MinNo Documentation:

0.0 Contents:

1.0 Getting Started

The Anatomy of A MinNo Script

Every MinNo file can be broken up into four sections:

- Global Values,
- a Setup function,
- a Loop function,
- and user defined functions.

This is very similar to the anatomy of a normal Arduino sketch. Global values are variables that you want to reference through out the sketch, the setup function runs once at boot, after the setup function the loop function will run until the board is reset or powered off. Any functions written other than those two will only be called if invoked by some other code.

A Quick Start

"Blink" is the Arduino's "hello world" program. It is a simple sketch that turns a pin on, waits, and than turns it off repeatedly. Here is a blink

sketch in MinNo:

```
#/
blink.minno

The classic 'blink' sketch in MinNo.
/#

let outputPin : int = 13; //set pin for reference

def setup none -> none {
    //set mode to output
    pinMode outputPin OUTPUT;
}

def loop none -> none {
    // blink on and off with 1 sec intervals
    digitalWrite outputPin HIGH;
    delay 1000;
    digitalWrite outputPin LOW;
    delay 1000;
}
```

The first block of text is a multi-line. MinNo uses "#/" and "/#" to delimit these comments. A single line comment is started with a "//" similar to C. The next line:

```
let outputPin : int = 13;
```

It an example of one of the kinds of variable declarations in MinNo. "let" is like Javascript's "var" stating that you are about to reference a variable that will need to be allocated. "outputPin" is used as the values ID and it's type is stated right after the colon separating it from the ID. This "colon type"

marker is non optional. The rest of the line is fairly straight forward, noting that the literal value associated with this ID is the number 13. A semi-colon acts as a statement delimiter, similar to C.

After declaring the value, there are two function definitions. These are the required Setup and Loop functions in every MinNo script. The keyword "def" denotes that the following ID will be used for a function definition. After the ID is the type signature (arguments and return type). The Setup and Loop functions both take no arguments and return nothing, so "none -> none" says "I take nothing and I given nothing." As mentioned earlier, the Setup function runs once and the Loop functions runs as long as the board is powered after that. In Setup the only line in it's definition (delimited by "{" and "}") calls the function "pinMode" to set up a pin for output. It is passed "outputPin" (the value we defined) and "OUTPUT" (a builtin value common in Arduino programming). MinNo function calls consume all whitespace-separated IDs and literals after the function name until a colon is found. No parenthesis needed. Section 2.3.0 goes over how to nest function calls within the same line (as a preview, you wrap inner function calls in parenthesis). The Loop function calls two functions twice: "digitalWrite" and "delay". "digitalWrite" sets an output value to a pin ("outputPin" in this case). The value is set by the second arguement. This is case it flips between "HIGH" and "LOW", other builtin values comparable to "on" and "off". In between each call to "digitalWrite" there is a delay followed by "1000". This will cause the Arduino board to stop everything for 1000 milliseconds (1 seccond) and then continue. So this script turns pin 13 on, waits a second, turns it off, and than waits a second before turning it back on again (and again and again).

That's it. That's a basic MinNo script.

2.0 The Language

This is a more formal and thorough walk through MinNo's builtin keywords

and structures.

2.0.1 Types and Literals

MinNo's types closely match C's. Here are the builtin types in MinNo:

- int : 1, 2, 3, -56 and so on. This is for negative or positive whole numbers.

  let exampleInt : int = 5 ;

  let exampleInt2 : int = -535 * 10 ;

- float : and decimal number, positive or negative.

  let exampleFloat: float = 3.14 ;

- char : this any single character wrapped in double quotes.

  let exampleChar : char = "a" ;

- array : a collection of any of the other types. This subtype is denoted by it's keyword after the array keyword wrapped in square brackets. This is also how strings are made (as arrays of chars).

  let exampleIntArray : arrayt[int] = [1,2,3,4,5];

  let exampleString : array[char] = "This is how a string is declared.\n";

  Arrays are index using zero-indexing by referencing the arrays ID with a integer corresponding the the index you want wrapped in square brackets after it.

2.0.2 Operators

The arithmetic operators in MinNo are the standard "+", "-", "*", and "/". They are addition, subtraction, multiplication, and division respectively. There are infix boolean operators as well:

- == : check to see if the numbers on both side are equal, in which case true, otherwise false.
- > : if the number of the right is greater than the left, true, otherwise false.
- < : like ">" but reversed.
- >= : like ">" but will also return true if both numbers are equal.
- <= : like "<" but will also return true if both sides are equal.
- && : a boolean AND. If both sides are true, than it is true, otherwise it is false.
- || : the boolean OR. If either side is true, than the whole thing is true, otherwise it is all false.

The order of these operations are all the same as they are in C. When in doubt, wrap the first thing that you want to resolve in the inner most parenthesis and work out from there.

2.1.0 Declaring Values

One of the most fundamental parts of any programming language is how to declare variables. Given that MinNo targets Harvard Architecture processors (which can handle variables in memory very differently from x86 and ARM processors), variable declaration is actually one of the more nuanced sections of the language. The main difference between declarations in MinNo will be broken down into two sections. Before getting into that here is some general information that is true for both major classes of data.

