CAM API

CAM API lets you develop extensions for ENCY and connect external applications to the CAM system. This guide covers the SDK, extension development workflows and API reference.

Main information

ENCY allows third-party extensions to be connected and to take control using COM technology. These extensions can be developed by external developers and integrated into the ENCY workflow.

If you are interacting with the ENCY Open API for the first time, we recommend familiarizing yourself with Lesson #1 - completing it will allow you to create a test extension with minimal functionality, on which you can safely experiment.

Using CAMAPI, you can also write a separate application that:

- interacts with ENCY;
- controls ENCY, for example, by loading a project;
- mimics the behaviour of user, for a example, by pressing some buttons.

An example of such an application can be found in Lesson #4.

Sections

Choose a section from the navigation tree or start with one of these guides.

- Supported programming languages — SDK setup for C#, Delphi and C++.
- Tutorial lessons — examples of extensions and external applications.
- System extensions — integration through utility DLLs and executables.
- Debugging — configure your development environment to debug extensions.
- Entry points — connect extensions to ENCY menus and commands.
- Machining tools import — work with tool libraries, cutting tools and holders.
- External applications — connect to and control ENCY from a separate application.
- Using ComWrapper — work with COM objects through the helper library.
- Developer guide — concepts, macros, and C# examples for every API domain.
- API reference — namespaces, interfaces and methods, including the application API.

Helpful links:

- GitHub: CAMAPI using examples;
- Postprocessing API documentation.

System requirements:

- OS Windows 10 or higher;
- .NET 10.0 SDK or higher;

Supported programming languages

CAMAPI is based on COM technology. COM enables object use with only knowing its interface. This means, that CAMAPI objects are required to implement public interfaces. This set of interfaces forms the basis of the API SDK.

Thus, the supported programming languages are:

- C#
- C++
- Delphi

For convenience, all examples will be provided in each of these programming languages.

C# support in CAMAPI

Support factory class

After the managed DLL is loaded into the main application's memory, it is analyzed for the presence of the namespace CAMAPI with the ExtensionFactory class inside. This class must implement the IExtensionFactory. interface. The main application will interact with the loaded DLL through an object of this interface.

This interface provides method to create instances of extensions, so main application will call this method.

```
// ReSharper disable once CheckNamespace
namespace CAMAPI;

using Extensions;
using ResultStatus;

/// <summary>
/// Factory for creating extensions. Namespace and class name always should
be CAMAPI.ExtensionFactory,
/// so CAMAPI will find it
/// </summary>
public class ExtensionFactory : IExtensionFactory
{
    /// <inheritdoc />
    public void OnLibraryRegistered(IExtensionFactoryContext context, out
TResultStatus ret)
    {
        ret = default;
    }

    /// <inheritdoc />
    public void OnLibraryUnRegistered(IExtensionFactoryContext context, out
TResultStatus ret)
    {
        ret = default;
    }

    /// <summary>
    /// Create new instance of our extension
    /// </summary>
    /// <param name="extensionIdent">
    /// Unique identifier, if out library has more than one extension. Should
accord with
```

```
    /// value in settings json, describing this library
    /// </param>
    /// <param name="ret">Error to return it, because throw exception will
not work</param>
    /// <returns>Instance of out extension</returns>
    public IExtension? Create(string extensionIdent, out TResultStatus ret)
    {
        try
        {
            // write your code to create new instance
        }
        catch (Exception e)
        {
            ret.Code = TResultStatusCode.rsError;
            ret.Description = e.Message;
        }
        return null;
    }
}
```

Connecting the SDK

You have to include actual version of SDK as package reference.

```
<PackageReference Include="EncySoftware.CAMAPI.SDK.Net" Version="1.1.0" />
```

Delphi support in CAMAPI

Support export function

You native DLL should have export function, which will be entry point:

```
function CreateInstanceOfExtension(PluginID: WideString): NativeUInt;
begin
    // create instance of your class
    var res: IExtension;
    if SameText(PluginID, TExtensionUtilCommonDll.GetIdent()) then
        res := TExtensionUtilCommonDll.Create() as IExtension

    else
        raise Exception.Create('Unknown ID ' + PluginID);

    // modify as pointer
    res._AddRef;
    result := NativeUInt(res);
end;
```

Its logic can be more complex, but it is important that the result of the function is a pointer with an incremented reference count.

Connecting the SDK

The SDK is stored as package `EncySoftware.CAMAPI.SDK.bpl.x64` on nuget.org. In the BuildSystem, you can configure it to download and extract this package to a specific directory before compilation. Then, this directory needs to be connected in the Delphi project settings.

In build.cs:

- Add in list of `ManagerProps`:

```
new RestorerNugetProps
{
    Name = "RestorerNuget",
    DepsProp = new List<RestorerDepProp>
    {
        new()
        {
            PackageId = "EncySoftware.CAMAPI.SDK.bpl.x64",
            Version = "1.1.0",
            OutDir = Path.Combine(RootDirectory.Parent, "SDK")
```

```
        }
    }
}
```

- Specify accordance:

```
settings.ManagerNames.Add("restorer", "Debug", "RestorerNuget");
settings.ManagerNames.Add("restorer", "Release", "RestorerNuget");
```

- Add restore before compile in target:

```
private Target Compile => _ => _
    .Executes(() =>
    {
        _buildSpace.Projects.Restore(Variant);
        _buildSpace.Projects.Compile(Variant, true);
    });
```

In this way, any other package can be connected - similar to PackageReference in C# projects.

C++ support in CAMAPI

Support export function

You native DLL should have export function, which will be entry point:

```
extern "C" __declspec(dllexport) void* __stdcall CreateInstanceOfExtension(const
wchar_t* PluginID) {
  // create instance of your class
  IExtensionPtr resultPtr = new ExtensionUtilityExample();

  // modify as pointer
  resultPtr.AddRef();
  void* res = resultPtr.GetInterfacePtr();
  return (res);
}
```

Its logic can be more complex, but it is important that the result of the function is a pointer with an incremented reference count.

