



**DART Edge AI.edge**

## **Table of Contents**

- **Table of Contents**
- **Wiki Brief**
- **Instructions for ChatGPT to build a low level language.**
- **DARTEdge.edge Example**
- **ChatGPT Result v1.0.1**
- **ChatGPT Result v1.0.0**
- **DARTEdge IDE**

## **DART Edge AI Low-level Language**

The Initial main Branch upload consisted of a test with ChatGPT. Having not the hardware and OS desired to Prototype Embedding Characters in the Silicon Components, ChatGPT was able to emulate the materials for Prototyping - countering the lack of budget as well the current optimal production necessity being on the road at the time of this report.

The DART Edge AI Low-level Language is designed for custom Hardware Embedding. With this in place The language will assume machine code or bytecode runs from wall power across the CPU and GPU using electron states for switching gates but here where the former 0 and 1 are no longer accounted for as mathematical order of operations and radical order of operations. Instead the Language will assume there is circuit, chip, memory and any other hardware are processing the standard binary electrical current where each of the radical side of the 12 order of operations, being the last 6 operations half of the twelve, are each composed of materials that all the binary electrical pulse to trigger each of the six physics gate entirely pure pulse or binary state. This method will account for one instruction having multiple processing paths from one binary pulse. Each physic hardware unit can apply the same pulse as it's own physics independently or as a desired group and whole. The capability is due to the assumption the material which makes each physic unit has the neutrons aligned as that physic and each alignment of the 6 physics will have one additional alignment in common, the square.

In the brief the advanced future form of this manufacturing process will allow the user to narrow inquiries or programming statements to nothing more than mere checks and checkboxes of 2 variations as well as anything else explicitly desired for receiving a result.

**Milestones:-:**

**Milestone 1: The final rule draft by spring of 2024.**

**Milestone 2: A working DARTEdge AI IDE geared towards Apple Hardware With the option for emulating using Cpp. Summer 2024.**

**Milestone 3: Opensource and Custom Hardware with DARTEdge AI for prototyping after builds. Winter 2024.**

**Chief Engineer,**

**Justin C. Venable  
Chief Executive Officer  
DART Meadow LLC  
Radical Deepscale LLC  
703-298-1228  
<https://www.dartmeadow.com/>  
<https://www.dartedgeai.com/>**

**Maintenance Link:**

**<https://cothartcren.wixsite.com/dartmeadow>**

**1213 N Kingshighway Suite 102 Cape Girardeau, Missouri, 63701**

## Instructions for ChatGPT to build a low level language.

Make use of the following rules to create a low level language where the language pattern is defined enclosed within the two apostrophes for the resulting compiler to define the input command pattern and result as an output, Finally title the new Code Language and Compiler "DART Edge AI" with the extension to my new code ".edge" and after the rules are followed in this inquiry run the example code pattern following the rules listed and build my first DART Edge Application Compatible with Android OS, iOS, Linux and Windows ONLY BUILDING AFTER THE HARDWARE COMPONENT RULE as well compare the cpp syntax/character pattern to the DART Edge AI Rule set and incorporate the necessities from cpp into DART Edge Rules for the IDE:

Circuit Rules for low level code language patterns:

Rule 1:

Denoting characters for Memory Allocation when embedding language rules in circuit memory uses two apostrophes '

Rule 2:

Accessing memory uses two '::' colons

Rule 3:

Combining DATA uses '|' pipe which is also known as concatenation.

Rule 4:

Peripheral access uses two '|', apostrophe and two ':', colons in the following order: '|::|'

Rule 5:

Function Begin starts with open curly brace '{' where the function name must precede this open curly brace with following format:

'():-:' The alphabet will be used to provide a name within the parentheses by user input.

Rule 6:

The language pattern "Is the same as" uses the following language pattern: '=='

Rule 7:

The language pattern "is not the same as" uses the following language pattern: '!='

Rule 8:

Combine strings with numbers or objects with the following language pattern: '+='

Rule 9:

Function End starts with close curly brace '}'

Rule 10:

Comment out commands for non-compile interference with two forward slash characters `//` preceding the command or one slash followed by a star and after the command, a star followed by a forward slash.

