#Minik Os for mini(k) spaces

Minik OS is a binary format embedded operating system. Unlike RTOS(es), Minik OS is not a framework. It's a compiled operating system, ready to run on embedded devices.

Minik is coded using C++ and Arduino Framework with the power of ESP32 bindings and FreeRTOS.

Minik is a multi-threaded operating system. Therefore, it can not be assumed as a "Real Time" operating system because of it's internal task switches.

On ESP32, a single mission critical task can be assigned to the second core. Or tasks can be shared between cores.

Minik OS includes an interpreted programming language called CManager. Syntax of CManager can be called a "pseudo-assembler". It is a bit ezoteric. Primarily because it is designed to run on 8bit small chips. Other interpreted languages like micro-python requires a bit more processing power and ram area for compiling.

Minik has internal daemons and tasks running within a semaphore. Daemons run before the tasks.

There are daemons for:

- Serial communications
- Interrupt handlers

`sinan islekdemir`
`sinan@islekdemir.com`

Programming on Minik

Minik works with an internal interpreter called CManager. It's not hard to code CManager as it has a pseudo-asm style syntax. But this syntax is far from your object-oriented programming languages.

Minik runs on many chips that are only 8 bits and has very small ram space. In order to create a language that can be compiled with the least amount of resources, syntax is choosen to be like this. Maybe if I had aimed to run Minik on stronger chips, the syntax could be better.

Each program is split into sub routines. They can be assumed as `void` functions. They get no arguments or return no values. They are just logical "jump" locations.

- Main entrypoint for a program is always the `main:` sub module.
- All variables are defined in the global context of a program.
- Each sub ends with three dash `---` including the last sub.

Common syntax:

```
sub:
COMMAND <argument> <argument> ...
---
```

A very basic Hello world program looks like:

```
main:
SPRINTLN "Hello world"
---
```

Constants

These constants are inherited from Arduino Framework to make the coding a bit more easier.

Their meanings and use cases are the same as Arduino. Please refer to their documentations as they do it better.

If you define a variable with the same name, reading the variable will return you the constant. Not the variable.

```
Constants:

HIGH
LOW
RISING
FALLING
ONLOW  # (ESP32 specific)
ONHIGH  # (ESP32 specific)
INPUT
OUTPUT
INPUT_PULLUP
INPUT_PULLDOWN  # (ESP32 specific)
PI
HALF_PI
TWO_PI
DEG_TO_RAD
RAD_TO_DEG
EULER
```

Serial Port

SPRINTLN - Print given variable, end with new line

```
main:
SPRINTLN "Hello"
SPRINTLN "World"
---
```

output:

```
Hello
World
```

SPRINT - Print given variable but don't end with new line

```
main:
SPRINT "Hello"
SPRINT "World"
---
```

output:

```
HelloWorld
```

SREADLN - Read a full line into given variable

```
main:
SPRINTLN "Hi, what is your name?"
SREADLN name
SPRINT "Welcome "
SPRINTLN name
---
```

output:

```
Hi, what is your name?
Sinan
Welcome Sinan
```

Basic Navigations

GOTO GOTO is your good old GOTO. GOTO moves execution to the given instruction within the same sub. It does not go outside the current sub. Comments are ignored by the interpreter.

Syntax:

```
GOTO instruction_index
```

Example:

```
main:
SPRINTLN "I print once"
```

#This is a dummy comment SPRINTLN "I loop" GOTO 2 —

CALL Call moves the execution to the top of the given sub. When the subs' execution ends, cursor returns back to where it was.

Example:

```
print_serial:
SPRINTLN my_string
---
main:
SET my_string "Hello world"
CALL print_serial
SET my_string "Hello again"
CALL print_serial
---
```

Variables

There are two types of variables in Minik. Numbers and Strings. Numbers are double type and strings are char arrays under the hood.

Constants Some constants are inherited from the Arduino Framework as:

```
HIGH
LOW
RISING
FALLING
ONLOW
ONHIGH
PI
HALF_PI
TWO_PI
DEG_TO_RAD
RAD_TO_DEG
EULER
```

And there are some additional constants that lets your reach some internal functions as:

```
MILLIS -> Return millis()
```

SET Defines a variable with a value.

NOTE It is not mandatory to create variables with SET. Also, some commands like mathematical operators are capable of creating variables when needed.

```
main:
SET my_number 1
SET my_string "sinan@islekdemir.com"
---
```

CPY Copy a variable within given positions.

