namísto 2.25, protože operandy 9 a 4 jsou typu int, který není schopen vyjádřit číslice za desetinnou čárkou. To také znamená, že u aritmetické operace může dojít k přetečení, pokud výsledek je větší než maximum, které může být uloženo v použitém datovém typu.

Jsou-li operandy různých typů, je pro výpočet použit vždy větší typ. Například, jestliže jeden z operandů je typu `float` a druhý typu `int`, bude výpočet probíhat v typu `float`.

Pro své proměnné zvolte vždy takový datový typ, který dokáže uložit dostatečně velká čísla jako výsledky vašich výpočtů. Mějte na paměti, v kterém okamžiku vaše proměnná přeteče a také to, co se stane při přechodu v opačném směru, např. (0 -1) nebo (0 -32768). Pro matematiku, která vyžaduje zlomky, zvolte proměnné typu `float`, ale buďte si vědomi jejich nedostatků: zabírají více místa v paměti a výpočet je pomalejší.

Pomocí operátoru přetypování např. (`int`) **`myFloat`** lze převést jeden typ proměnné na jiný aniž by bylo nutné ukládat přetypovanou hodnotu do proměnné. Například, **`i = (int) 3.6`** nastaví i rovno 3.

SLOŽENÉ PŘIŘAZENÍ

Složené přiřazení kombinuje aritmetické operace s přiřazením hodnoty proměnné. Tyto operace se běžně vyskytují v cyklech **`for`**, jak je popsáno dále.

Mezi nejčastější složená přiřazení patří:

```
x++ // je totéž jako x=x+1, tedy zvětšení hodnoty x o +1
x-- // je totéž jako x=x-1, tedy zmenšení hodnoty x o -1
x+=y // je totéž jako x=x+y, tedy zvětšení hodnoty x o hodnotu +y
x-=y // je totéž jako x=x-y, tedy zmenšení hodnoty x o hodnotu - y
x*=y // je totéž jako x=x*y, tedy vynásobení x y
x/=y // je totéž jako x=x/y, tedy podělení x y
```

Například operace **x *= 3** ztrojnásobí původní hodnotu x a znovu jí přiřadí výsledné hodnotě **x**.

RELAČNÍ OPERÁTORY

Porovnání jedné proměnné nebo konstanty s jinou se často používá v testech uvnitř příkazu `if`, kdy se porovnává, zda zadaná podmínka platí. Příklady naleznete na následujících stránkách.

Znak **??** zastupuje v dalším textu některou z následujících podmínek:

```
x==y // x je rovno y
x!=y // x není rovno y
x<y  // x je menší než y
x>y  // x je větší než y
x<=y // x je menší nebo rovno y
x>=y // x je větší nebo rovno y
```

LOGICKÉ OPERÁTORY

Logické operátory jsou obvyklým způsobem porovnávání dvou výrazů. Tyto operátory vrací hodnotu TRUE nebo FALSE v závislosti na typu operace.

K dispozici jsou tři logické operátory AND, OR a NOT, které jsou často používané v testu `if`:

LOGICKÉ AND:

```
if (x>0 && x<5)   // Vrací hodnotu TRUE, když oba
                  // výrazy mají hodnotu TRUE
```

LOGICKÉ OR:

```
if (x>0 || y>0)   // Vrací hodnotu TRUE, když kterýkoli
                  // z prvků má hodnotu TRUE
```

LOGICKÉ NOT:

```
if (!x)           // má za následek true jestliže operand je
                  // false a naopak
```

ŘÍZENÍ TOKU PROGRAMU

IF

Příkaz `if` testuje, zda bylo dosaženo určité podmínky, třeba zda analogová hodnota je větší než zadaná, a provádí všechny příkazy uvnitř závorek, pokud tvrzení je pravdivé (TRUE).

Pokud tvrzení pravdivé není (FALSE), program příkazy uvnitř závorek přeskočí.

Formát příkazu `if` je:

```
if(someVariable ?? value)
{
  příkazy;
}
```

Výše uvedený příklad porovnává hodnotu proměnné **`someVariable`** s jinou hodnotou, kterou může být opět buď proměnná nebo konstanta. Pokud je výsledek porovnání hodnot v závorce pravda (TRUE), jsou vykonány příkazy uvnitř složených závorek. Pokud ne, program je přeskočí a pokračuje za nimi.

