ESPEED32  
Slot Car Electronic Speed Controller

EN



Table of contents

[1 About this document 2](#_Toc179894760)

[2 Coding style and good practices 3](#_Toc179894761)

[2.1 Naming 3](#_Toc179894762)

[2.2 Comments 5](#_Toc179894763)

[2.3 Other good practices 6](#_Toc179894764)

[2.4 Versioning 7](#_Toc179894765)

[3 Code structure 8](#_Toc179894766)

[3.1 Task 1 9](#_Toc179894767)

[3.2 Task 2 13](#_Toc179894768)

[4 Revision History 18](#_Toc179894769)

# About this document

The goal of this document is to provide detailed information on how the code for the ESPEED32 Slot Car Electronic Speed Controller works, in order to ensure the possibility of future upgrades. This includes high level flow charts as well as low level implementation details.

This documents also includes hints at coding styles that **must be maintained** in order to obtain a clean, readable and understandable code.

Any changes to the code high-level structure or concept shall be documented here, as well as the rationale behind every change.

# Coding style and good practices

A good and coherent coding style allows the code to be easily understandable even to new programmers, ensuring the possibility to be maintained and upgraded in the future.

No more *“I don’t know what this function does, but if we remove it, it stops working”*.

It’s important that the standards that are defined in this chapter are meticulously followed.

## Naming

### Variables

The name of a variable should immediately convey functional relevant meaning. The name should also be made of fully formed, plain English words, rather than abbreviations that serve no purpose other than making the name harder to read:

uint16\_t selectedOption;

uint16\_t retValue;

uint16\_t throttleSetpoint; /\* Understandable English words \*/

Example 2.1 Good variable naming examples

uint16\_t xyz;

uint16\_t tmp;

uint16\_t throtStp; /\* Unintelligible abbreviations \*/

Example 2.2 Bad variable naming examples

To obtain a readable and coherent code, it’s important that a case style is chosen and followed throughout all the code. The proposed style is the following:

uint16\_t localVariable; /\* camelCase for variables, starting with a lower case \*/

#define CONSTANT\_MACRO /\* SNAKE\_CASE with all upper case letters for MACROS \*/

Example 2.3 Case style for variables and macros

Another good practice is to add a g\_ prefix in front of global variables, so that they can be identified immediately when looking at the code, while also preventing variable shadowing (local-scope variable with the same name of a global variable):

uint16\_t g\_globalVariable; /\* g\_ prefix for global variables \*/

Example 2.4 Global variables style

Finally, it might be useful to keep track of the measurement unit in the variable itself, especially when the measurement unit must be considered when the variable is used, or when there are similar variable expressed in different units. This is done by adding a suffux \_<unit>. When naming variables that store current or previous values, use the prefix curr or prev, respectively.

uint16\_t actualTrigger\_raw;

uint16\_t deadBand\_pct;

uint16\_t deltaTime\_uS;

Example 2.5 Good variable naming includes measurement units

### Functions

The guidelines for naming functions are the same as for naming variables:

* Meaningful names;
* Coherent style (camelCase);
* Using fully formed words rather than useless abbreviations.

The naming of the functions should also be consistent in the form of verbNoun or actionNoun style:

void showWelcomeScreen();

void initMenuItems();

uint16\_t addDeadBand();

Example 2.6 Good function naming examples

void displayInit(); /\* wrong order (nounVerb), should be verbNoun \*/

void calcThrot(); /\* Useless abbreviations make the name unintelligible \*/

uint16\_t init\_Encoder(); /\* wrong case style, should be camelCase \*/

Example 2.7 Bad function naming examples

### Typedefs

The naming conventions for enum and structs shall follow the same guidelines stated above, with the exception for the casing style, which shall be CamelCase, starting with an upper case letter (as opposed to variables, whose style is camelCase starting with a lower case letter). A suffix \_enum and \_type shall be added respectively to enumerations and structs:

typedef struct {

unit16\_t inputThrottle;

unit16\_t outputSpeed;

} ThrottleCurveSetpoint\_type;

typedef enum {

ITEM\_SELECTION,

VALUE\_SELECTION

} MenuState\_enum;

Example 2.8 Typedefs naming examples

## Comments

A good code is a commented code!

### Line comments

A portion of a code should never leave you wondering what the intent of the original programmer was: comments should follow every non-obvious statement and describe what it is doing.

Additionally, variables, struct members and macro should also be commented.

Line comments should be placed in the same line of the statement that they are describing, and should not be too long (continue the comment in the line below if needed).

Multiple adjacent line comments should be aligned, whenever it’s possible

Line comments shall start with /\* and end with \*/. Double slash comments (//) are not allowed!

uint16\_t carSelected; /\* index of the currently selected car \*/

unit16\_t encoderMainSelector; /\* this comment is aligned to the previous ones

If the comment is too long, continue it to a new line, but keep the commments aligned! \*/

unit16\_t deltaValue = abs(maxValue – minValue); /\* Calculate the difference between max and min value \*/

Example 2.9 Good line comments examples

### Function comments

Every function body shall be commented as to include:

* What it does in detail
* Important remarks on the implementation
* For every argument (parameter): what it is, and in what measurement unit is expected to be
* What value it returns, in what measurement unit

The comment should follow the doxyge’s format, as shown below:

/\*\*

\* Saturate an input value between an upper and lower bound

\*

\* @param paramValue The input value to be saturated

\* @param minValue The lower bound

\* @param maxValue The upper bound

\* @return The saturated input value.

