Using SDL2
with eForth Windows

version 1.0 - mardi 29 octobre 2024

Autor

- Marc PETREMANN

Contents

Introduction

SDL2, or Simple DirectMedia Layer 2, is a cross-platform library designed for handling graphics, audio, input, and other multimedia elements. It is widely used for the development of video games and multimedia applications.

Here are some key features of SDL2:

- Cross-platform: SDL2 works on Windows, macOS, Linux, iOS, and Android, allowing developers to create applications that run on multiple operating systems.
- Graphics Management: SDL2 allows you to draw 2D graphics and manage textures, surfaces and images.
- Audio Support: The library allows playing sounds and music, making it easy to integrate audio into applications.
- Input Management: SDL2 manages input from keyboards, mice, gamepads, and other devices.
- Extensibility: SDL2 is often used with other libraries to extend its functionality, such as OpenGL for 3D graphics.

In summary, SDL2 is a powerful and flexible tool for developers looking to create multimedia applications and games.

Installing eForth and SDL2

Installing the Forth language development environment for SDL2 requires only two components:

- Brad NELSON's eForth version for Windows;
- the SDL2 library in a dll file;
- A good text file editor.

It is a very compact environment which combines, thanks to eForth, an interpreter and a compiler.

Installing eForth for Windows

The eForth version for Windows is available here:
https://eforth.appspot.com/windows.html

Download the uEf64-7.0.7.20.exe version. This is a 64-bit version. It is very stable and very robust. The file is only 265 KB.

Create an eforth folder in your usual workspace:

eforth

Copy the previously downloaded file into this folder:

eforth
uEf64-7.0.7.20.exe

Run this program from this **eforth folder** . Often, Windows issues a warning. If so, accept the execution of this program. You should end up with this window:

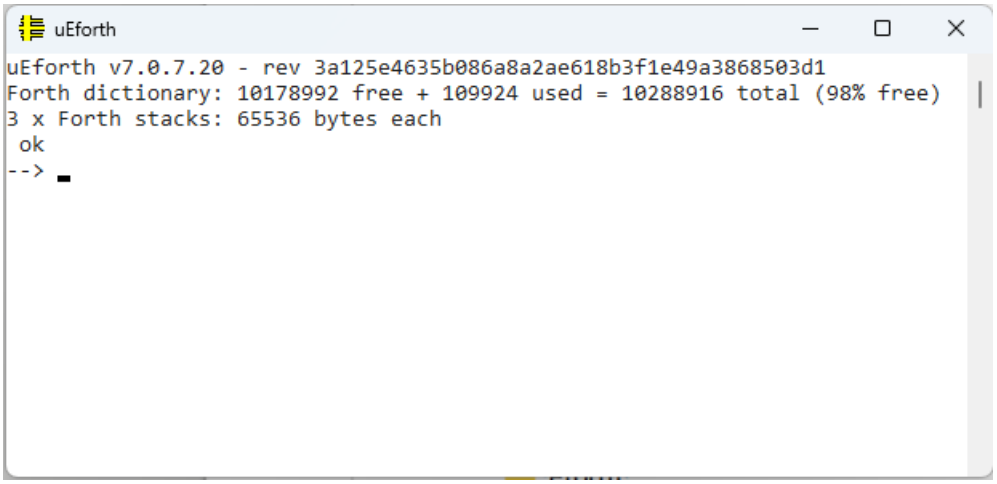


Figure 1: eForth window

There you have it! You have your hands on a Forth version with three stacks:

- a stack of data
- a return stack
- a stack for real numbers

Each stack has a data space of 64KB. Since each element weighs 64 bits (8 bytes) in the stack, that's 8000 values that can be stacked!

The dictionary has 10,178,992 bytes of free space! So we have more than enough development space.

eForth is an interpreter AND a compiler. The very first word at your disposal, just to see what's under the hood, is `words,` which when executed displays this, summarized in the first three lines:

```
FORTH graphics argv argc visual set-title page at-xy normal bg fg ansi
editor list copy thru load flush update empty-buffers buffer block save-buffers
default-use use open-blocks block-id scr block-fid file-exists? needs required…
...etc.
```

All these words make up the language for compiling and running programs written in the Forth language.