Dejte si pozor na náhodné použití „=“ místo „==“ v příkazu `if(x = 10)`, zápis je syntakticky správný, ale nastavuje hodnotu proměnné x na hodnotu 10 a výsledkem je tedy vždy pro nenulovou hodnotu pravda (TRUE). Místo „=“ je nutno použít výraz „==“, tedy `(x == 10),` který jen testuje, zda x se rovná hodnotě 10, nebo ne. Myslete na „=“ jako na "rovná se" na rozdíl od „==“, které znamená „se

IF .. ELSE

Operace `if..else` umožňuje rozhodování stylem „buď – nebo“ a větvení programu podle výsledku operace. Například, pokud chcete otestovat stav digitálního vstupu a pak provést nějakou činnost, pokud je vstupní pin ve stavu HIGH nebo naopak provést něco jiného, pokud je vstupní úroveň nízká, můžete to zapsat tímto způsobem:

```
if(inputPin == HIGH)
{
  doThingA;
}
else
{
  doThingB;
}
```

nebo může také předcházet jiné `if`, pokud testujeme více vzájemně se vylučujících podmínek. Testy lze spustit současně. Je dokonce možné mít neomezený počet těchto větvení `else if`. Pamatujte si však, že je možné vždy spustit v závislosti na podmínkách testů jen jednu část kódu:

```
if(inputPin < 500)
{
  doThingA;
}
else if(inputPin >= 1000)
{
  doThingB;
}
else
{
  doThingC;
}
```

Příkaz if pouze testuje, zda je podmínka v závorce pravdivá nebo nepravdivá. Touto podmínkou může být jakýkoli platný příkaz jazyka C jako například, `if (inputPin == HIGH)`. V tomto příkladu 'if' pouze zkontroluje, zda opravdu má zadaný vstup logickou úroveň HIGH neboli 5 V.

FOR

Příkaz `for` se používá k několikanásobnému opakování bloku příkazů, uzavřených do složených závorek. Záhlaví smyčky se skládá ze tří částí, oddělených středníkem (;):

```
for(inicializace; podmínka; výraz)
{
  příkazy;
}
```

Nejprve je deklarována lokální proměnná **`inicializace`** ve funkci přírůstkového čítače. Tato proměnná se deklaruje jen jednou. Za deklarací proměnné následuje podmínka, která se testuje při každém průchodu smyčkou. Dokud je podmínka pravdivá, jsou provedeny příkazy a výrazy uzavřené ve složených závorkách a výraz, která je součástí hlavičky for. Když se podmínka stane nepravdivou, smyčka se ukončí.

Následující příklad testuje hodnotu proměnné **`i`** typu integer, nastavené na 0 a zjišťuje, zda je její hodnota stále menší než 20. Je-li to pravda, provede se tělo cyklu a zvětší se hodnota proměnné **`i`** o 1:

```
for(int i=0; i<20; i++)      // deklaruje proměnnou 'i', testuje
                             // zda je menší než 20 a pokud
                             // ne, zvětší j i o 1
{
  digitalWrite(13, HIGH);    // nastaví pin 13 na HIGH
  delay(250);                // čeká 250 ms
  digitalWrite(13, LOW) ;    // nastaví pin 13 na LOW
  delay(250);                // čeká 250 ms
}
```

Programovací jazyk C má smyčky mnohem flexibilnější, než jsou podobné smyčky v jiných programovacích jazycích, včetně BASICu. Některé z prvků záhlaví nebo i všechny tři mohou být vynechány, středníky ale zůstávají povinné. Také pro inicializaci, podmínku i výraz může být použit jakýkoli platný příkaz jazyka C s nezávislými proměnnými. Takto definované záhlaví příkazu `if` může nabídnout řešení některých neobvyklých programových problémů.

WHILE

Smyčka `while` se bude nekonečně cyklicky opakovat, dokud výraz uvnitř závorek nebude nepravdivý (FALSE). Pokud se hodnota testované proměnné nezmění, program smyčku `while` neopustí. Změnou může být ve vašem programu například inkrementace proměnné nebo změna externí podmínky, jakou může být výsledek testu senzoru.

```
while (someVariable ?? value)
{
  příkazy;
}
```

Následující příklad testuje, zda je hodnota proměnné **`someVariable`** menší než 200 a je-li to pravda, provede příkazy uzavřené mezi v závorkami. Tak se bude tato smyčka opakovat až do doby, kdy proměnná **`someVariable`** nabyde hodnoty 200 nebo větší.

```
while (someVariable < 200) // test, zda je proměnná menší než 200
{
  příkazy;                 // vykonání příkazů
  someVariable++;          // zvětšení proměnné o 1
}
```

DO .. WHILE

Tato smyčka pracuje stejně jako smyčka `while`, jen s tím rozdílem, že podmínka je testována nikoli na začátku, ale až na konci smyčky, takže smyčka proběhne vždy nejméně jednou.

```
do
{
  příkazy;
}
while (someVariable ?? value);
```