\*/

uint16\_t saturateParamValue(uint16\_t paramValue, uint16\_t minValue, uint16\_t maxValue)

{

/\* Function does something here… \*/

}

Example 2.10 Good function comment example

## Other good practices

The following guidelines should be followed, in order to obtain a clean, readable and coherent code.

### Curly bracket placements { }

Curly bracket shall be opened in a new line, rather than on the same line.

It’s also a good practice to put a [space] after keyword statements (if, for, while, etc…).

if (condition)

{

/\* if true do something… \*/

}

else

{

/\* Do something else… \*/

}

Example 2.11 Correct curly bracket placement

Opening a curly bracket in the same line makes the code more difficult to read. Also, even if single line statements don’t require to be placed within curly bracket, it’s important to use them, for the sake of readability

if (condition) {

/\* if true do something… \*/

}

if (condition)

return;

Example 2.12 Wrong curly bracket placement

### Mathematical statements

To make statement more readable and understandable, apply the following guidelines:

* Try to fit all the operations in one single statement, if possible;
* Put a [space] in between operators and operands;
* Use parenthesis to make the math clear!
* Avoid magic numbers, use constants definitions

inputThrottle\_pct = (inputThrottle \* 100) / THROTTLE\_NORMALIZED;  /\* TakeinputThrottle (from 0 to THROTTLE NORMALIZED) and convert it to a 0% to 100% value \*/

counter ++; /\* Increase counter \*/

Example 2.13 Correct math statements style

inputThrottle\_pct = inputThrottle\*100/255;  /\* No space between operators.

No parenthesis, it’s unclear what is the order of the operations.

Magic numbers \*/

Example 2.14 Wrong math statements style

## Versioning

Keep versioning updated within the code. This allows to compare software versions

The following lines (13, 14 and 15 of the main .ino file) must be kept updated.

#define SW\_MAJOR\_VERSION 1

#define SW\_MINOR\_VERSION 99

/\* Last modified: 15/10/2024 \*/

Figure 2.1 Versioning code lines

In particular, the SW\_MAJOR\_VERSION and SW\_MINOR\_VERSION macro are also used by the code to verify during the initialization what version of the code is currently flashed in the device.

If the versions don’t match, the EEPROM memory is erased. **Keep updated the version numbers, especially if there was a change in the structure of the EEPROM stored variables!**

# Code structure

The code of the ESPEED32 is structured, as most microcontrollers, in two parts: Setup and Loop; in the case of the ESPEED32 the Loop part is replaced by two parallel Tasks, that run side by side on two different Cores.

The Setup simply consist of setting up the hardware pins and initializing the essential hardware components such as the magnetic trigger sensor and the motor controller; other non-essential components (such as the OLED display and rotary encoder) are initialized later, in order to reduce the start-up time.

After the Setup, two separate Tasks are defined and pinned to two different Cores.

Task 1 is the low priority task and is pinned to Core 0. This task takes care of managing the OLED display, which includes elaborating the user input (by means of the rotary encoder) to navigate the menu and change the values of the parameters, while also displaying important telemetry data to the user.

Task 2 is the high priority task and is pinned to Core 1. This task takes care of periodically (every 0.5 ms) reading the trigger position and processing it (throttle-to-speed pipeline), and correctly setting the PWM output of the motor controller.