Installing the SDL2 library

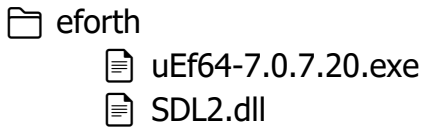
Access to the SDL2 site: https://www.libsdl.org/

This site provides a lot of resources and documentation to understand the use of the SDL2 library. WARNING: the C language functions are adapted to eForth Windows. We will see this later.

To download the latest SDL2 version:
https://github.com/libsdl-org/SDL/releases/tag/release-2.30.8

Get the **SDL2-2.30.8-win32-x64.zip file** .

Open this zip file and transfer the single **SDL2.dll file** to the eforth directory:



That's it! There's nothing else to do on the installation side. Now let's see how to prepare the development environment.

File organization

SDL2 subdirectory :

- eforth
  - uEf64-7.0.7.20.exe
  - SDL2.dll
  - SDL2

We will fill this directory with the files available here:
https://github.com/MPETREMANN11/SDL2-eForth-windows/tree/main/SDL2

Retrieve only these files:

- eforth
  - uEf64-7.0.7.20.exe
  - SDL2.dll
  - SDL2
    - SDL2.fs
    - SDLconstants.fs
    - main.fs
    - tests.fs

Contents of these files:

- **SDL2.fs** contains all the words accessing the SDL library contained in **SDL2.dll** ;
- **SDLconstants.fs** contains a number of commonly used constants specific to their use with words defined in the `SDL2` vocabulary ;
- **main.fs** is the main script responsible for aggregating the various components of your application;
- **tests.fs** catch-all file for performing tests.

WARNING: The content of these files is likely to change constantly on the Github repository. It is therefore strongly recommended to monitor their content.

There is one last file to install in the **eforth directory** :

- eforth
  - uEf64-7.0.7.20.exe
  - SDL2.dll
  - SDL2
    - SDL2.fs
    - SDLconstants.fs
    - main.fs
    - tests. fs
  - SDL2.fs

Contents of this **SDL2.fs file**

```
\ pre-load tools
```

```
\ s" tools/dumpTool.fs" required
\ load SDL2
s" SDL2/main.fs" included
```

Consider the contents of this file as a batch process, but executable only from eForth.

To check if the SDL2 library is working properly, launch eForth, then enter this command:

```
include SDL2.fs
```

This command loads the contents of the **SDL2.fs** file located in the root of the eforth subdirectory . Then type:

```
SDL2 vlist
```

You should see the words defined in the SDL2 vocabulary appear, here the first three lines:

```
SDL.CreateWindow SDL.init SetRenderDrawColor Quit RenderPresent RenderClear
PollEvent Init GetError GetCursor GetCPUCount GetBasePath GetAudioStatus
DestroyWindow DestroyRenderer CreateWindow CreateRenderer SDL_MAX_LOG_MESSAGE
...etc…
```

ATTENTION: The content of this vocabulary is constantly evolving. It only represents the compilation of the definitions described in the **SDL2/SDL2.fs** file .

The main.fs file

This is the file containing the script responsible for aggregating the components of your applications. Example of the content of this file:

```
\ load SDL2 library
s" SDL2.fs"     included

\ load SDL2 tests
s" tests.fs"    included
```

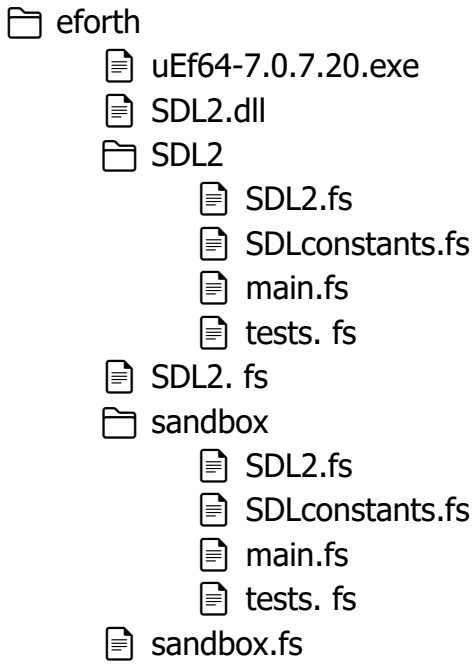
You can modify its content without any worries. Let's take the case where you want to test the display of windows. You create a file **testWindows.fs** in which you will put the window tests with the words of the **SDL2** vocabulary. Here is how to integrate this file **testWindows.fs** into **main.fs** :

```
\ load SDL2 library
s" SDL2.fs"     included

\ load SDL2 tests
\ s" tests.fs"    included

\ load windows tests with SDL2
s" testWindowss.fs" included
```

The word \ comments out the rest of the line. The next time **SDL2.fs** is loaded which is in the **eforth** root directory , only the contents of the files **SDL2/SDL2.fs** and **SDL2/testWindows.fs** will be processed by eForth.