Následující příklad přiřadí funkci `readSensors()` hodnotu proměnné **x**, zastaví činnost na 50 milisekund, poté bude smyčku opakovat, dokud nebude **x** menší než 100:

```
do
{
  x = readSensors();    // přiřaď funkci readSensors()
                        // hodnotu proměnné 'x'
  delay(50);            // čekej 50 milisekund
} while (x < 100);      // znovu na začátek dokud 'x'
                        // není menší než 100
```

FUNKCE

Funkce je část kódu, který je pojmenován a který ve svém těle obsahuje blok příkazů, které jsou provedeny při spuštění této funkce. Činnosti funkcí `void setup()` a `void loop()` již byly popsány výše a další vestavěné funkce budou popsány dále.

Vlastní funkce mohou vykonávat opakující se úkony a slouží také k udržení přehlednosti programu. Funkce musí být před použitím nejprve deklarovány čili musí být uvedeno, jakého jsou typu. To je datový typ hodnoty, kterou bude funkce vracet jako výsledek své činnosti, například `int` (zkratka pro integer), pokud funkce vrací celočíselnou hodnotu. Pokud se nemá vracet žádná hodnota, deklarujeme funkci typu `void(prázdná)`. Po uvedení typu následuje název funkce a v závorce seznam parametrů, předávaných funkci.

```
type functionName(parameters)
{
  příkazy;
}
```

Následující funkce `delayVal()` typu integer slouží v programu k nastavení velikosti zpoždění čtením hodnoty napětí z potenciometru. Nejprve deklarujeme lokální proměnnou **v**, vložíme do ní hodnotu přečtenou z potenciometru, která může nabývat hodnoty v rozsahu mezi 0 a 1023, pak se využije funkce `map`, abychom získali číslo v rozsahu mezi 0 a 255, a nakonec vrátíme tuto hodnotu zpět do hlavního programu.

```
int delayVal( )
{
  int v;                        // vytvoř í dočasnou proměnnou 'v'
  v = analogRead(pot);          // přečte hodnotu potenciometru
  v = map(v, 0, 1023, 0, 255); // konvertuje číslo z rozsahu 0 až
                               // 1023 ma 0 - 255
  return v;                     // vrátí finální hodnotu 'v'
}
```

DIGITÁLNÍ VSTUPY A VÝSTUPY

PINMODE(PIN, MODE)

Tato konstanta se používá ve funkci `void setup()` a nastavuje určitý pin na vstup (INPUT) nebo výstup (OUTPUT).

```
pinMode(pin, OUTPUT); // nastav 'pin' jako výstupní
```

Digitální piny Arduina jsou standardně nastaveny jako vstupní, takže je není nutno do tohoto stavu znovu nastavovat pomocí funkce `pinMode()`. Piny nakonfigurované jako vstupní jsou ve stavu vysoké impedance.

Na piny ale mohou být připojeny zdvihací (pull-up) rezistory o velikosti 20 až 50 kΩ, které jsou obsaženy v interní struktuře mikrokontroléru. Ve výchozím nastavení mikrokontroléru jsou tyto zdvihací rezistory odpojeny, ale je možno je programově připojit.

Provádí se to následujícím způsobem:

```
pinMode(pin, INPUT);           // nastav 'pin' jako vstupní
digitalWrite(pin, HIGH);       // zapni pul l-up rezistory
```

Všimněte si, že ve výše uvedeném příkladu se nepřepíná pin na výstup, ale je to jen způsob aktivace vnitřního pull-up rezistoru. Pull-up rezistory se běžně používají, pokud na vstup připojujeme například spínač.

Piny, nakonfigurované jako výstup, mohou poskytnout proud až 40 mA do jiných zařízení či obvodů. Takový proud postačí pro jasné rozsvícení LED (nezapomeňte na předřadný rezistor), ale není to dostatečný proud pro sepnutí většiny relé, solenoidů nebo motorů.

Zkraty na pinech Arduina nebo jejich zatížení nadměrným proudem může obvod výstupního pinu poškodit nebo zničit, případně poškodit celý čip ATmega.

Proto je vhodné při připojování výstupního pinu k externímu zařízení použít rezistor s odporem 470 Ω nebo 1 kΩ, zapojeným v sérii s připojovaným zařízením.