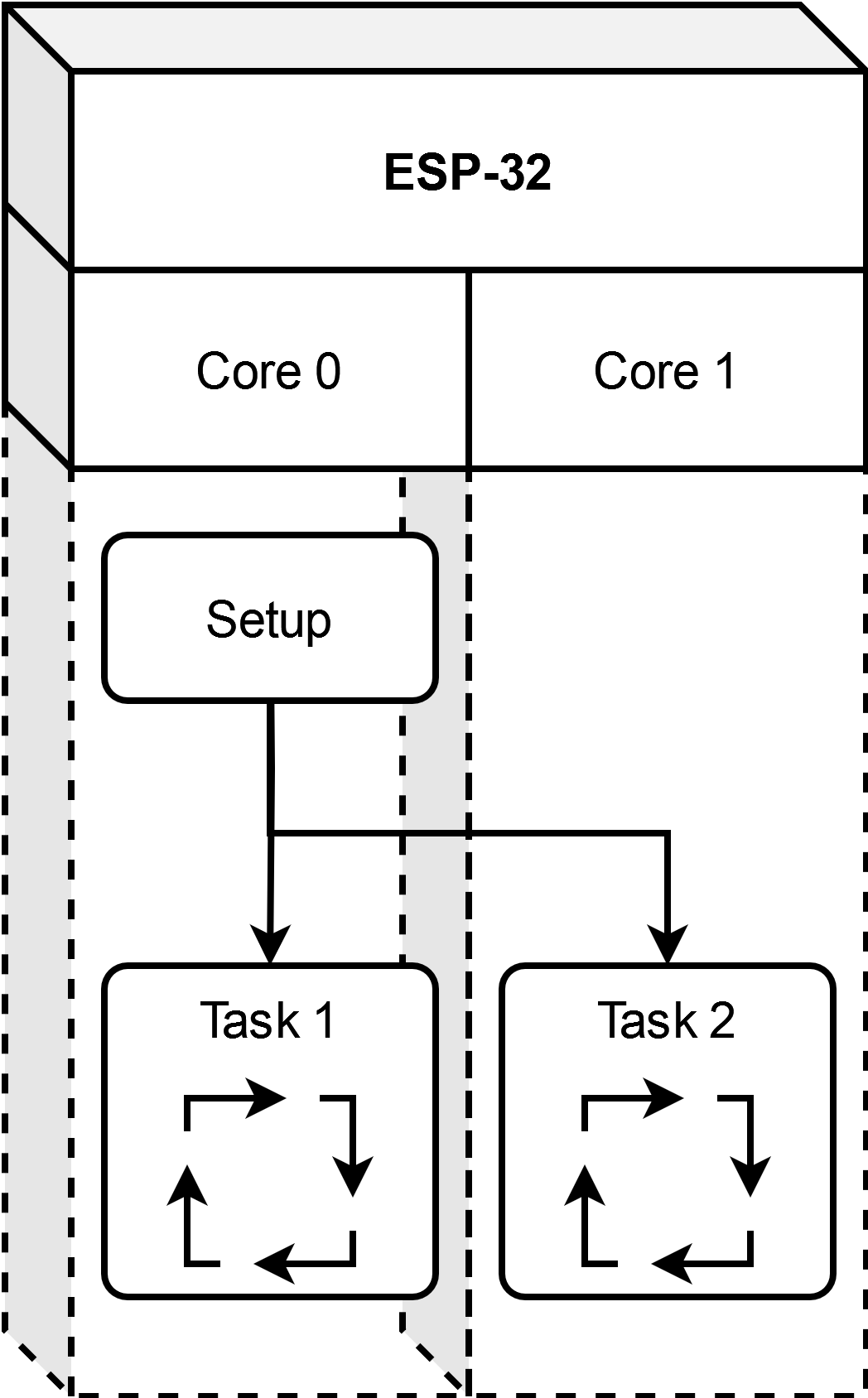


Figure 3.1 High-level structure of the main code

The implementation details of the two tasks are discussed in the sections below.

## Task 1

Task 1 is the lower priority task, that takes care of managing the OLED display. It manages the encoder inputs to allow the user to navigate the menu, select the items and change their value.

The Task 1 code is structured as a Finite State Machine with four states:

* **INIT**, where the EEPROM stored variables are checked. If they are okay, the machine transitions to the WELCOME state. If the stored variables are outdated or not present, the machine transitions to the CALIBRATION state. It’s also possible, for the user, to manually go to the CALIBRATION state by keeping pressed the encoder button during the power-on.
* **CALIBRATION**, where the user is asked to fully press and release the trigger, in order to save the maximum and minimum readings of the trigger sensor. Then the machine transitions to the WELCOME state.
* **WELCOME**, where a simple welcome screen is shown for 1 second, before transitioning to the RUNNING state.
* **RUNNING**, where the main menu is shown. The machine remains in this state, and the only way to transition is to power-off the device. This is the state where the encoder inputs are taken to navigate the menu, select the parameters and change their value.

There is an additional fifth state, which is the **FAULT**. At the current implementation, this state is empty and unreachable, but there might be an use for it in the future.

### Menu state

Figure 3.2 Task 1 state machine

While in the RUNNING state, the OLED display will show the main menu. The menu is a list of different items, followed by their value. The main menu will also behave as a state machine, with two possible state:

* ITEM\_SELECTION: this is the starting state, where scrolling the encoder will scroll the items in the menu. Only one item is highlighted (this is the selected item), and rotating the encoder will change the selected item. Upon pressing the encoder, the menu state will switch to VALUE\_SELECTION.
* VALUE\_SELECTION: in this state, the selected item’s value will also be highlighted. Rotating the encoder now, will change the value of the selected item. Upon pressing the encoder button again, the new value will be saved in the EEPROM, and the menu will transition back to the ITEM\_SELECTION state.

There also are special items, that upon selected will invoke a special callback. Once the callback returns, the menu will stay in the ITEM\_SELECTION state.

This menu state machine is handled in the rotary\_oButtonClick function, which is called at the beginning of the RUNNING state, if the encoder button has been clicked.

When transitioning from the ITEM\_SELECTION to the VALUE\_SELECTION menu state, the global variable g\_encoderSelectedValuePtr is set to point to the variable that stores the actual value of the selected item (e.g., when BRAKE is highlighted, pressing the encoder button will set g\_encoderSelectedValuePtr to point to g\_storedVar.carParam[g\_carSel].brake). Now, in the VALUE\_SELECTION menu state, rotating the encoder will increase and decrease the value of the actual variable, through the pointer.

### Menu and Menu items

The Menu\_type and MenuItem\_type structs were created to have a modular design for a scrollable menu, that allows to quickly and easily add items.