So you can easily chain tests or portions of code together for the final application.

To avoid accidentally overwriting your work, it is advisable to create several subdirectories, for example sandbox where you can do lots of small tests:



Example of the contents of **sandbox.fs** :

```
\ s" tools/dumpTool.fs" required
\ load sandbox
s" sandbox/main.fs" included
```

So, when launching eForth, you just need to enter:

```
include sandbox.fs
```

It's up to you to organize yourself to be as efficient as possible during your developments.

Editing and managing source files for eForth

As with the vast majority of programming languages, source files written in Forth are in plain text format. The extension of Forth language files is free:

- **txt** generic extension for all text files;
- **forth** used by some Forth programmers;
- **fth** compressed form for Forth;
- **4th** other compressed form for Forth;
- **fs** our favorite extension…

Text File Editors

edit file editor is the simplest:

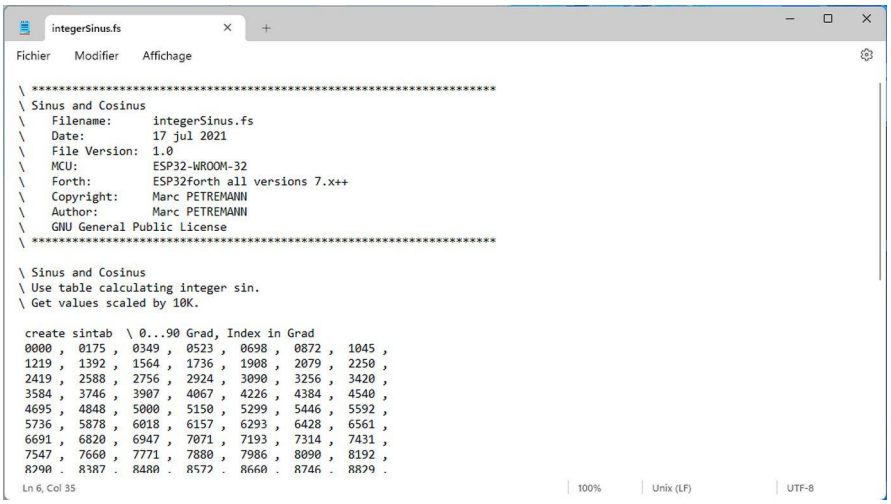


Figure 2: editing with edit under windows 11

Other editors, such as **WordPad** , are not recommended because you risk saving the Forth source code in a file format that is not compatible with eForth.

If you use a custom file extension, such as **fs** , for your Forth source files, you must have this file extension recognized by your system to allow them to be opened by the text editor.

Using an IDE

Nothing prevents you from using an IDE [1]. For my part, I have a preference for **Netbeans** that I also use for PHP, MySQL, Javascript, C, assembler… It is a very powerful IDE and as efficient as **Eclipse** :

1 Integrated Development Environment

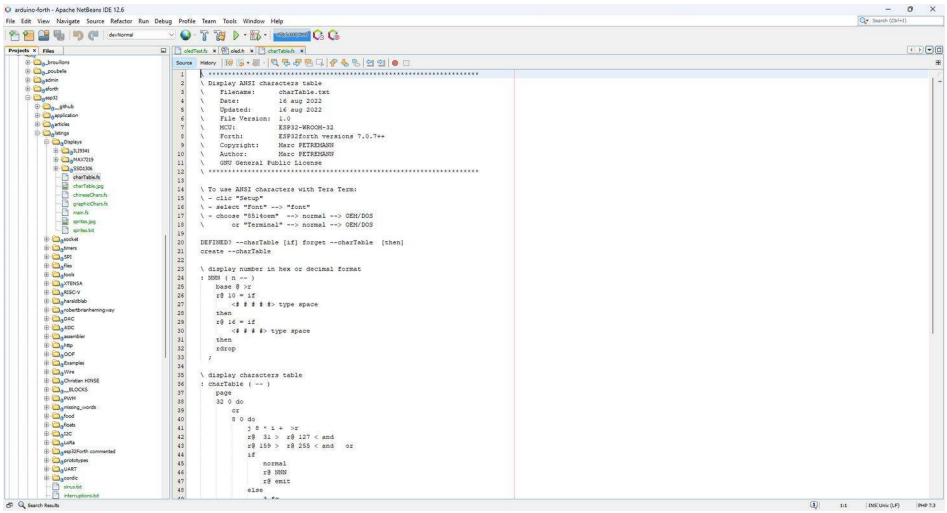


Figure 3: editing with Netbeans

Netbeans offers several interesting features:

