ESPEED32  
Slot Car Electronic Speed Controller

EN



Table of contents

[1 About this document 2](#_Toc179472025)

[2 Coding style and good practices 2](#_Toc179472026)

[2.1 Naming 2](#_Toc179472027)

[2.2 Comments 4](#_Toc179472028)

[2.3 Other good practices 5](#_Toc179472029)

[2.4 Tips 6](#_Toc179472030)

[2.5 Electrical Specification 7](#_Toc179472031)

# 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.

# 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. 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. 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. 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. 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

## Tips

When adding pictures always prefer scalable formats (.eps, .svg, etc…) and always add a caption **below** the picture

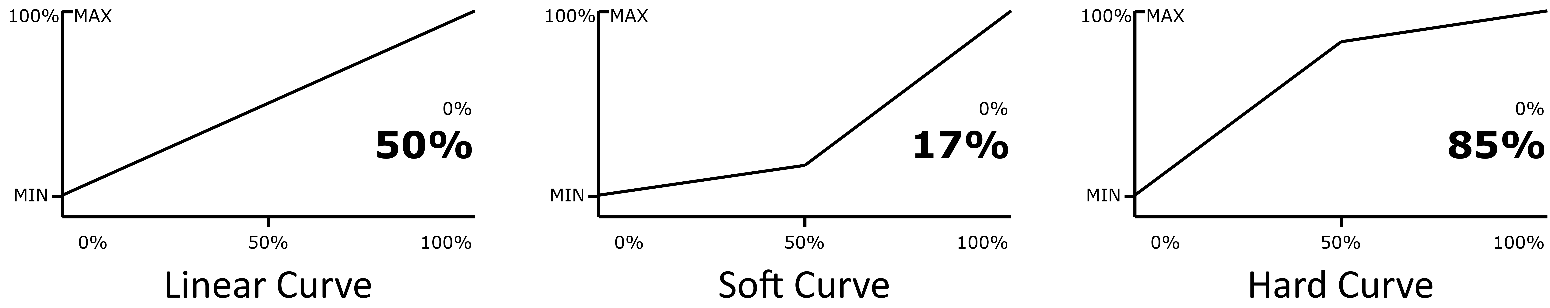


Figure 1 Example of a picture.

When adding a table use the custom format and make sure the table length stretches until the margins (use View 🡪 gridlines to make sure you reach the margin): in this way all the tables have the same length. Don’t forget to add a caption **above** the table. Also, try not to split table over multiple pages if possible.

Table 1 Settings overview

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ****Setting**** | ****Recommended value**** | ****Range**** | | ****Unit**** |
| **Min.** | **Max.** |
| SENSI | 10 | 0 | 90 | % |
| BRAKE | 90 | 0 | 100 | % |
| ANTIS | 130 for 1/32 cars  0 for 1/24 cars | 0 | 250 | ms |
| CURVE | 50 | 10 | 90 | % |
| DRAGB | 100 | 0 | 100 | % |
| PWM\_F | 3 | 0.2 | 5.0 | kHz |

## Electrical Specification

The parameters listed in this section represent the characteristics of the ESPEED32 Electronic Speed Controller and its requirements to ensure correct operation.

Table Electrical specifications

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ****Parameter**** | ****Symbol**** | ****Value**** | | | ****Unit**** |
| **Min.** | **Typ.** | **Max.** |
| Supply voltage |  |  |  |  | V |
| Input current |  |  |  |  | mA |
| Operating temperature |  | -40 | - | 125 | °C |