DIGITALREAD(PIN)

Funkce `digitalRead(pin)` přečte hodnoty z určeného digitálního pinu. Výsledkem je vysoká nebo nízká úroveň. Pin může být zadán buď jako proměnná nebo jako konstanta (0-13).

```
value=digitalRead(pin);   // nastav proměnnou 'value' do stejného
                          // stavu, jako má vstupní pin
```

DIGITALWRITE(PIN, VALUE)

Nastaví zadaný digitální pin na vysokou (High) nebo nízkou (Low) úroveň. Číslo pinu může být zadáno buď jako proměnná nebo jako konstanta (0-13).

```
digitalWrite(pin, HIGH); // nastav 'pin' na HIGH
```

Následující příklad čte stav tlačítka, připojeného na digitální vstup a pokud je sepnuto, rozsvítí LED, připojenou k digitálnímu výstupu.

```
int led = 13;           // připoj LED na pin 13
int pin = 7;            // připoj tlačítko na pin 7
int value = 0;          // proměnná pro uložení přečtené hodnoty
void setup()
{
  pinMode(led, OUTPUT) ;     // nastav pin 13 jako výstup
  pinMode(pin, INPUT) ;      // nastav pin 7 jako vstup
}
void loop()
{
  value=digitalRead(pin);   // vlož do proměnné 'value' stav
                            // vstupního pinu
  digitalWrite(led, value); // nastav hodnotu proměnné 'led' podle
}                           // stavu tlačítka
```

ANALOGOVÉ VSTUPY A VÝSTUPY

ANALOGREAD(PIN)

Přečte hodnotu napětí z určeného analogového pinu v 10-bitovém rozlišením. Tato funkce pracuje pouze s analogovými piny (0-5). Výsledná hodnota je celočíselná s rozsahem od 0 do 1023.

```
value = analogRead(pin);      // nastav hodnotu proměnné 'value'
                              // na hodnotu proměnné 'pin'
```

Analogové piny, na rozdíl od digitálních nemusí být nejprve deklarovány jako vstupní nebo výstupní.

ANALOGWRITE(PIN, VALUE)

Zapíše pseudo-analogové hodnoty, generované pomocí hardwarově řízené pulzní šířkové modulace (PWM) na výstupní piny označené jako PWM. Na novějších Arduinech s čipy ATmega168 tato funkce pracuje na pinech 3, 5, 6, 9, 10 a 11. U starších Arduin s čipy ATmega8 jsou k dispozici pouze piny 9, 10 a 11.

Hodnota v rozsahu 0-255 může být zadána jako proměnná nebo konstanta.

```
analogWrite(pin, value); // zapíše 'value' analogově na 'pin'
```

Hodnota 0, zapsaná do **`value`**, nastavuje na zadaném výstupním pinu napětí 0 voltů (log.0), hodnota 255 nastavuje na zadaném výstupním pinu napětí 5 voltů (log.1). Aby bylo možno dosáhnout hodnot napětí mezi 0 a 5 volty, střídá se na výstupním pinu hodnota 0 a 1. Poměr tohoto střídání je určen hodnotou **`value`** (0 až 255) – čím vyšší hodnota, tím déle je na pinu vysoká logická úroveň (5 V). Například, pro hodnotu 64 je na pinu úroveň 0 tři čtvrtiny času, a 1 čtvrtinu času, pro hodnotu 128 bude na pinu úroveň 0 polovinu času a 1 druhou polovinu, a hodnota 192 znamená, že na pinu bude hodnota 0 čtvrtinu času a 1 tři čtvrtiny času.

Protože se jedná o hardwarovou funkci, bude pin po zavolání funkce `analogWrite` na pozadí běhu programu generovat nastavenou PWM až do příštího volání funkce

`analogWrite` (nebo do zavolání funkce `digitalRead` či `digitalWrite` na stejném pinu).

Analogové piny – na rozdíl od těch digitálních, nemusí být nejprve deklarovány jako vstupní nebo výstupní.

Následující příklad čte analogovou hodnotu ze vstupního pinu, upraví její hodnotu vydělením čtyřmi a pošle signál na výstup PWM:

```
int led = 10;           // LED s rezistorem 220 ohm na pinu 10
int pin = 0;            // potenciometer na analogový pin 0
int value;              // definice proměnné value
void setup()            // žádné nastavení není potřebné
{

}
void loop()
{
  value = analogRead(pin);    // přepiš hodnotu 'value' do 'pin'
  value = map(pin, 0, 1023, 0, 255); // konvertuj rozsah 0-1023 na
                                     //0-255
  analogWrite(led, value);    // vyšli výstupní PWM signal na pin
                              // led
}
```

ČASOVÁNÍ

DELAY(MS)

Pozastaví program na dobu určenou v milisekundách, hodnota 1000 tedy odpovídá jedné sekundě.

```
delay(1000); // čekej jednu sekundu
```

MILLIS()

Vrací počet milisekund od okamžiku rozběhu aktuálního programu na desce Arduino ve formátu unsigned long.

```
value = millis(); // nastav 'value' shodně s millis()
```

Hodnota této proměnné přeteče (přetočí se zpět na nulu), přibližně po 9 hodinách nepřetržitého běhu Arduina.