- version control with **GIT** ;
- recovery of previous versions of modified files;
- file comparison with **Diff** ;
- one-click **FTP transmission** to the online hosting of your choice;

With the **GIT option** , you can share files on a repository and manage collaborations on complex projects. Locally or collaboratively, **GIT** allows you to manage different versions of the same project, then merge these versions. You can create your local GIT repository. Each time you *commit* a file or a complete directory, the developments are kept as is. This allows you to find old versions of the same file or folder of files.

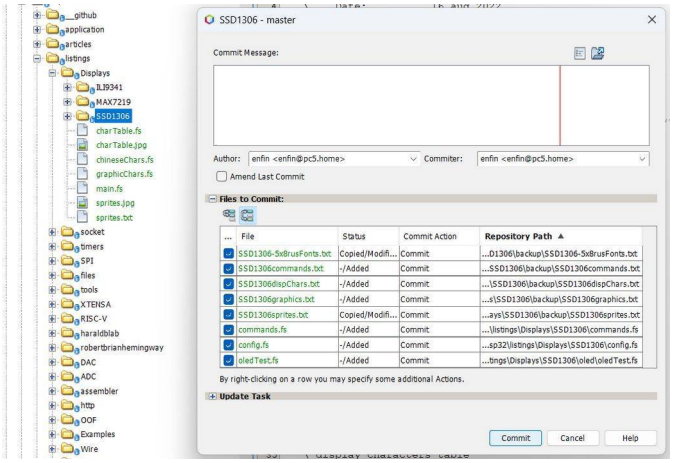


Figure 4: GIT operation in Netbeans

With NetBeans you can define a development branch for a complex project. Here we create a new branch:

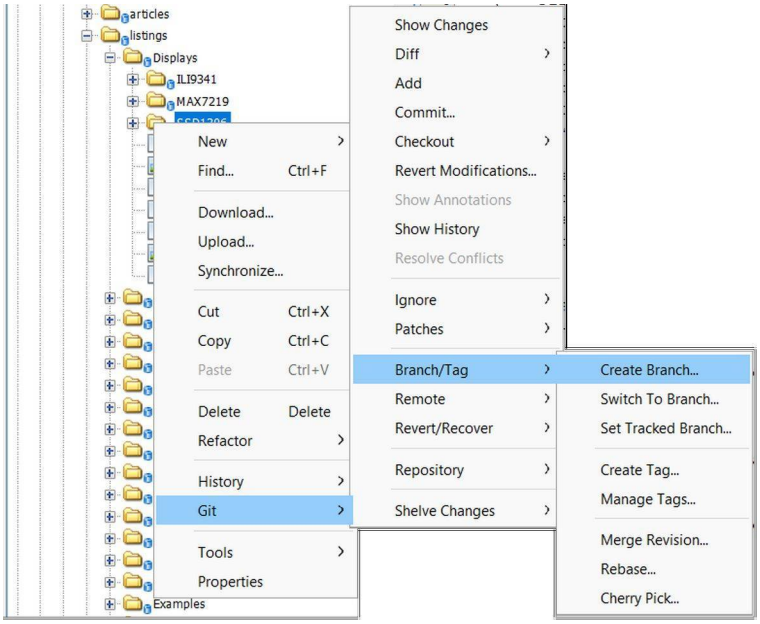


Figure 5: creating a branch on a project

Example of a situation that justifies the creation of a branch:

- you have a working project;
- you are considering optimizing it;
- create a branch and do the optimizations in that branch…

Changes to source files in a branch do not affect files in the *main trunk* .

Incidentally, it is more than advisable to have a physical backup medium. An SSD hard drive costs around €50 for 300Gb of storage space. The read or write access speed of an SSD medium is simply amazing!