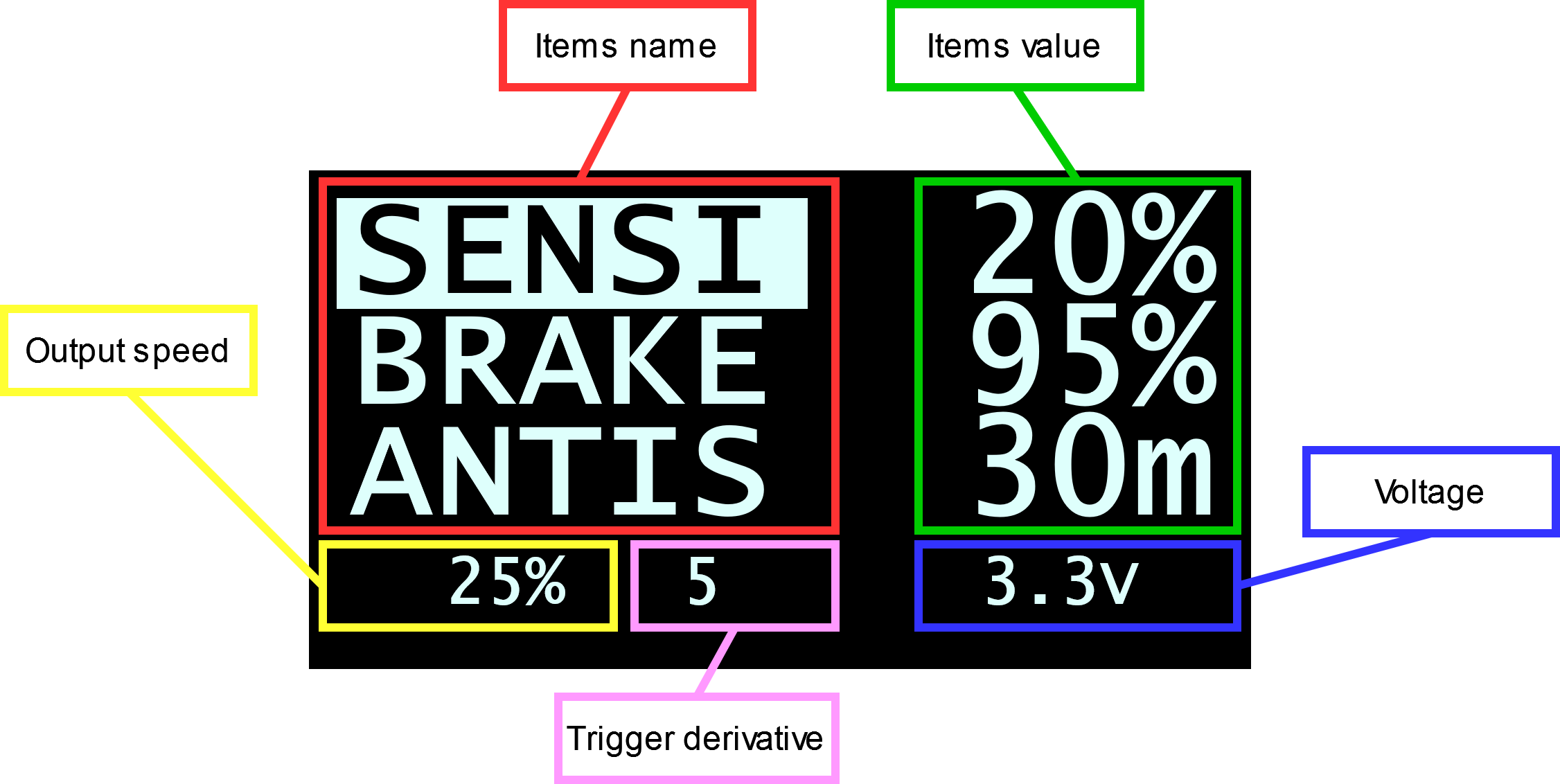
The menu and its items are only “graphical” elements, and they don’t actually store any parameter.

Figure 3.3 OLED display showing the main menu on the ITEM\_SELECTION state.

It’s easy to think as a menu as a series of items that are displayed on the screen, along with their value. The Menu\_type struct contains only two members: the uint16\_t lines, which indicates how many items are displayed at a time, and an array of MenuItem\_type, which are the actual items.

The MenuItem\_type was designed to be as modular as possible. It describes an item in a menu.

It contains an array of char to store the name of the item: that is the name that is displayed on the menu (e.g., “SENSI”, “BRAKE”, “CURVE”, etc…).

It contains a pointer to a value: this is done so that the actual value is stored in other global variables. The value could be a number (e.g., SENSI, BRAKE, CURVE, etc…) or it could be a string (e.g., the CAR item value is the name of the selected CAR). An item can also not contain a value. The contents of the pointer (address of the variable that store the actual value) is passed to g\_encoderSelectedValuePtr when the item is selected, as explained in [Menu state](#_Menu_state).

The item also contains the upper and lower boundaries for the actual value, that are used to set the encoder boundaries in rotary\_oButtonClick.

Finally, an item can contain a callback, that is a pointer to a function that shall be execute when the item is selected and the encoder button is pressed.