MATEMATICKÉ FUNKCE

MIN(X, Y)

Vypočítá poměr dvou čísel jakéhokoli datového typu a vrátí menší z nich.

```
value = min(value, 100);    // nastaví 'value' na menší z hodnot
                            // 'value' nebo 100, čímž zajistí, že
                            // proměnná 'value' nepřekročí 100
```

MAX(X, Y)

Vypočítá poměr dvou čísel jakéhokoli datového typu a vrátí větší z nich.

```
value = max(value, 100);  // nastaví 'value' na větší z hodnot
                          // 'value' nebo 100, čímž zajistí, že
                          // proměnná 'value' neklesne pod 100
```

NÁHODNÁ ČÍSLA

RANDOMSEED(SEED)

Nastavuje hodnotu jako výchozí bod pro funkci `random()`.

```
randomSeed(value); // použij hodnotu proměnné 'value'
```

Arduino není schopno vytvořit skutečně náhodné číslo, funkce randomSeed umožňuje umístit proměnnou, konstantu, nebo jinou funkci do funkce random. Tento postup pomáhá vytvořit náhodnější "náhodná" čísla. Existuje celá řada různých funkcí, které mohou být použity jako základ funkce `random`, včetně funkce `millis()` nebo dokonce funkce `analogRead()`, která může číst elektrický šum přes analogový pin.

Random seed („náhodné semínko“) je náhodné číslo (nebo pole), které se používá při inicializaci generátoru pseudonáhodných čísel. Generátor při použití jiného semínka vrací jinou sekvenci pseudonáhodných dat.

RANDOM(MAX), RANDOM(MIN, MAX)

Funkce `random` vrací pseudo-náhodná čísla v rozsahu stanoveném hodnotami min a max. Pokud je uveden pouze jeden parametr, bere se jako maximální hodnota.

```
value = random(100, 200);     // zapiš do 'value' náhodné
                              // číslo mezi 100-200
```

Nejprve použijte funkci `randomSeed()`.

Následující příklad vytvoří náhodnou hodnotu mezi 0-255 a použije ji pro řízení úrovně PWM signálu pinu PWM:

```
int randNumber;          // proměnná pro uložení náhodného čísla
int led = 10;            // LED s rezistorem 220 ohm na pin 10
void setup()             // nic se nenastavuje
{

}
void loop()
{
  randomSeed(millis()); // použij funkci millis() jako hodnotu
  randNumber = random(255);  // ulož náhodné číslo mezi 0-255 do
                             // proměnné
  analogWrite(led, randNumber); // zapiš hodnotu proměnné na výstup
  delay(500) ;                  // čekej 0,5 sekundy
}
```

KOMUNIKACE

SERIAL.BEGIN(RATE)

Otevře sériový port a nastaví komunikační rychlost pro přenos dat. Typická přenosová rychlost pro komunikaci s počítačem je 9600 bps, ale jsou podporovány i jiné přenosové rychlosti.

```
void setup()
{
  Serial.begin(9600);   // otevři sériový port
}                       // nastav přenosovou rychlost na 9600 bps
```

Při použití sériové komunikace nelze použít piny 0 (RX) a 1 (TX) současně jako digitální.

SERIAL.PRINTLN(DATA)

Vytiskne data na sériový port jako čitelný ASCII text, následovaný znakem zalomení (ASCII 13 nebo '\r') a znakem nového řádku (ASCII 10 nebo '\n'). Tento příkaz má stejný tvar jako `Serial.print()`, ale jednodušeji se s ním pracuje, pokud zobrazujeme data v programu Serial Monitor.

```
Serial.println(analogValue);  // odešli hodnotu
                              // proměnné 'analogValue'
```

Další informace o různých možnostech funkcí Serial.println() a Serial.print() naleznete na webových stránkách Arduino.cc.

Následující jednoduchý příklad čte data z analogového pinu 0 a odesílá tato data jednou za sekundu do počítače.

```
void setup()
{
  Serial.begin(9600);   // nastav komunikační rychlost na 9600bps
}
void loop()
{
  Serial.println(analogRead(0));    // odešli analogovou hodnotu
  delay(1000);                      // čekej 1 sekundu
}
```