Storage on GitHub

GitHub [2]website is, along with **SourceForge** [3], one of the best places to store your source files. On GitHub, you can share a working folder with other developers and manage complex projects. The Netbeans editor can connect to the project and allows you to push or retrieve file changes.

2 https://github.com/
3 https://sourceforge.net/

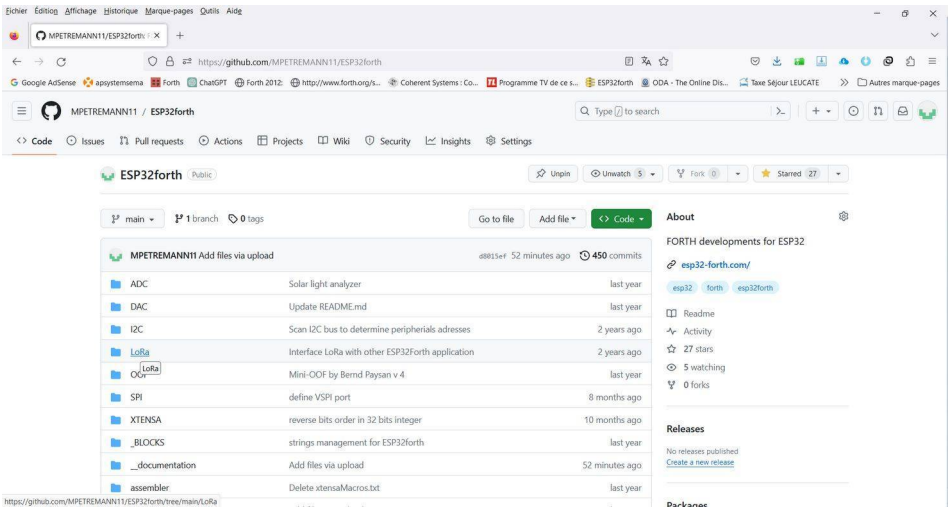


Figure 6: storing files on Github

On **GitHub** , you can manage project branches (*forks*). You can also make parts of your projects confidential. Here are the branches in flagxor/ueforth's projects:

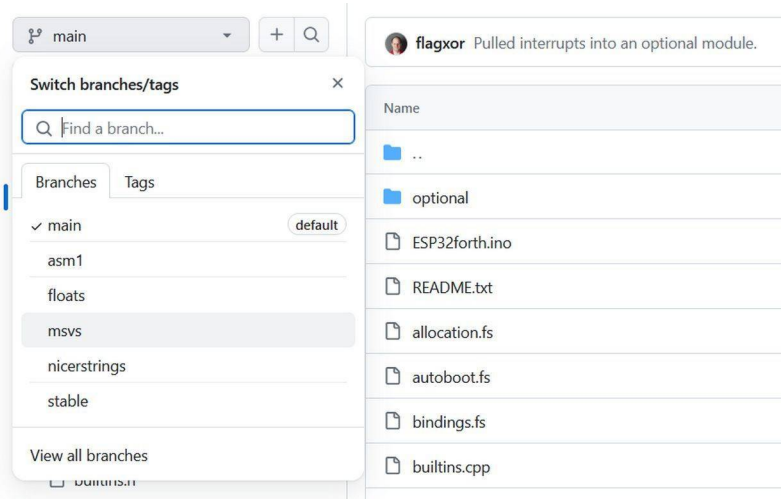


Figure 7: access to a project branch

Some good practices

The first best practice is to name your working files and folders well. You are developing for eForth, so create a folder named **eforth** .

sandbox subfolder in this folder .

cards subfolder .

If you have general-purpose scripts, create a **tools folder** . If you use a file from that **tools folder** in a project, copy and paste that file into that project's folder. This will prevent a change to a file in **tools** from disrupting your project later.

The second best practice is to split the source code of a project into several files:

- **config.fs** to store project settings;

- **documentation** directory to store files in the format of your choice, related to the project documentation;
- **myApp.fs** for your project definitions. Choose a file name that is fairly descriptive. For example, to manage a **pacman** game , take the name **pacman-commands.fs** .