Table 1 Details of the MenuItem\_type struct

|  |  |  |
| --- | --- | --- |
| Member | Type | Description |
| name | char[] | The name of the item, as it will be displayed on the menu, stored in an array of char. |
| value | void \* | Pointer to the variable that contains the value of the item, stored as a generic void pointer. This is done so that it can print to different type of variables. When assigning this value, it must be cast to (void \*)  Value = (void \*)&g\_storedVar.carParam[g\_carSel].brake;  If the item doesn’t contain any value, this must be set to ITEM\_NO\_VALUE. |
| type | ItemValueType\_enum | Indicates which type of variable address is stored in value. Possible enumerations are   * VALUE\_TYPE\_INTEGER: value points to a uint16\_t variable (must be 3 digits) * VALUE\_TYPE\_DECIMAL: value points to a uint16\_t variable, but it will be printed as a decimal value (see decimalPoint). * VALUE\_TYPE\_STRING: value points to an array of chars. |
| maxValue | uint16\_t | The maximum integer value that can be assigned to the actual variable. Only applies if type is VALUE\_TYPE\_INTEGER or VALUE\_TYPE\_DECIMAL. |
| minValue | uint16\_t | The minimum integer value that can be assigned to the actual variable. Only applies if type is VALUE\_TYPE\_INTEGER or VALUE\_TYPE\_DECIMAL. |
| unit | char | A single characted that represents the measurement units of the value, which will be printed next to the value in the menu. |
| decimalPoint | uint8\_t | In case type is set to VALUE\_TYPE\_DECIMAL, this indicates where the decimal point is placed. Possible values are 1 (decimal point is placed between right-most digit and middle digit) or 2 (decimal point is placed between left-most digit and middle digit). The variable holding the actual value must store an integer value (it’s up to the application to convert it when it’s used).  e.g.: PWM freq is a value that we want represented in kHz, ranging from 0.2 to 5.0. We are gonna create the menu item type with the following fields:  type: VALUE\_TYPE\_DECIMAL;  maxValue: 50;  minValue: 2;  decimalPoint: 1  Meaning that internally the values ranges from 2 to 50. When selected with the encoder in the main menu the value can be increased 1 by 1. The value that is displayed will be divided by 10^decimalPoint, to show values that ranges from 0.2 to 5.0. The application must take into account that the stored value will range from 2 to 50 during the calculations. |
| callback | FunctionPointer\_type | The address of a function with no arguments that returns void, that will be invoked when the item is selected (e.g., CURVE item has callback = &showCurveSelection).  If no callback, this should be set to ITEM\_NO\_CALLBACK |

### Printing the menu

The code does not have a standard function to print the menu, since the Menu\_type struct is used both for the main menu and the select car menu, which have different necessities; however, the same “Frame” approach is used.

A Frame is delimited by two numbers, a lower bound and an upper bound. The difference between the upper and lower bound is fixed to be the value of Menu\_type.lines. Only the menu items within the Frame are displayed on screen. Scrolling the encoder allows to select the item displayed on screen. If the value of the encoder crosses the boundaries of the Frame, the Frame is shifted, so that the value of the encoder is within the Frame.

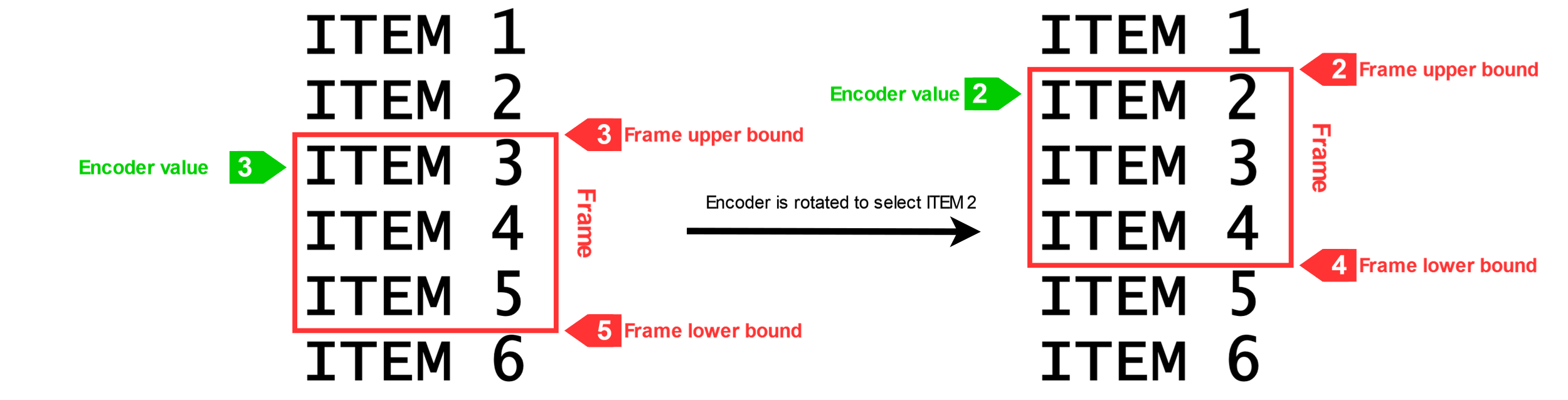
The items and their value are printed in a for loop for (uint8\_t i = 0; i < menu.lines; i++) where they are addressed as menu.item[frameUpper + i]. It’s important that the encoder boundaries match the item array size.

Figure 3.4 Frame explanation.

## Task 2

Task 2 is the high priority task, that takes care of calculating the correct speed output (duty cycle) depending on the user parameters and trigger position.

The task code is executed every 0.5 ms (500 µs), in order to provide a timely and smooth response to the trigger.

The code of Task 2 can be divided into three parts:

* Read Trigger position (input)
* Throttle-to-speed pipeline (elaboration)
* Set PWM (output)

### Trigger reading

Figure 3.5 Task 2 flow chart

The trigger position is read at the beginning of the periodic Task, by calling the HAL\_ReadTriggerRaw() function; this produces a **raw** value. The current raw trigger reading and the previous one (kept from the previous execution) are summed and divided by two, then stored in the escVar.trigger\_raw variable.

### Throttle-to-speed pipeline

This part of the Task takes the trigger\_raw input and elaborates it to produce a speed output. The pipeline is composed of different elaborations, which are described below.