You have to give a value upon initializing a variable in MinNo. No null types here. This means that some sections of code might result in long lines, but it avoids null pointer errors. You must also give each variable a type. Speaking of which, here are the builtin types in MinNo:

- int : 1, 2, 3, -56 and so on. This is for negative or positive whole numbers.

let exampleInt : int = 5 ;

let exampleInt2 : int = -535 * 10 ;

- float : and decimal number, positive or negative.

  let exampleFloat: float = 3.14 ;

- char : this any single character wrapped in double quotes.

  let exampleChar : char = "a" ;

- array : a collection of any of the other types. This subtype is denoted by it's keyword after the array keyword wrapped in square brackets. This is also how strings are made (as arrays of chars).

  let exampleIntArray : array[int] = [1,2,3,4,5];

  let exampleString : array[char] = "This is how a string is declared.\n";

  Arrays are index using zero-indexing by referencing the arrays ID with a integer corresponding the the index you want wrapped in square brackets after it.

The name of a variable can be any collection letters, underscores, and numbers (as long as they don't start the ID name). Any re occurring name in the script must be the same type and mutability class as the other occurrences regardless of scope. This helps to stop "x" being reused as a throw-away name all over a single sketch.

2.1.1 Immutable Variables

By default all variables are immutable it MinNo. Specifically they act like a variable in C declares with the keyword "const". You cannot reassign a new value after the variable is declared.

Within immutable declarations, there are two classes of data: stack and

program data. Stack immutables are immutable values defined within function definitions. These exact exactly like C's "const" values. The only restriction they entail is that they cannot be reassigned.

The second group of immutable data structures are call "program data". This is to leverage the Harvard memory model. Any immutable value at the global level is stored in PROGMEM instead of flash storage and ram. variables in PROGMEM are quickly accessible and entirely separated from the other storage areas. The details of declaring and accessing these values are handled by MinNo, so you don't need to worry about it if you don't want to.

2.1.2 Mutable Variables

Now sometimes you want mutable variables for iterators or summing variables. MinNo allows the declaration of mutable variables with the "mutable" keyword placed before it's type. For example:

```
 let exampleMutableInt: mutable int = 6;
```

This creates a mutable integer that allows you to reassign it's value like so:

```
 exampleMutableInt = 10;
```

2.2.0 Defining Functions

All function definitions take a similar form in MinNo:

```
 def <some name> <ID-type tuple for arguments or "none"> -
> <type or "none"> {
     <Your code>
 }
```

An example of a simple function definition would be the Setup function

from the quick start section:

```
def setup none -> none {
    //set mode to output
    pinMode outputPin OUTPUT;
}
```

This follows the form specified above. A more complicated example could be this contrived function to multiply two numbers and add 2 to the result

```
def multAndAddTwo x:int, y:int -> int {
    return (x * y) + 2 ;
}
```

Now we have some arguments to parse. After the function ID, there are two arguments separated by a comma (as each argument needs to be). One is labeled "x" and the other as "y". Both are integers.

2.3.0 Using Functions and Values Together

To avoid ambiguity in parsing, when you need to nest functions or operations within each other in MinNo, each sub expression need to be wrapper in parenthesis. Here is an example of a call to the previously defined "multAndAddTwo" being called and passed to another invocation of itself:

```
multAndAddTwo (multAndAddTwo 6 7) 3;
```

This calls the inner most "multAndAddTwo" passing "6" and "7", takes the result and passes it along with "3" to another "multAndAddTwo". The same is true for operations:

```
multAndAddTwo (5 * 9) 2;
```

2.4.0 Iteration

Though the leaning towards immutability might suggest a leaning towards recursive function as well (thusly leaning towards the functional programming camp), Arduino's memory is far too limited to deal with recursion nicely. So MinNo has both "for" and "while" loops. It also supports recursive definitions if you feel like shooting yourself in the foot.

For most iteration, a "for" loop will do. A for loop in MinNo works like any other for loop in C or Javascript. It is made of a few basic parts:

```
  for <declaration or assignment of variable> ;
conditional ; incrementor ;{
      <code>
  }
```

The declaration or assignment of a variable allows the creation or prep of a variable to track the for loops iteration. The conditional, when false, will stop the for loop to continue through your sketch, and the incrementor happens at the end of every pass through the for loop. Here is an example with some filler values that sum subsequent integers:

```
let sum: mutable int = 0;
for let i: mutable int = 1; i <= 10 ; i = i + 1;{
    sum = sum + i;
}
```

For more general iteration, "while" loops should suffice. The form for "while" loops looks like this:

```
while <conditional> {
    <your code>
}
```

The while loop will keep executing your code until the conditional statement is proven false. After that it will continue through your script.

2.5.0 Conditional Branching

Conditional branching in MinNo is handled by the "if" and "else" keywords. The "else" branch is entirely optional. After the if, some boolean expression must follow. A conditional branch has this form:

```
if <boolean expression> {
    <your code if true>
} else {
    <your code if false>
}
```

Your boolean expression can be nested boolean expressions within each other to get fairly complicated behavior. Currently there is no ability to chain "else-if" branches. This should be fixed soon.