Syntax:

```
CPY <destination> <from_index> <size> <source>
```

Example:

```
SET my_string "hello world"
CPY new_string 2 3 my_string
```

#new_string = llo

DEL Delete the given variable from memory.

Syntax:

```
DEL <variable_name>
```

Mathematical Operators

Mathematical Operators only accept 3 arguments. 1 for result and 2 for the operations. Minik does not support regular math symbols due to higher complexity requirements in the tokenizer.

```
ADD <result> <number1> <number2>
SUB <result> <number1> <number2>
DIV <result> <number1> <number2>
MUL <result> <number1> <number2>
XOR <result> <number1> <number2>
OR <result> <number1> <number2>
AND <result> <number1> <number2>
POW  <result> <number1> <number2>
```

Logic Operators

Logic operators enable you to navigate / jump to addresses in your code.

CMP Compare two variables. Result is saved in a stack for jump operators. This should be familiar for those who have some Assembler experience.

```
CMP var1 var2
```

JE / JNE / JG / JGE / JL / JLE

```
JE: Jump if equals
JNE: Jump if not equals
JG: Jump if var1 > var2
JGE: Jump if var1 >= var2
JL: Jump if var1 < var2
JLE: Jump if var1 <= var2
```

Syntax:

```
JE label
```

Example:

```
main:
SET number 1
ADD number number 1
CMP number 10
JE exit
GOTO 1
---
exit:
HALT
---
```

Bit Operators

Given a byte with 8 bits:

```
00010011 = 19

LSHIFT x 2 = 01001100 = 76
RSHIFT x 2 = 00000100 =  4
LROTATE x 2 = 01001100 = 76
RROTATE x 2 = 11000100 = 196
```

There are two types of bit operations in Minik

Rotations - Circular shift. A bit that falls of at one end are put back to the other end.

Shift - Simple bit shifts.

LROTATE - Left Rotate bits

```
SET byte 16
LROTATE byte 2
```

#byte is now 64

RROTATE - Right Rotate bits

```
SET byte 64
RROTATE byte 2
```

LSHIFT - Left shift bits

```
SET byte 64
LSHIFT byte 3
```

RSHIFT - Right Shift bits

```
SET byte 64
RSHIFT byte 3
```

System Statements

HALT Stop execution of the program. No arguments.

```
main:
HALT
---
```

SLEEP Sleep given amount of millis.

NOTE: This is not atomic. Sleep duration will not be less then given amount of milli seconds. But based on other tasks in the cycle, this might sleep a few milli seconds more than expected.

Electronics

INT - Attach Interrupt Unlike the external hardware interrupts, this is a soft interrupt. Works with the same logic but if the interrupt lasts for a very short amount of time, there might be a small risk of interrupt not getting caught.

On the other hand, all INPUT pins can be used for this interrupt.

This sets the given pin mode to INPUT.

Why INPUT instead of INPUT_PULLUP?

Not all boards or pins can have their built-in pull-up resistors. A 10k resistor should do the trick. If you are not familiar with the pull-up or pull-down resistors, check out this random url that I have found:

https://roboticsbackend.com/arduino-input_pullup-pinmode/

```
RISING = 1
FALLING = 2
CHANGE = 3
ONLOW = 4
ONHIGH = 5
```

Syntax:

```
main:
INT 3 5 pin_3_high
INT 5 5 pin_5_high
INT 9 5 exit_pin
SLEEP 60000
GOTO 3
```

```
---
pin_3_high:
SPRINTLN "Pin 3 status HIGH"
CALL main
---
pin_5_high:
SPRINTLN "Pin 5 status HIGH"
CALL main
---
exit_pin:
SPRINTLN "Bye bye"
HALT
---
```

PINMODE

PINMODE works the same way as Arduino framework does.

```
PINMODE pin mode
```

pin: Integer mode: One of: INPUT, OUTPUT, INPUT_PULLUP, INPUT_PULLDOWN

AREAD

Analog Read - directly from Arduino framework binding. Read analog value from given pin to destination variable

```
AREAD destination pin
```

AWRITE

Analog Write - Uses 8 bit resolution with 12kHz. This mimics arduino mega behavior on ESP32.

```
AWRITE pin value
```

DWRITE

Digital Write binding for Arduino framework.

```
DWRITE pin value
```

Value can be: 0 1 or HIGH, LOW

DREAD

Digital Read from given pin, another Arduino binding.

```
DREAD destination pin
```