#### ****Normalization****

The raw input is taken, constraint between the maximum and minimum calibration values of the sensor, then re-scaled in the range 0 to THROTTLE\_NORMALIZED (defined to be 256), which has a higher granularity with respect to the typical 0-100 range. This passage produces a **normalized** value.

#### DeadBand

A 3% deadband is added to the trigger\_raw, to increase stability at full-pressed and full-released trigger positions. This elaboration sets to 0 every input that’s lesser than 3% of normalized range, and sets to THROTTLE\_NORMALIZED every value that’s higher than 97% of the range.

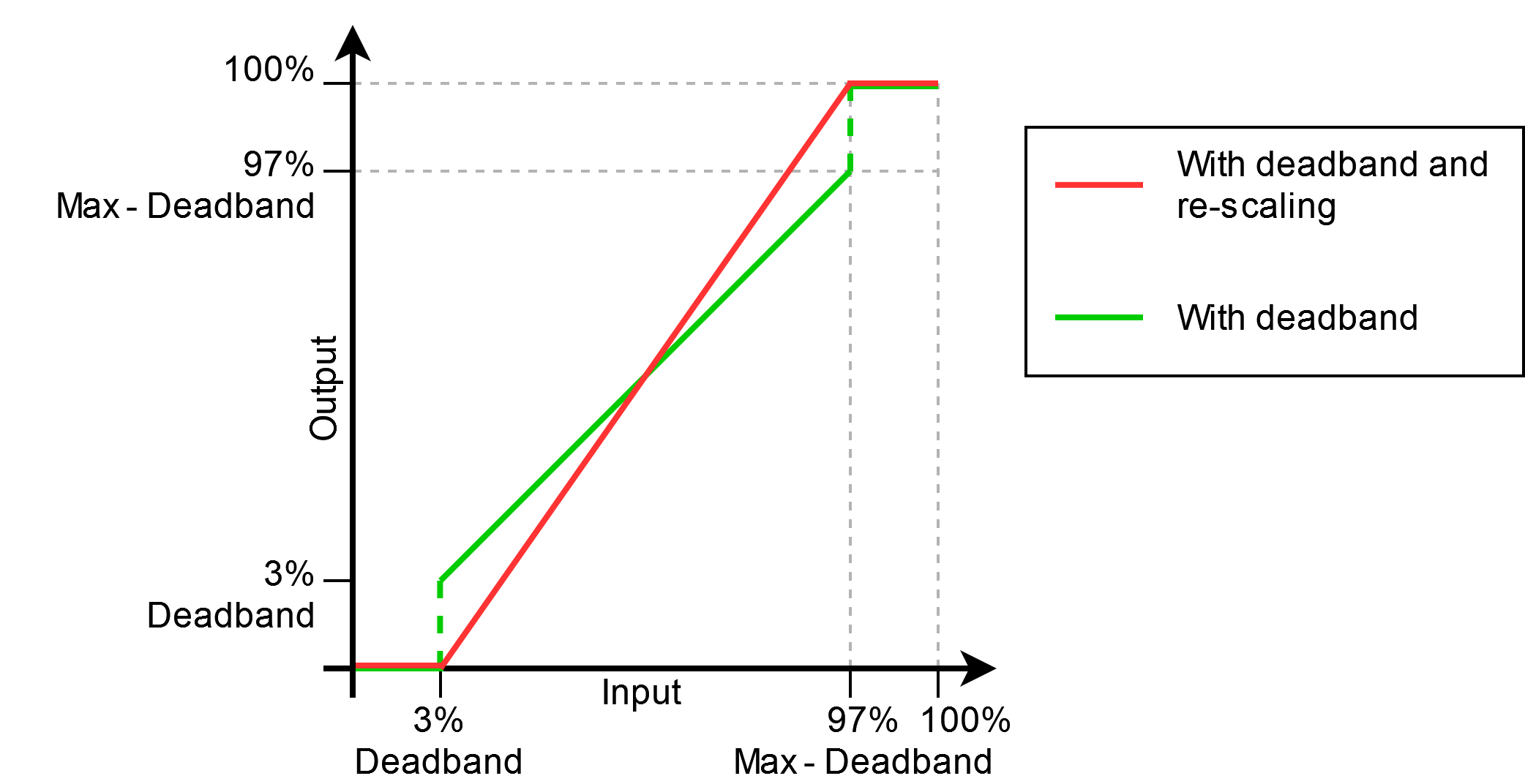
Then, all the in-between values, that are not affected by the deadband and range from 3% to 97% of THROTTLE\_NORMALIZED, are scaled from 0 to THROTTLE\_NORMALIZED, to eliminate the initial and final step.

Figure 3.6 Output of the deadband elaboration (green) and scaling (red).

#### ****Throttle Curve****

After the deadband, we have a smooth, normalized throttle value. This value needs to be mapped to a output speed (duty cycle). The throttle curve is a function that takes as input a normalized throttle value (that ranges from 0 to THROTTLE\_NORMALIZED) and returns a speed output expressed as duty cycle percentage (that ranges from 0% to 100%).

To be precise, the throttle curve returns a speed output that ranges from the min speed (SENSI parameter) to the max speed (LIMIT parameter)

The throttle curve is composed of two consecutive segments that intersect in a single point called vertex.