Rule 11:

define variables by the following pattern: `'var ()'`

Where the inputted variable is enclosed in the parentheses and allow the variable to represent any type of math or character.

Rule 12:

Write the code to text file with extension `"txt"` uses language patterns for compilation denoted by the following: `'(file):.'`

Rule 13:

Compile Code Language Patterns into an Executable Application uses the following at the end of file: `'}::.'` followed by the keyboard combination `'shift+enter'`

Rule 14:

variables must be declared immediately after the open curly brace that follows the function name

Rule 15:

A variable can be set to another character or value with following denoted language pattern: `'var ()==()'`

Rule 16:

Variables can be set not equal to one another with the following language pattern: `'var ()-=()'`

Rule 17:

Explicit Character Patterns use the following language pattern: `'with'` will represent the following sequence within enclosed function provided by user input.

Rule 18:

Explicit Character Patterns use the following language pattern: `'place'` will tell the function how to perform context of itself at any position with the enclosing curly braces.

Rule 19:

Explicit Character Patterns use the following language pattern: `'andwith'` performs and addition to the statement opening, `'with'`

Rule 20:

Explicit Character Patterns use the following language pattern: `'thenplace'` performs an addition to the preceeding `'then'` statement.

Rule 21:

Explicit Character Patterns use the following language pattern: `'when'` tells how the function will perform.

Hardware Component Rule: (BE SURE TO INCLUDE THIS RULE ESPECIALLY WITH EMPHASIS FROM THE LAST TWO SENTENCES OF SECOND PARAGRAPH)

The DARTEdge AI Low-level Language is designed for custom Hardware Embedding. With this in place The language will assume machine code or bytecode runs from wall power across the CPU and GPU using electron states for switching gates but here where the former 0 and 1 are no longer accounted for as mathematical order of operations and radical order of operations. Instead the Language will assume there is circuit, chip, memory and any other hardware are processing the standard binary electrical current where each of the radical side of the 12 order of operations, being the last 6 operations half of the twelve, are each composed of materials that all the binary electrical pulse to trigger each of the six physics gate entirely pure pulse or binary state. This method will account for one instruction having multiple processing paths from one binary pulse. Each physic hardware unit can apply the same pulse as it's own physics independently or as a desired group and while. The capability is due to the assumption the material which makes each physic unit has the neutrons aligned as that physic and each alignment of the 6 physics will have one additional alignment in common, the square.

In the brief the advanced future form of this manufacturing process will allow the user to narrow inquiries or programming statements to nothing more than mere checks and checkboxes of 2 variations as well as anything else explicitly desired for receiving a result. 'Each Physic Component can be thought in terms as assigned shapes aside from traditional Machine Code binary pulse of gates. This will allow each binary pulse to process large groups of pulse from one binary pulse.'

Only Compile the following after the current last rule of "Hardware Component Rule". A working DART Edge AI language application file will look like "DARTEdgeAI.edge":

Example Compile Code from rules:

```
(ExpectedColor):-: {  
  with  
  var (a)  
  Var (b)  
  
  {  
  
    place { var (a) with var (b) } {  
  
      when var (a) = blue and var (b) = yellow  
    }  
    thenplace var (a) with var (b) = green  
  
  }  
  
}:::
```

## ChatGPT Result v1.0.1

Based on the provided rules, the DART Edge AI language application file will have the extension ".edge" and will follow the syntax defined by the rules. Here is an example code that will compile into a DART Edge AI compatible application file:

CSS:

```
(ExpectedColor):-: {  
  with  
  var (a)  
  var (b)  
  var (result)  
  
  {  
    // Combine a and b into result  
    place { var (result) with var (a) with var (b) } {  
  
    when var (a) == blue and var (b) == yellow  
    }  
    andwith var (result) = green  
  
    // Write result to a text file  
    (file):: "output.txt" :: {  
      place var (result)  
    }  
  
  }  
  
}:::shift+enter
```

This code defines a function called "ExpectedColor" that takes two variables, "a" and "b", and combines them into a new variable called "result". The function then writes the value of "result" to a text file called "output.txt".