3.0 The Builtins

3.1.0 Built In Constants

There are a few pre defined constants similar to Arduino's.

HIGH and LOW

> *HIGH* and *LOW* are constants to act as *on* and *off* value respectively. They are most commonly used as values for *digitalWrite* or used to compare agains the output of *digitalRead*.

INPUT and OUTPUT

> *INPUT* and *OUTPUT* are used exclusively as values to pass to

> *pinMode* what setting up digital pins.

3.2.0 Built In Functions

MinNo's built-in functions act, effectively, as a super set to the Arduino built-ins. Any function you find in Arduino's own documentation will likely be present here. Though in practice only a small subset of those are used. This documentation will go over that subset.

Digital I/O

---

pinMode

> **pinMode** sets a digital pin (pin's 2 through 13) to be in either **INPUT** or **OUTPUT** mode. This should be called in a Setup function at the beginning of a script.
>
> arguments:
>
> - *pin*: An integer representing the pin you are setting.
> - *mode*: Either **OUTPUT** or **INPUT** depending on whether you want to read or write from a pin.

Example:

```
def setup none -> none {
    pinMode 13 OUTPUT; //sets pin 13 to write mode.
    pinMode 9 INPUT; //sets pin 9 to read values values
}
```

---

digitalWrite

> **digitalWrite** outputs a given value (typically **HIGH** or **LOW**) to a

> digital pin.
>
> arguments:
>
> - *pin*: An integer representing the pin you are setting.
> - *value*: Typically **HIGH** or **LOW** (on or off respectively).
>
> Example:

```
def loop none -> none {
    digitalWrite 13 HIGH; //turns pin 13 On
    digitalWrite 13 LOW; //turns pin 13 Off
}
```

---

digitalRead

> **digitalRead** reads and returns either **LOW** or **HIGH** from a specified digital pin.
>
> arguments:
>
> - *pin*: An integer representing the pin you want to read from

Example:

```
 let exampleRead : mutable int = digitalRead 13; //assigns the value read by 13 to exampleRead
```

Analog I/O

---

analogWrite

> **analogWrite** outputs a given value to any pin that is marked as

> analog or for Pulse Width Modulation (PWM). This value can be within a range of 0 to 255.
>
> arguments:
>
> - *pin*: An integer representing the pin you are setting.
> - *value*: Can be any 8-bit value for analog pins or any 8 bit value for digital PWM. Please reference your board to see which is which.

Example:

```
 def loop none -> none {
     analogWrite 1 255; //turns analog pin 1 to full
analog output, typically this is 5 volts.
 }
```

---

analogRead

> **analogRead** reads and returns some value between 0 and 1024 from a specified analog pin.
>
> arguments:
>
> - *pin*: An integer representing the pin you want to read from

Example:

```
 let exampleRead : mutable int = analogRead 4; //assigns
the value read by 4 to exampleRead
```

Time Related Functions

milliseconds

> Returns the amount of milliseconds since the arduino was turned on.

Example:

```
//An example to measure time passed during code execution
let start: int = milliseconds ;
// <your code>
let end: int = milliseconds ;
let totalTime: int = end - start;
```

---

micros

> Like **milliseconds** but it returns microseconds instead.

---

delay

> **delay** halts all operations on the board for a specified number of milliseconds. After that time has passed, the board resumes execution.
>
> arguments:
>
> - *time*: the number of milliseconds you want to pause for.

---

delayMicroseconds

> Like **delay**, but using microseconds instead of milliseconds.

Math Utilities

---

min

> Returns the small of two passed numbers. This can be used to limit the largest number assigned to a variable.
>
> arguments:
>
> - *x*: One of the two values to possibly be returned.
> - *y*: The other value to compare it to.

---

max

> Like *min* but returns the larger of the two.
>
> arguments:
>
> - *x*: One of the two values to possibly be returned.
> - *y*: The other value to compare it to.

---

abs

> Calulates and return the absolute value of a passed integer.
>
> arguments:
>
> - *x*: some integer to calculate the aboslute value of.

---

map

> Takes a given number, it's possible range, and a desired range and constrains that given value from its original range into the desired one. This is useful for restricting the input from **analogRead** to some analog output value (a 10-bit to 8-bit map).

> arguments:
>
> - *valueToMap*: The value you want to constrain.
> - *originalMin*: the lower bounds of the original input.
> - *originalMax*: the upper bounds of the original input.
> - *newMin*: the lower bound of the desired input.
> - *newMax*: the upper bound of the desired input.

Example:

```
 //This function will read an analog pin and automatically
map it to a PWM writable range.
 def readAndMap pin:int -> int {
     let pinread: int = analogRead pin ;
     return map pinread 0 1024 0 255 ;
 }
```

---

pow

> Computes and returns a passed number raise to another passed number.
>
> arguments:
>
> - *value*: the number to raise.
> - *exponent*: the number to raise *value* by.

---

sqrt

> Computes and returns the square root of a passed number.
>
> arguments:
>
> - *x*: the integer to computer a square root for.