The vertex point coordinates are stored in the throttleCurveVertex variable in the following way:

* inputThrottle is the X coordinate of the vertex and is fixed by default at 50% of the throttle range (that is THROTTLE\_NORMALIZED / 2). At this moment the user has no way to change this value
* curveSpeedDiff indicates how to calculate the Y coordinate of the vertex: this variable indicates the value of the Y coordinate as a percentage of the difference between the min and max speed (*e.g.,* curveSpeedDiff = 50% means that the Y coordinate of the vertex is exactly the middle point between Min and Max speed).   
  This implementation was chosen over an absolute value because it adapts automatically to changes in min and max speed. It is also more understandable from the user point of view. **This variable corresponds to the** CURVE **parameter**.

The Y coordinate of the vertex is calculated as:

Where CURVE is a percentage which ranges from 10% to 90%.

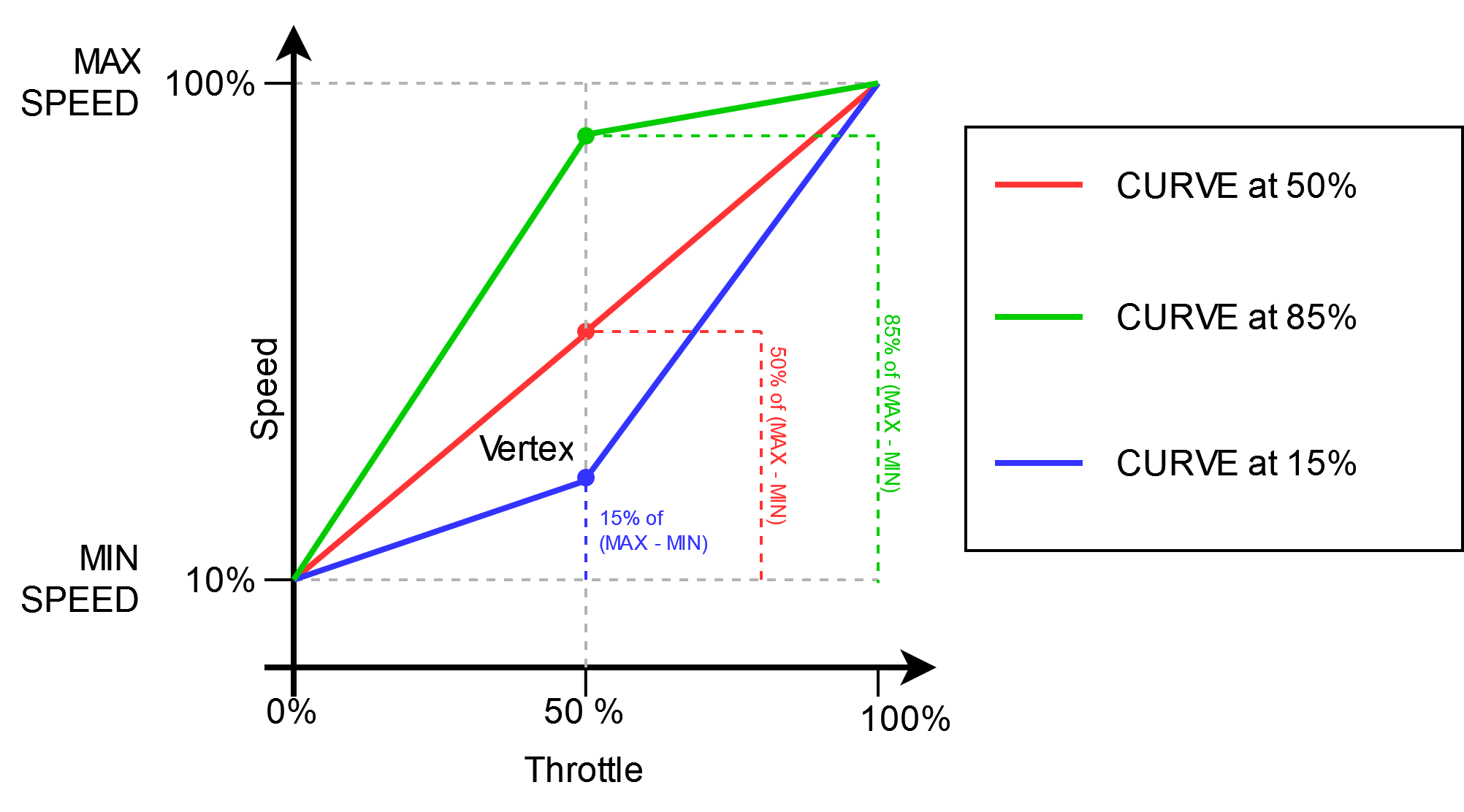
Once the coordinates of the throttle curve vertex are known (fixed at and calculated as shown above), the throttle curve is simply a function defined by cases

Figure 3.7 Example of throttle curve mapping with different CURVE values.

#### Antispin

The last passage of the throttle-to-speed pipeline is the antispin calculation. This passage applies the ANTIS parameter that limits the maximum increase in speed, in order to prevent the car from spinning due to a sudden acceleration.

The calculation takes into account the time between execution, the current requested speed and the output speed of the antispin calculation of the previous execution. The speed variables used in the calculation are multiplied by 1000, to increase the granularity.

Antispin is only applied during acceleration (when the current requested speed is higher than the previous output speed), while during deceleration the requested speed is not modified.

In this passage, the maximum possible speed increase between the current and previous execution is calculated as:

Where is the time between executions, and is the antispin parameter.