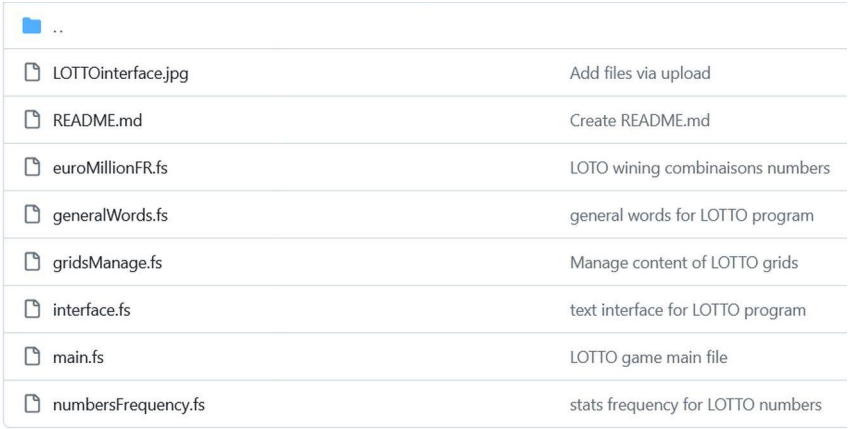


Figure 8: Forth source file naming example

It is the contents of these files that will be processed by eForth via `include` so that eForth interprets and compiles the Forth code.

Creating the SDL2 vocabulary

We will see how the binding is ensured between eForth Windows and the functions available in **SDL2.dll** .

The keyword is `dll` . This word is available in the `windows vocabulary` . It creates an access ticket to windows resources. If you have correctly installed eForth windows and **SDL2.dll** , we will create our access ticket:

```
windows
\ Entry point to SDL2.dll library
z" SDL2.dll" dll SDL2.dll
```

Here, our ticket has been named `SDL2.dll` . We could have named it more simply `SDL2` or `SDL2ticket…`

Creating and Using a binding Ticket

Let's take the line that creates the binding ticket:

```
z" SDL2.dll" dll SDL2.dll
```

We see there:

- in blue, the name of the library file to which we will connect by binding;
- in green the word `dll` which creates the binding ticket;
- in red our binding ticket.

SDL2 Vocabulary Definition

In order to avoid any software collision with predefined words from the forth vocabulary, it is strongly recommended to define all words specific to software binding in a specific vocabulary:

```
vocabulary SDL2
```

This creates the `SDL2` vocabulary. All binding words with `SDL2.dll` will be defined in this vocabulary. To create these new definitions, we will need to access the other `forth`, `windows` and `structures` vocabularies :

```
only FORTH also  windows also  structures also
```

And finally, we force the definition of new words in this `SDL2` vocabulary :

```
SDL2 definitions
```

Here is the complete sequence of operations for defining and exploiting this SDL2 vocabulary:

```
vocabulary SDL2
only FORTH also  windows also  structures also
SDL2 definitions

\ Entry point to SDL2.dll library
z" SDL2.dll" dll SDL2.dll
SDL2
```

If we run `vlist`, we should see this:

```
--> vlist
SDL2.dll
```

Here, `SDL2.dll` is the first word in our `SDL2` vocabulary.

Defining software bindings

Our `SDL2.dll` binding has become a definition word. To create the `GetError` word :

```
\ Retrieve a message about the last error that occurred on the current thread
z" SDL_GetError"            0 SDL2.dll GetError ( -- zstr )
```

Here we find:

- in blue, the name of the function as defined in the **SDL2.dll library** under Windows;
- in green the number of parameters to pass and the binding ticket;
- in red the new word created in the `SDL2 vocabulary` .

We can verify the creation of `GetError` by typing `vlist` which displays:

```
--> vlist
GetError SDL2.dll
```

Passing parameters to software bindings

In the previous example, the `GetError` word does not require a parameter. Let's see the case of a word requiring several parameters:

```
\ Set the color used for drawing operations (Rect, Line and Clear)
z" SDL_SetRenderDrawColor"  5 SDL2.dll SetRenderDrawColor
```

Here, in red, the value 5 which indicates to the binding ticket that five parameters will have to be taken into account for the word `SetRenderDrawColor`.

Here is a word that requires only one parameter:

```
\ Destroy the rendering context for a window and free associated textures
z"SDL_DestroyRenderer" 1 SDL2.dll DestroyRenderer
```

There is no indication to give on the nature and number of parameters retrieved by Forth after execution of the words defined by a binding ticket.

Understanding SDL2 Documentation

The reference documentation is available here:
https://wiki.libsdl.org/SDL2/CategoryAPI

For eForth Windows, it is not a question of creating all the words associated with these functions. On the one hand, some functions like **SDL_GetAndroidSDKVersion** do not concern the Windows environment. On the other hand, the purpose of this documentation is to give you the keys to add words if you want to use a function whose software binding has not been defined in `SDL2` .

Let's see a specific case, the **SDL_SetTextureColorMod** function, the documentation for which is available here:
https://wiki.libsdl.org/SDL2/SDL_SetTextureColorMod

The part that interests us is here:

SDL_Texture *	**texture**	the texture to update.
Uint8	**r**	the red color value multiplied into copy operations.
Uint8	**g**	the green color value multiplied into copy operations.
Uint8	**b**	the blue color value multiplied into copy operations.

Figure 9: SDL_SetTextureColorMod settings

Here we see that there are four parameters: **texture** , **r** , **g** and **b** . The only thing to remember is this number four to define our `SetTextureColorMod` word:

```
z" SDL_SetTextureColorMod"     4 SDL2.dll  SetTextureColorMod
```

And that's it!

When executed, the `SetTextureColorMod` word returns 0 or a negative value in case of error. This data will be processed when the program is executed.

Documentation for each word in the `SDL2 vocabulary` is available here:
https://eforth.arduino-forth.com/index/glossaire-windows/

Use the text search to find the complete documentation of a word in the forth or SDL2 vocabulary. Here is the detailed documentation of `GetWindowSizeInPixels` :
https://eforth.arduino-forth.com/help/explain-eforth-windows/id/1141

Initializing the SDL2 environment

The single parameter of the word **Init** in the SDL2 vocabulary allows to define the components that the library must initialize. These can be:

- **SDL_INIT_TIMER** Initializes the timer subsystem
- **SDL_INIT_AUDI** O Initializes the subsystem for audio
- **SDL_INIT_VIDE** O Initializes the subsystem for rendering
- **SDL_INIT_JOYSTICK** Initializes the subsystem for joysticks
- **SDL_INIT_HAPTIC** Initializes the subsystem for force feedback
- **SDL_INIT_GAMECONTROLLER** Initializes the subsystem for game controllers
- **SDL_INIT_EVENTS** Initializes the subsystem for events
- **SDL_INIT_EVERYTHING** Initializes all subsystems named above.

For example, to load the audio and video systems, we will need to use this line:

```
SDL2  \ selecd SDL2 vocabulary
SDL_INIT_VIDEO SDL_INIT_AUDIO or Init
```

The **Init word** can return two values:

- 0 if initialization was successful;
- a negative value if the initialization could not be done correctly.

With this option, the SDL2 library will not implement a cleanup function when receiving fatal signals.

In the code highlighted above, the return value of the function is checked in order to exit the program in the event of an error. When an error occurs, the value is different from 0. In order to inform the user (and, very often, the programmer), a clear message is displayed on the standard error output. The GetError word allows to retrieve a description of the error encountered by the SDL2 library:

```
\ Initialize SDL with error management
: SDL.init ( n -- )
    z" Could not Initialize environement " SetError drop
    Init 0= if
        -1 SDL.error
    then
  ;
SDL_INIT_VIDEO SDL_INIT_AUDIO or SDL.Init
```

Once the library is initialized, you can use it and therefore open a window.

Exit the SDL2 environment

At the end of the program, it is necessary to do the reverse operation to the initialization with the word **Quit**. It will also be necessary to do it if you encounter an error while running the program.

All manipulations must be performed between **Init**, the starting point of our program, and the word **Quit** which is its end.

The word **Quit** does not take any parameters and does not return any value, you just call it like this:

```
SDL2
SDL_INIT_VIDEO SDL_INIT_AUDIO or SDL.Init
\ ...here the main program...
Quit
```

Note: our Quit attempts leave a parameter on the stack…. No explanation!

Ressources

- **eForth for Windows**
  project created and maintained by Brad NELSON
  https://eforth.appspot.com/windows.html
- **Analyseur de code FORTH**
  transforms undocumented FORTH code into a version with syntax highlighting and hyperlinks to known words.
  https://analyzer.arduino-forth.com/

GitHub

- **SDL2 eForth windows project**
  SDL2 project codes and documentations
  https://github.com/frenchie68/Z79Forth

SDL

- **Simple DirectMedia Layer**
  SDL library development platform providing access to audio, keyboard, mouse and graphics hardware resources via OpenGL and Direct3D.
  https://www.libsdl.org/

Index