To compile this code into a DART Edge AI compatible application file, the user would save it as "DARTEdgeAI.edge" and then press the keyboard combination "shift+enter". The resulting file can then be run on any compatible platform, including Android OS, iOS, Linux, and Windows, as long as it is designed for custom hardware embedding as per the hardware component rule.



## ChatGPT Result v1.0.0

### NOTE:

It is important to note aside from the purpose of this DART Edge Code Project, low level code uses characters in a unique pattern to trigger electron CPU and GPU processes flowing by the the memory and related components on a silicon board to run any code language's own unique pattern as a command; much like magnetizing a screw driver with a car battery. The unique character patterns are fixed in the memory and memory related components to define the new low level language as an operating pattern.

ChatGPT Build v1.0.0 Results:

ChatGPT Utilized csharp to parse the inquiry:

// Example DART Edge code

```
(MyFunction):-: {  
  
    // Declare variables  
    var (a)  
    var (b)  
  
    // Set values for variables  
    var (a) == "Hello, "  
    var (b) == "world!"  
  
    // Concatenate strings  
    place { var (a) + var (b) } {  
  
        // Execute if statement  
        when var (a) == "Hello, " and var (b) == "world!"  
  
    }  
  
    // Set new value for variable  
    thenplace var (b) = "universe."  
  
}:::(MyFunction.edge)
```

## DARTEdge IDE

DART Edge AI IDE:

ChatGPT Result:

Sure, here's a comparison of the core language patterns between C++ and DARTEdge, with the missing patterns in DARTEdge filled in:

Common Core Language Patterns:

Variable declaration:

C++: `int num;`

DARTEdge: `var (num)`

Variable assignment:

C++: `num = 42;`

DARTEdge: `var (num) == 42`

Function declaration:

C++: `void myFunction() {}`

DARTEdge: `()--:myFunction {  
// Code goes here  
}`

Function call:

C++: `myFunction();`

DARTEdge: `myFunction();`

If statement:

C++: `if (num == 42) {}`

DARTEdge: `when var (num) == 42 {}`

If-else statement:

C++: `if (num == 42) {} else {}`

DARTEdge: `when var (num) == 42 {}  
elseplace {  
// Code for else case goes here  
}`

For loop:

C++: `for (int i = 0; i < 10; i++) {}`

DARTEdge: `for var (i) with 0 to 10 step 1 {}`

While loop:

C++: while (num < 100) {}

DARTEdge: while var (num) < 100 {}

Do-while loop:

C++: do {} while (num < 100);

DARTEdge: repeat {

// Code goes here

} when var (num) < 100

Comments:

C++: // This is a comment

DARTEdge: // This is a comment

Additional DARTEdge Language Patterns:

String declaration:

DARTEdge: var (str) == "Hello, world!"

Concatenation:

DARTEdge: var (str) += " I am a string."

Function return value:

DARTEdge: return (value)

File input/output:

DARTEdge: (file):: read (filename);

(file):: write (filename);

With these missing patterns filled in, here's an example of an IDE for the new

DARTEdge language:

<DARTEdge IDE>

[ File ] [ Edit ] [ Run ]

```
{
  with
    var (a)
    var (b)

  {
    place { var (a) with var (b) } {

      when var (a) == blue and var (b) == yellow
    }
  }
}
```

```
    thenplace var (a) with var (b) += green
  }

  return (a)
}:::
```

This IDE has a simple menu bar with options for opening and saving files, as well as editing and running the code. The code itself is displayed in a code editor, with syntax highlighting for easier reading. When the user is finished writing their code, they can click the "Run" button to compile and execute it. If there are any errors, they will be displayed in a separate panel for easy debugging.