Then the output speed is calculated as

Ensuring that the increase in speed is at most .

### Setting the PWM

At the end of the throttle-to-speed pipeline we have obtained a speed value that ranges from 0% to 100%. This speed value corresponds to the duty cycles of the PWM that controls the car motor.

In this last part of the Task 2 code, the PWM is set calling the HalfBridge\_SetPwmDrag function, that takes as argument the duty cycle percentage and the drag percentage.

If the speed is 0%, then the output duty cycle will be 0%, and only the brake is applied (whose value corresponds to the BRAKE parameter).

If the speed is greater than 0%, than the speed value will be used as output duty cycle, and the value of the drag will be proportional to the trigger derivative, scaled from 0 to DRAGB.

In this case, only the negative derivative is used (when the trigger is being released).

#### Computing the trigger derivative

The trigger derivative is computer during the throttle-to-speed pipeline, and is calculated using the trigger value, after the normalization and the deadband elaboration.

A circular buffer that stores the latest 60 trigger values is used (that is, the amount of Task 2 executions that occur in the span of TRIG\_AVG\_TIME, 30 ms). At every execution, the current trigger value replaces the oldest value in the buffer.

The rolling average trigger value is calculated as the average of all the trigger values in the buffer, and the derivative is computer as the difference between the current trigger and the rolling average.

This operation returns a signed 16-bit integer value.

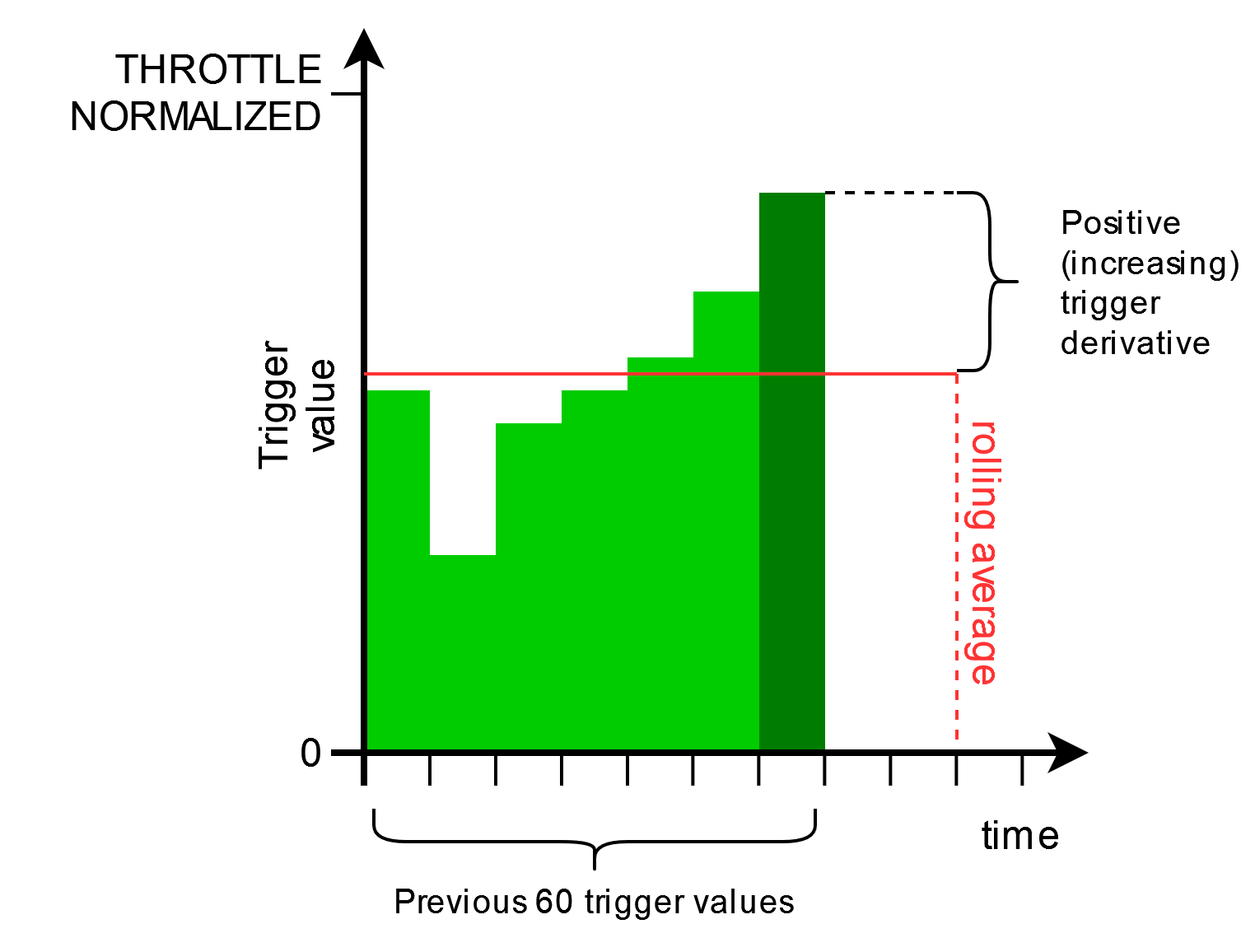


Figure 3.8 Example of trigger derivative calculation.

# Revision History

|  |  |  |
| --- | --- | --- |
| Version | Date | Description |
| V1.0 | 15/10/2024 | Initial version |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |