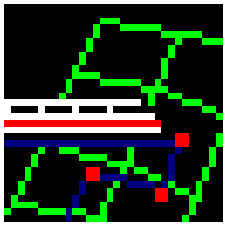
RouteBuilder SDK



Version 1.1 for RouteBuilder 1.1 and newer

Welcome to the documentation for RB's AddonInterface and the RB Software Development Kit (SDK). It allows you to develop Addons for RouteBuilder 1.1 using Delphi or Visual C++ or another Development environment capable of creating Windows DLLs.

RB currently features an addon interface version 2.0. It's quite simple to develop your own addon, but there are not so many things the addon can do with RB yet. This is mainly due to the fact that we for ourselves don't know what could make sense. If you have any idea, just tell us.

The addon interface version 2.0 is incompatible to version 1.0. Addons built with interface version 1.0 will only run in RB 1.0. Addons built with interface version 2.0 will only run in RB 1.1 and newer. We recommend you build only addons with the current interface version 2.0.

You can make your own addon if you have some basic programming knowledge and an development environment capable of producing a native windows DLL like Visual C++ or Delphi. You just need to make a DLL with certain function exports and put the DLL in the *addons* directory in the RB folder. RB will notice the DLL at startup and make it accessable in the addons menu. The DLL has basically one initialization function which is called once on RB startup. This function tells RB what the addon is, i.e. description, author name and so on. Additionally, the addon can initialize itself. The second function the DLL has to export is the worker function which is called by RB when the user runs the addon. This function should display a dialog, allow the user to do some input and to press a start button which makes the addon do for what it has been made. Then, the addon ends the worker function, closes its dialog and returns control to RB. As long as the addon has control, it can call a support function inside RB to get some information about the current project or to control RB to a certain degree (future versions of interface).

The SDK contains a Delphi unit RBAddonInterface with some basic definitions. You can easily translate it into C if you want. The SDK also contains a small example.

Your DLL has to export and implement two functions:

```
function RBAddonInit(_addonIn: PRBAddonIn; _addonOut: PRBAddonOut): boolean;
stdcall;
function RBAddonRun(): boolean; stdcall;
```

In C this would look as follows:

```
BOOL RBAddonInit(PRBAddonIn _addoIn, PRBAddonOut _addonOut);
BOOL RBAddonRun();
```

The PRBAddonIn type is a pointer to a RBAddonIn record looking as follows:

```
type
   RBAddonIn=record
     RBAddonInVersion: word;
     RBAddonFunc: RBAddonFuncType;
   end;
```

The PRBAddonOut type is a pointer to a RBAddonOut record looking as follows:

```
type
   RBAddonOut=record
     AddonName: PChar;
     AddonAuthor: PChar;
     AddonVersion: PChar;
     AddonWebsite: PChar;
     AddonEmail: PChar;
     AddonDescription: PChar;
     AddonCopyright: PChar;
   end;
```

On startup, RB calls RBAddonInit for each addon found. While _addonIn points to a structure with some information about RB, the addon has to fill the _addonOut structure before returning true. If something went wrong (i.e. RB addon interface version too small), return false, and the addon will be unloaded. Take care that the PChar pointers remain valid even when the init function of the addon has been left, so declare them as const or global.

When the user starts the addon, RBAddonRun is called.

The addon can call a support function within RB using a pointer in the _addonIn structure. Depending on the function parameter, the support function returns some information, for example the object

directory path or the project name. This will be extended in the future. The function pointer is declared as follows:

```
RBAddonFuncType= function(const what: Word ): pchar;
PRBAddonFuncType=^RBAddonFuncType;
```

Where *what* currently can be one of the following values:

```
RBAGetRBVersion = 1;
RBAGetCurrentProjectName = 2;
RBAGetCurrentProjectAuthor = 3;
RBAGetObjectLibraryPath = 4;
RBAGetRbDirectory = 5;
RBAGetProjectVariables_email = 6;
RBAGetProjectVariables_description = 7;
RBAGetProjectVariables_credits = 8;
RBAGetProjectVariables_homepage = 9;
RBAGetProjectVariables_projectfile = 10;
RBAGetProjectVariables_logo = 11;
```

The name of the constant describes what it means. Most interesting is the library path constant. By calling the addon support function with this value, you receive a pointer to a null terminated string containing the complete object library path. Your addon can for example produce a b3d object and put it into the library so it can be used immediatly after closing the addon in the RB object browser.

To to this, you should make a copy of the AddonIn structure in the init function:

```
function RBAddonInit(_addonIn: PRBAddonIn; _addonOut: PRBAddonOut): boolean;
stdcall; export;
begin
  // copy in
  AddonIn := _AddonIn^;
```

Later, call the support function as follows:

```
var libpath: string;
begin
...
libpath := AddonIn.RBAddonFunc(RBAGetObjectLibraryPath);
```

You see, it's quite easy and it works. We added the RBADummy project to the SDK so you can try it. you can use the RBADummy as basis and extend it for your needs. There is also an equivalent dummy addon project in C++. To extend its functionality, just extend the CRBAddon::Run method.

If you have questions regarding this SDK, don't hesitate to ask – just go to the bve-tools.com forum.

Have fun,

Thomas

Uwe

RB_SDK Version 2.0 (Delphi/CPP)

03-12-20 (u)