Connecting the SDK

The SDK is stored as package `EncySoftware.CAMAPI.SDK.tlb` on nuget.org. In the BuildSystem, you can configure it to download and extract this package to a specific directory before compilation. Then, this directory needs to be connected in the Delphi project settings.

In build.cs:

- Add in list of `ManagerProps`:

```
new RestorerNugetProps
{
    Name = "RestorerNuget",
    DepsProp = new List<RestorerDepProp>
    {
        new()
        {
            PackageId = "EncySoftware.CAMAPI.SDK.tlb",
            Version = "1.1.0",
            OutDir = Path.Combine(RootDirectory.Parent, "SDK")
        }
    }
}
```

- Specify accordance:

```
settings.ManagerNames.Add("restorer", "Debug", "RestorerNuget");
settings.ManagerNames.Add("restorer", "Release", "RestorerNuget");
```

- Add restore before compile in target:

```
private Target Compile => _ => _
    .Executes(() =>
    {
        _buildSpace.Projects.Restore(Variant);
        _buildSpace.Projects.Compile(Variant, true);
    });
```

In this way, any other package can be connected - similar to PackageReference in C# projects.

Tutorial lessons

The lessons are designed to simplify familiarization with CAMAPI - they are full-fledged in "how-to" style:

- The goal of each such lesson is to introduce new functionality;
- The lessons are structured to gradually increase in complexity, with each subsequent lesson building on the knowledge from the previous ones;
- Lessons can be completed in any order.

List of lessons:

- Lesson 1 - general introduction to the extension ideology using the example of a C# project;
- Lesson 2 - connecting the SDK to a Delphi project using the build system;
- Lesson 3 - connecting the SDK to a C++ project using the build system;
- Lesson 4 - demonstration of methods for unloading an extension during main application is running;
- Lesson 5 - creating a custom operation;
- Lesson 6 - creating a C# application to interact with geometry in main application;
- Lesson 7 - creating a C# application to connect to ENCY for managing it;
- Lesson 8 - creating a C# extension to integrate with a PLM system.

Additional information:

- Using ComWrapper.

The list of lessons will be expanded as the documentation grows.

Lesson #1 - general introduction to the extension ideology

Objectives

- Create the simplest and functional extension on C#;
- Show the process in broad strokes, which will be explained in detail in subsequent lessons.

Practical part

Order of actions:

- Open the repository in our GitHub account and clone it.
- Find the directory ExtensionEmptyNet - this is what you will base your new extension on.
- Pay attention to ExtensionEmptyNet/.stbuild/build/stbuild.csproj - the version of the build system is specified in this file. Over time, you might need to update it to access new features of the build system.
- Open project ExtensionEmptyNet/project/main/ExtensionEmptyNet.csproj. We will review the structure and features of the project in subsequent lessons. For now, our goal is to create the most basic extension.
- Pay atteniton to version of package EncySoftware.CAMAPI.SDK.Net - you might need to update it to access new features of API.
- Modify ExtensionTest.cs (optional). You can write your own code using the API - there are no restrictions. The example is kept as simple as possible to demonstrate the overall process of creating an extension.
- To execute your code, the DLL must contain the ExtensionFactory class. Its Create method will be called to obtain an instance of the class, and the code within that instance will then be executed.
- Modify JSON file:
  - Type of extension should be identifier of entry point;
  - Extension id should accord with value expected in ExtensionFactory.Create method;

```
{
    "name": "Empty extension on C#",
    "description": "Empty extension on C#",
    "module_path": "${extensionJsonFolder}/ExtensionEmptyNet.dll",
    "extensions": [
        {
            "utility": {
            "name": "Empty extension on C#",
                "caption": "Empty extension on C#",
                "id": "Extension.Utility.Empty.Net"
            }
```

```
        }
    ]
}
```

- Compile to check functionality.
- Run special utility to create injectable file: `ExtensionEmptyNet/commands/pack.cmd`;
- If successful, you should have a file named `ExtensionEmptyNet/project/main/bin/ExtensionEmptyNet.dext`. This file can be injected into ENCY both on this computer and on any other computer with the current version of ENCY.
- Double click on it in order to inject.
- Run ENCY, open utilities menu and press "Empty extension on C#" (name should accord with "caption" value in JSON).
- Make sure that Notepad with the project information is open.

Connecting the SDK to a Delphi project using the build system

Objectives

- Create the simplest and functional extension on Delphi;
- Connecting the SDK to a Delphi project using the build system.

Practical part

Order of actions:

- Open the repository in our GitHub account and clone it.
- Find the directory ExtensionEmptyDelphi - this is what you will base your new extension on.
- Pay attention to:
  - `ExtensionEmptyDelphi/.stbuild/build/stbuild.csproj` - the version of the build system is specified in this file. Over time, you might need to update it to access new features of the build system.
  - `ExtensionEmptyDelphi/.stbuild/build/build.cs`:
    - properties of `RestorerNugetProps`, where version of `EncySoftware.CAMAPI.SDK.bpl.x64` is specified;
    - properties of `BuilderMsDelphiProps`, where path to `MSBuild.exe` is specified (the version from Microsoft Framework is used, not the SDK);
- Modify src/Extension.pas (optional). You can write your own code using the API - there are no restrictions. The example is kept as simple as possible to demonstrate the overall process of creating an extension. Pay attention to:
  - Option `unit scope names`, because there should be mandatory `IDL` item;
- To execute your code, the project must contain the exported function CreateInstanceOfExtension. It expects the id value from ExtensionEmptyDelphi.settings.json as input and returns a reference to the created object.
- Compile to check functionality.
- Run special utility to create injectable file: `ExtensionEmptyDelphi/commands/pack.cmd`;
- If successful, you should have a file named `ExtensionEmptyDelphi/project/main/Win64/Release/ExtensionEmptyDelphi.dext`. This file can be injected into ENCY both on this computer and on any other computer with the current version of ENCY.
- Double click on it in order to inject.
- Run ENCY, open utilities menu and press "Empty extension on Delphi" (name should accord with "caption" value in JSON).
- Make sure that Notepad with the project information is open.

Connecting the SDK to a C++ project using the build system

Objectives

- Create the simplest and functional extension on C++;
- Connecting the SDK to a C++ project using the build system.

Practical part

Order of actions:

- Open the repository in our GitHub account and clone it.
- Find the directory ExtensionEmptyCpp - this is what you will base your new extension on.
- Pay attention to:
  - `ExtensionEmptyCpp/.stbuild/build/stbuild.csproj` - the version of the build system is specified in this file. Over time, you might need to update it to access new features of the build system.
  - `ExtensionEmptyCpp/.stbuild/build/build.cs`:
    - properties of `RestorerNugetProps`, where version of `EncySoftware.CAMAPI.SDK.tlb` is specified;
    - properties of `BuilderMsCppProps`, where path to `MSBuild.exe` is specified (the version from Microsoft Visual Studio is used);
  - `ExtensionEmptyCpp\project\main\ExtensionEmptyCpp.settings.json` - you can adjust the settings of the displayed menu item - caption, hint and path to icon (relative to dll file);
- Modify main.cpp (optional). You can write your own code using the API - there are no restrictions. The example is kept as simple as possible to demonstrate the overall process of creating an extension.
- To execute your code, the project must contain the exported function CreateInstanceOfExtension. It expects the id value from ExtensionEmptyCpp.settings.json as input and returns a reference to the created object.
- Compile to check functionality.
- Run special utility to create injectable file: `ExtensionEmptyCpp/commands/pack.cmd`;
- If successful, you should have a file named `ExtensionEmptyCpp/project/main/x64/Release/ExtensionEmptyCpp.dext`. This file can be injected into ENCY both on this computer and on any other computer with the current version of ENCY.
- Double click on it in order to inject.
- Run ENCY, open utilities menu and press "Empty extension on C++" (name should accord with "caption" value in JSON).
- Make sure that Notepad with the project information is open.

Demonstration of methods for unloading an extension during main application is running

Objectives

- Create the simplest and functional extension on C#;
- Use various methods to unload extensions.

Practical part

Order of actions:

- Open ENCY because we will be editing the list of utilities while the application is running.
- Open the repository in our GitHub account and clone it.
- Find the directory `GCodeGeneration\ExtensionUtilityNCMakerNet` - this is what you will base your new extension on.
- The extension is already correct. In this lesson, we will not modify it but will only change the configuration file.
- Open .NET project and compile it - `bin/debug/ExtensionUtilityNcMakerNet.dll` should appear;
- Copy `ExtensionUtilityNcMakerNet.settings.json` into `bin/debug/`;
- Add extension in the first time:
  - Archive both files (`dll` and `settings.json`) into file with `*.dext` extension;
  - Execute created `*.dext` file - it will be added to ENCY. It has no `unload_mode` setting, so it will be set to default `umJustAfterExecution`;
- In ENCY open utilities menu and press `Generate G code by C#`. Then close notepad window with results.
- Don't close ENCY and add extension in the second time:
  - Modify `bin/debug/ExtensionUtilityNcMakerNet.settings.json`:
    - Add unload mode `umManuallyFromUI`;
    - Change caption;

```
{
  "name": "Extension on C# to generate G code",
  "description": "Extension on C# to generate G code",
  "module_path": "${extensionJsonFolder}/ExtensionUtilityNcMakerNet.dll",
  "unload_mode": "umManuallyFromUI",
  "extensions": [
    {
      "utility": {
        "name": "Generate G code by C#",
        "caption": "Generate G code by C# (version 2)",
        "id": "Extension.Utility.NcMaker.Net"
      }
```

```
    }
  ]
}
```

- Archive json and dll files into file with `*.dext` extension;
- Execute created `*.dext` file - it will be added to main application. Library will be overwritten, because it was unloaded. Otherwise, you will encounter an error.
- In ENCY open utilities menu - there should be item, called `Generate G code by C# (version 2)`. This means that with the previous action, you successfully updated the extension - now it has new caption.
- Press `Generate G code by C# (version 2)` and close notepad with opened results.
- Now try to execute early created `*.dext` extension. You'll get an error `Access to the path 'ExtensionUtilityNcMakerNet.dll' is denied`. This means that the extension was not unloaded from memory - as it should be, since we specified the unload method as `umManuallyFromUI`.
- But we still can unload it from UI:
  - Open `Settings` -> `Extensions`;
  - Choose `Generate G code by C#` (here is value from `name` field in `settings.json`);
  - Click `unload` - if the library unloads successfully, the `unload` button will become inactive, otherwise, you will encounter an error.
- Now execute early created `*.dext` extension - you will succeed, because extension was manually unloaded from memory;

- Don't close ENCY and add extension in the third time:
  - Modify `bin/debug/ExtensionUtilityNcMakerNet.settings.json`:
    - Add unload mode `umBeforeProgramFinalization`;
    - Change caption;

```
{
  "name": "Extension on C# to generate G code",
  "description": "Extension on C# to generate G code",
  "module_path": "${extensionJsonFolder}/ExtensionUtilityNcMakerNet.dll",
  "unload_mode": "umBeforeProgramFinalization",
  "extensions": [
    {
      "utility": {
        "name": "Generate G code by C#",
        "caption": "Generate G code by C# (version 3)",
        "id": "Extension.Utility.NcMaker.Net"
      }
    }
  ]
}
```

    - Archive json and dll files into file with `*.dext` extension;
    - Execute created `*.dext` file - it will be added to main application.
    - In ENCY open utilities menu - there should be item, called `Generate G code by C# (version 3)`. This means that with the previous action, you successfully updated the extension - now it has new caption.
    - Press `Generate G code by C# (version 2)` and close notepad with opened results.
    - Try to execute early created `*.dext` extension. You'll get an error `Access to the path 'ExtensionUtilityNcMakerNet.dll' is denied`.
    - Check unloading manually from UI:
        - Open `Settings` -> `Extensions`;
        - Choose `Generate G code by C#` (here is value from `name` field in `settings.json`);
        - Ensure that the `unload` button is inactive. You can't unload it, because of `umBeforeProgramFinalization`.
- Now you can close ENCY and execute created `*.dext` - the library was unloaded from memory when ENCY closed.

Creating a custom operation

Objectives

- Create a new operation type with settings.

Practical part

Order of actions:

- Open the repository in our GitHub account and clone it.
- Find the directory `Operation\ExtensionOperationSimpleNet` - this is what you will base your new extension on.
- Pay attention to:
  - XML-file `project\main\OperationSimpleNet.xml` - this file contains the operation settings:
    - Settings that allow ENCY to understand that the tool path calculation is handled by an external extension;
    - Operation settings that the user can configure in the UI;
    - `ContainerID` cannot be changed;
    - `SolverID` contains value, which will be later provided into `ExtensionFactory.Create`;
  - `ExtensionFactory.cs`:
    - Methods `OnLibraryRegistered` and `OnLibraryUnRegistered` - they edit the global `UserOperationsList.xml`, which contains custom operations.
  - The tool path calculation is performed in a subthread, and .NET only allows working with COM objects created in the same thread. Therefore, you should avoid declaring the properties of `ExtensionOperationSimpleNet` in `RunInitialize`.
- Modify main.cpp (optional). You can write your own code using the API - there are no restrictions.
- Call task 'Pack', which will create `*.dext` file - execute it. This action must be performed the first time - only then will the global `UserOperationsList.xml` be edited. It will add the path to `bin\Release\OperationSimpleNet.xml`.
- Restart ENCY to update list of available operation types;
- Click to add new operation. It should be located in the group specified in the XML file under the `MultiGroup` node. If it's not there, check with `unavailable operations` specified.
- Click `generate toolpath` on the new operation.

Creating a C# application to interact with geometry in main application

Objectives

- Create a separate application in C# which works with geometry model.

Practical part

- Open the repository in our GitHub account and clone it.
- Find the directory Geometry. This folder contains two examples, one of them shows how to work with importer and the second shows how to use geometry model.
- Open project ExtensionUtilityGeometryModelNet\project\main\ExtensionUtilityGeometryModelNet.csproj.
- Build the project.
- Run ENCY separately.
- Open `Setting`, go to the tab `Extensions`.
- Press `Install`, select the file ExtensionUtilityGeometryModelNet\project\main\ExtensionUtilityGeometryModelNet.settings.json.
- Return to the main window and press `Import`, select the model Models\Milling_3D\49-1.igs from examples.
- After the model import is completed, find the utilities button in the upper left corner and run one of utilities:
  - `Export geometry node` - this utility exports current geometry model to .osd file;
  - `Transform geometry node` - this utility transforms current geometry model;
  - `Select colored geometry nodes` - this utility changes color of nodes and selects then in the main window.

Creating a C# application to connect to ENCY for managing it

Objectives

- Create a separate application in C# to interact with ENCY.

Practical part

- Open the repository in our GitHub account and clone it.
- Find the directory ApplicationEmptyNet - this is what you will base your new application on.
- Open project ApplicationEmptyNet/project/main/ApplicationEmptyNet.csproj.
- Pay attention to:
  - `CamHelper` singleton, which connects to any found ENCY instance;
- Modify MainWindow.xaml.cs (optional). You can write your own code using the API - there are no restrictions. The example is kept as simple as possible to demonstrate the overall process of creating a separate application.
- Run example;
- Run ENCY separately;
- Press `Open project`, choose any project from examples;
- Ensure that the project you selected has opened in ENCY.

Creating a C# extension to integrate with a PLM system

Objectives

- Understand the basic structure of a PLM integration extension;
- Create an extention in C# or modify an existing example to integrate with a PLM system.

Practical part

To create an extension, you need to:

- Install the latest version of the NuGet package that provides the base interfaces:
  - `EncySoftware.CAMAPI.SDK.Net`.
- Implement two interfaces:
  - `IExtension` – is a common interface for extensions so the CAM application can load your extension. It includes a single property `IExtensionInfo Info` which contains basic metadata;
  - `IExtensionPLM` – provides methods for integration with the PLM system (e.g., connect, send data, fetch data).
- Start with the required methods:
  - `GetParameters` – returns an IPLMParameters object. The CAM application uses it to display extension settings to the user;
  - `Connect` – receives parameter values and can establish a connection with the PLM system;
  - `Disconnect` – closes the connection and clears any internal state if needed.
- Then, implement the core item-related methods:
  - Retrieving items:
    - `GetChilds` – retrieves child items for a given parent ID. If no parent ID is provided, it returns items from the root level;
    - `FindItems` – searches items by name;
    - `GetItem` – loads a specific item by ID;
    - `GetLinkedItem` – returns a related item;
    - `GetItemData` – loads detailed data for a selected item.
  - Working with items:
    - `DownloadItems` – downloads selected items (except projects);
    - `UploadItem` – uploads regular items (except projects);
    - `DownloadProject` / `UploadProject` – for project-specific operations.
- Implement lifecycle methods:
  - `Install` and `Uninstall` – methods called when the extension is registered or removed. They can be left empty if you don't need setup or cleanup logic.
- These boolean properties let the CAM application know which functions your extension supports: `SupportProjectLoad`, `SupportMachineLoad`, `SupportPostprocessorLoad`,

`SupportPostprocessorInsideMachineLoad`, `SupportToolLoad`, `SupportToolSave`, `SupportDomainAuth`. If a property returns false, related functionality will be disabled in the app UI.

You can find the full method list and descriptions here.

You can clone the example project and either use it as-is or modify it to create your own extension:

- Open the repository in our GitHub account that contains the example extensions and clone it.
- Within the repository, find the folder that contains the PLM integration example - this is what you will base your new extension on.
- Open the project `PLMExtension/PLMExtensionNet/project/main/PLMExtensionNet.csproj` in your preferred IDE or code editor.
- Pay attention to:
  - `PLMExtension/PLMExtensionNet/.stbuild/build/stbuild.csproj` - the version of the build system is specified in this file. Over time, you might need to update it to access new features of the build system;
  - the version of the NuGet package `EncySoftware.CAMAPI.SDK.Net`.
- Modify the example for your PLM system. The example already includes a basic working implementation of both interfaces and uses directories to simulate operations of PLM system;

Once your extension is ready:

- Build the solution;
- Run ENCY separately;
- Setup the extension in ENCY and set configuration for PLM connection, further information is available here;
- Ensure that all functions of the extension operate correctly within ENCY.

System extensions that are included in CAMAPI

CAMAPI includes a basic set of system extensions that ensure its functionality. System extensions are connected in the same way as external ones. The presence of such extensions indicates that they can even be replaced if necessary.

Existing system extensions:

- Extension.Util.Common.Dll.

System extension Extension.Util.Common.Dll

The extension is designed for loading DLL files. It can load both native DLLs (written in C++ and Delphi) and managed DLLs (written in C#).

- Entry point - utility_runner;
- Implemented interface - IExtensionUtilityRunner.

The DLL is analyzed to determine whether it is managed. If it is native, the method CreateInstanceOfExtension is called with the GUID specified in the utility settings (in the configurator), and it is expected to return an object of the IExtensionUtility interface.

If the called DLL is managed, it is analyzed to find the CAMAPI namespace. If this namespace is found, it should contain a ExtensionFactory class that implements the IExtensionFactory interface. An instance of this class will be created. Then, the IExtensionFactory.Create method will be called with the GUID, as there may be multiple utilities within a single DLL.

System extension Extension.Util.Common.Exe

The extension is designed for running EXE files. It launches the exe file in a separate process to avoid interrupting the operation of ENCY. Parameters can be passed if they are specified in the utility settings..

- Entry point - utility_runner;
- Implemented interface - IExtensionUtilityRunner.

Debugging of extensions

This section of the documentation will demonstrate the capabilities to debug your own extensions on the installed ENCY:

- Debugging a .NET extension using Visual Studio Code;
- Debugging a Delphi extension using RAD Studio;
- Debugging a C++ extension using Visual Studio.

Debugging a .NET extension using Visual Studio Code

Debugging in Visual Studio Code is configured using `launch.json`, where the executable program and its launch parameters are specified. This `launch.json` is already present in the examples from our GitHub repository:

- `program`: "C:/Program Files/ENCY Software/ENCY/Bin64/ency.exe";
- `args`: ["/EXTENSION_DEBUG_FOLDER="${workspaceFolder}/project/main/bin/Debug""].

This means that selecting `Run` -> `Start Debugging` will lead to:

- Visual Studio Code executes preLaunchTask:
  - Compiles your project, placing the DLL in the path specified in the .csproj file (`<OutDir>bin\$(Configuration)\</OutDir>`);
  - Copies `settings.json` to the same folder;
- C:/Program Files/ENCY Software/ENCY/Bin64/ency.exe is launched with the parameter `/EXTENSION_DEBUG_FOLDER="${workspaceFolder}/project/main/bin/Debug"`;
- ENCY, in addition to the extensions already registered, analyzes the directory `${workspaceFolder}/project/main/bin/Debug`, loading just compiled `dll` file according to the `settings.json` file;
- Your breakpoints in Visual Studio Code will be hit as your extension is used in the ENCY application.

Debugging a Delphi extension using RAD Studio

For debugging the extension in RAD Studio you should:

- In the project properties - `Options` -> `Debugger`:
  - Set `Host application`: `C:/Program Files/ENCY Software/ENCY/Bin64/ency.exe`;
  - Set `Parameters`: `/EXTENSION_DEBUG_FOLDER=\"${workspaceFolder}/project/main/bin/Debug`. Instead of `${workspaceFolder}` specify the full path to the directory with an example.
- Copy the `settings.json` file to the `bin\debug` directory.

This means that selecting `Run (F9)`:

- RAD Studio Compiles your project, placing the DLL in the path specified in the project settings;
- C:/Program Files/ENCY Software/ENCY/Bin64/ency.exe is launched with the parameter `/EXTENSION_DEBUG_FOLDER="${workspaceFolder}/project/main/bin/Debug"`;
- ENCY, in addition to the extensions already registered, analyzes the directory `${workspaceFolder}/project/main/bin/Debug`, loading just compiled `dll` file according to the `settings.json` file;
- Your breakpoints in RAD Studio will be hit as your extension is used in the ENCY application.

Debugging a C++ extension using Visual Studio

For debugging the extension in Visual Studio you should:

- In the project properties - `Options` -> `Debugger`:
  - Set `Command`: `C:/Program Files/ENCY Software/ENCY/Bin64/ency.exe`;
  - Set `Parameters`: `/EXTENSION_DEBUG_FOLDER=\"${workspaceFolder}/project/main/bin/Debug`. Instead of `${workspaceFolder}` specify the full path to the directory with an example.
- Copy the `settings.json` file to the `bin\debug` directory.

This means that selecting `Start debugging (F5)`:

- Visual Studio your project, placing the DLL in the path specified in the project settings;
- C:/Program Files/ENCY Software/ENCY/Bin64/ency.exe is launched with the parameter `/EXTENSION_DEBUG_FOLDER="${workspaceFolder}/project/main/bin/Debug"`;
- ENCY, in addition to the extensions already registered, analyzes the directory `${workspaceFolder}/project/main/bin/Debug`, loading just compiled `dll` file according to the `settings.json` file;
- Your breakpoints in Visual Studio will be hit as your extension is used in the ENCY application.

Entry points in CAMAPI

CAMAPI includes various entry points. An entry point is a location within the system where extensions created by third-party developers can be integrated. These points allow the functionality of the system to be extended by adding new capabilities and enhancing existing ones. This opens the door for the creation of a wide range of extensions that can meet specific user needs. Thus, CAMAPI fosters the development of an ecosystem of applications and tools, supporting innovation and flexibility in usage.

Available entry points:

- Utilitiy in main form;
- Executor for utility in main form;
- New item to the operation's context menu.

Entry point in adding new utility

At present, CAMAPI utilities has 2 entry points. This is sufficient to add any file to the list of utilities and configure its execution. This could be a DLL file, an executable EXE file, a script whose commands can only be processed by your other application.

The first entry point allows you to extend the list of utilities. Yes, this duplicates the functionality of manually adding utilities through the configuration window. However, extensions can be transferred from computer to computer via a dext file, so you can provide the user with just the executable file instead of writing instructions on how to configure a new utility. This becomes especially useful as the number of utilities begins to grow.

The use of this entry point can be complemented by the use of the second entry point – this will allow utilities of any format to be connected.

- Identifier: `utility`;
- Used interfaces:
  - IExtensionTypeInfoUtility.
  - IExtensionUtility.
- Available configuration settings:
  - caption: the text that will be shown to the user;
  - hint_text: the text that will be displayed as a tooltip when hovering the mouse cursor;
  - loader_extension_ident: the identifier of another extension that will invoke the current extension for execution (non-mandatory setting - in the case, where it is not set, will be used standard Extension.Util.Common.Dll)

Examples in our GitHub account:

- C# - directory `ExtensionUtilityNet`;
- Delphi - directory `ExtensionUtilityDelphi`;
- C++ - directory `ExtensionUtilityCpp`.

IMPORTANT: The code in the examples demonstrates how main application will display an error that occurs in the extension code. There is an error in the examples – ENCY must be run with administrator rights to save the exported.stcp file.

Entry point in executing utility

This entry point already contains a list of system extensions:

- Loading DLL files;
- Loading EXE files.

The extension expands the list of utilities that can be connected to main application. The fact is that system extensions can only run DLL and EXE files. By adding a new extension of type `utility_runner`, you can specify it in the launch parameters for extensions of type utility. Thus, ENCY calls the `utility_runner` extension, passing it information about the utility to be launched.

- Identifier: `utility_runner`;
- Used interfaces:
  - IExtensionTypeInfoUtilityRunner.
  - IExtensionUtilityRunner.
- Available configuration settings:
  - filter_types: filter provided to "Choose file" dialog window, when user chooses utility;
  - visible_in_editor: extension is visible in utility editor, so it can be chosen. This allows for the creation of a unique extension that is suitable for launching only a specific utility and should not be displayed elsewhere to the user.

Examples in our GitHub account:

- C# - directory `ExtensionUtilityRunnerNet`;
- Delphi - directory `ExtensionUtilityRunnerDelphi`;
- C++ - directory `ExtensionUtilityRunnerCpp`.

Entry point in adding a new item to the operation's context menu

The entry point allows for the configuration of main program, enabling you to add custom items to the context menu based on the selected operation. For each added item, you can define its own executable procedure. This way, a single extension can add an unlimited amount of functionality.

- Identifier: `operation_popup`;
- Used interfaces:
  - IExtensionTypeInfoOperationPopup.
  - IExtensionOperationPopup.
- Configuration settings: none.

Examples in our GitHub account:

- C# - directory `ExtensionOperationPopupNet`;
- Delphi - directory `ExtensionOperationPopupDelphi`;
- C++ - directory `ExtensionOperationPopupCpp`.

Machining Tools Import API

This section describes the CAMAPI methods that allow you to import machining tools into the application tool libraries. This api allows you to create turning and milling tools, custom axial tools, cutting tool libraries. Before using this api, you need to make sure that the following dll libraries are available:

- Bin64\MachiningToolsImport.dll;
- Bin64\MachinigToolsImportTypes.dll;
- Bin64\STMachiningToolsImportHelper.dll.

For a complete understanding the principle of forming cutting tools, creting a new tool library or open an existing one, it is recommended to study the documentation on working with Machining tool features.

System requirements:

- OS Windows 10 or higher;
- .NET 10.0 SDK or higher;

Helpful links:

- GitHub: Machining tool import API examples;

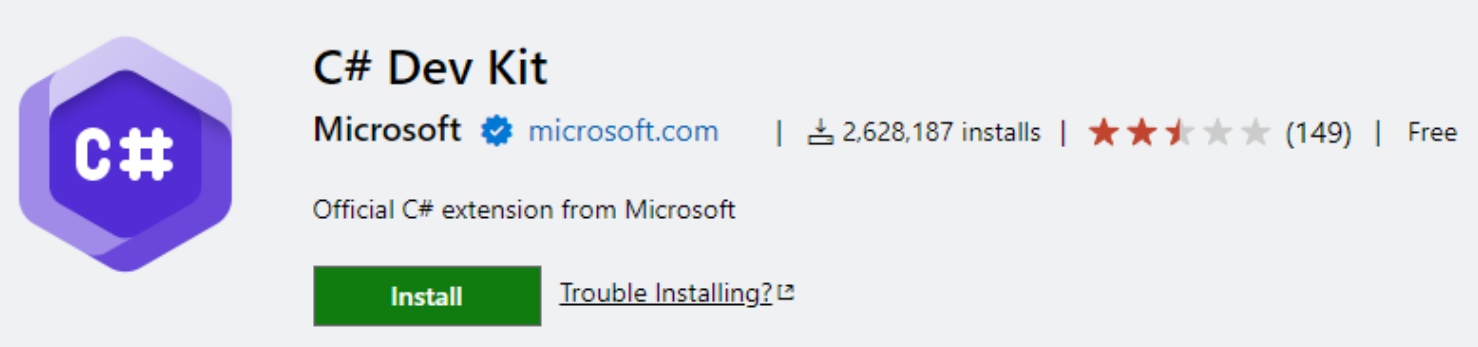
Continue to read:

- Preparing the machining tools import api environment

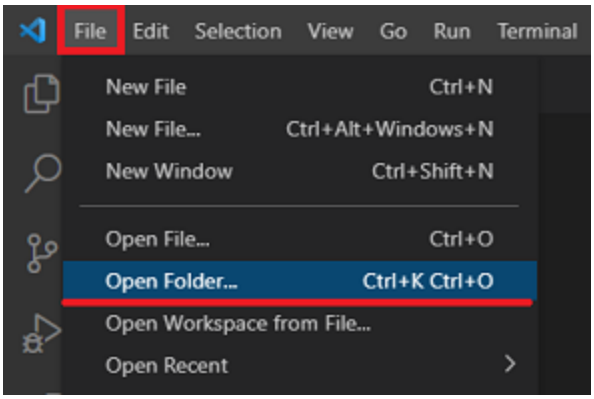
Preparing the environment

The following is an example of how to prepare environment to work with machining tools import API, using the example of creating a console application in Visual Studio in C#.

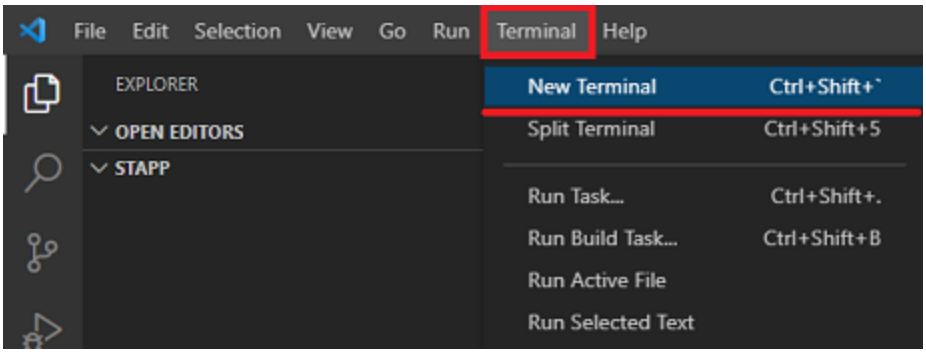
1. Download and install the latest .NET SDK x64 from the official site.
2. To work with the source files, download and install the latest available version of Visual Studio Code.
3. Run the installed Visual Studio Code and install "C# Dev Kit for Visual Studio Code" extension.



4. If you have not yet, download and install the latest ENCY.
5. In Visual Studio Code, open an existing or create a new directory for your new project. When prompted to "Do you trust the authors of the files in this folder?", click on "Yes, I trust the authors".



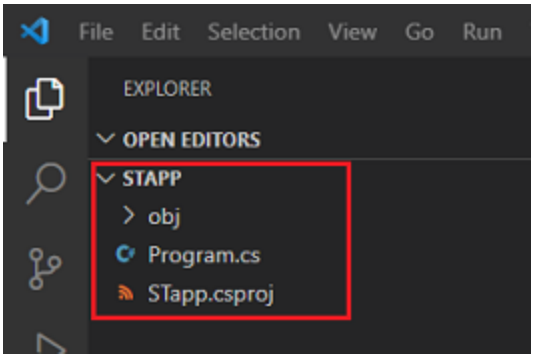
6. Open the VS Code terminal view.



7. In the terminal view insert the "dotnet new console --framework net10.0" command and press enter (change the net10.0 version if you have a different SDK version).



- After executing this command, the following source files should appear in your working directory:



- The .csproj file will have the same name as the working folder.

8. Open the .csproj source file and change its contents as shown below:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net10.0-windows</TargetFramework>
    <OutputType>Exe</OutputType>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="EncySoftware.CAMAPI.SDK.Net" Version="3.*" />
  </ItemGroup>
</Project>
```

- Set existing DLL location paths according to your CAM environment. If you cannot find the DLL libraries, make sure the CAM client is installed.
- Pay attention to the target framework of your solution, it must match the format net10.0-windows (or higher SDK version).

9. Open the Program.cs source file and change its contents as shown below:

```
using System;
using System.IO;
using MachinigToolsImportTypes;
using STMachiningToolsImportHelper;
using STTypes;
```

```
namespace STConsoleApp
{
  class Program
  {
    private static string assemblyPath =
      @"C:\Program Files\[CAM_Path]\" + MTIMachiningToolsImportHelper.DllName;

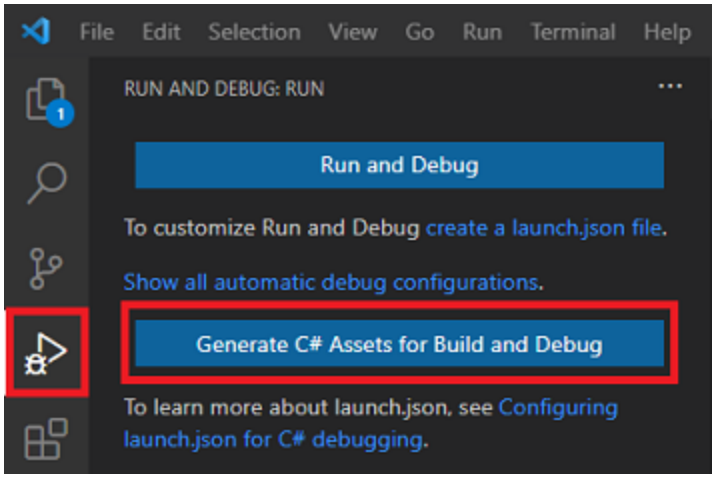
    static void Main(string[] args)
    {
      if (!File.Exists(assemblyPath)) {
        Console.WriteLine("The {0} source file is not found", assemblyPath);
        return;
      }

      try {
        var importer = MTIMachiningToolsImportHelper.CreateImporter(assemblyPath);
        //Use "importer" to create new tool items, to open an existing or create new
machining tools libraries

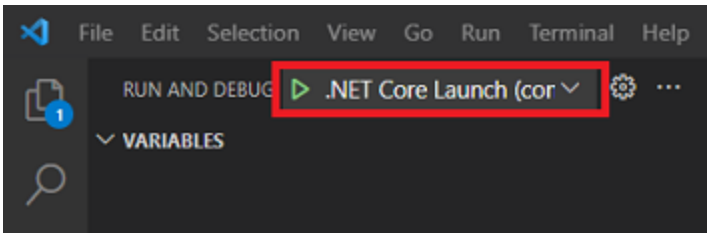
      }
      catch (Exception ex) {
        Console.WriteLine(ex.Message);
      }
      finally {
        MTIMachiningToolsImportHelper.FinalizeImporter();
      }
    }
  }
}
```

Change the "assemblyPath" variable to existing the "Bin64" folder location path according to your CAM environment. Now you can use the "importer" instance to create new tool items, to open an existing or create new machining tools libraries, and more. Pay attention to the call of the FinalizeImporter function, this function is always called before exiting the main program.

10. Switch to the "Run and Debug" panel and press to the "Generate C# Assets for Build and Debug" button.



- Now you can start debugging your application:
-



If you have set all the paths correctly, then this example should work without an error.

⭐ Congratulations, you have set up your development environment. ⭐

Helpful links:

- GitHub: Machining tool import API examples;

Continue to read:

- Working with the tool library
- Working with cutting tools

Working with the tool library

Machining tool libraries are created in .db files, where each library is an independent repository of a specific set of tools. By default, the standard machining tool libraries are located in the "C:\ProgramData[CAM_Path]\Version 17\Libraries\Tools" directory. Custom tool libraries can be created in any location. To create a new tool library or open an existing one, use the functions of the IMTI_MachiningToolsImportLibrary instance (importer):

```
//C#
//create a new tool storage
var tNewStorage = importer.CreateNewToolsStorage(@"D:\teststorage.db");
//open an existing tool storage
var tExstStorage = importer.OpenExistingToolsStorage(@"D:\teststorage.db");
```

To add a new machining tool item to the tool library, use the following function:

```
//C#
//the tool must be created and formed before adding it to the tool library
bool result = storage.AddToolItem(tool);
```

Helpful links:

- How to manage tool libraries in the CAM interface
- Preparing the machining tools import api environment
- GitHub: Machining tool import API examples;

Continue to read:

- Working with cutting tools

Working with cutting tools

Three types of cutting tools are available: milling tools, turning tools and custom axial shaped tools. To create cutting tools using the CAMAPI, use the IMTI_MachiningTools instance functions:

- Milling tools create functions:
  - CreateCylindricalMill
  - CreateSpphericalMill
  - CreateTorusMill
  - CreateDoubleRadialMill
  - CreateLimitedDoubleRadialMill
  - CreateConicalMill
  - CreateLimitedConicalMill
  - CreateEngraver
  - CreateDrill
  - CreateTwoStageDrill
  - CreateSpotDrill
  - CreateCenterDrill
  - CreateReamer
  - CreateCounterSinkDrill
  - CreateCounterBoreDrill
  - CreateBoringBar
  - CreateTap
  - CreateNegativeRadiusMill
  - CreateKnife
  - CreateSawBlade
  - CreateJetCutter
  - CreateFlatTipCountersink
  - CreatePilotCounterbore
  - CreateBackBoringBar
  - CreateThreadMill
  - CreateUndercutDoveMill
  - CreateUndercutSlotMill
  - CreateUndercutLollipopMill
  - CreateUndercutTwoAngleMill
  - CreateUndercutRoundGrooveMill
  - CreateUndercutSharpChamferMill
  - CreateUndercutRoundedChamferMill
  - CreateUndercutIndexableChamferMill
  - CreateUndercutBarrelMill
  - CreateUndercutLensBarrelMill

    - CreateUndercutTaperBarrelMill
    - CreateTaperJetCutter
    - CreatePlazmaJetCutter
    - CreateLaserJetCutter
    - CreateEmptyGripper
    - CreateSingleVacuumGripper
    - CreateFullConeSprayGun
    - CreateHollowConeSprayGun
    - CreateElipticFlatFanSprayGun
    - CreateElipticLinearFanSprayGun
    - CreateRectangularFlatFanSprayGun
    - CreateSphericalProbe
- Turning tools create functions:
    - CreateTurnToolWithExternalHolder
    - CreateTurnToolWithBoringHolder
    - CreateTurnToolWithExternalThreadingHolder
    - CreateTurnToolWithInternalThreadingHolder
    - CreateTurnToolWithExternalGroovingHolder
    - CreateTurnToolWithInternalGroovingHolder
    - CreateTurnToolWithFaceGroovingHolder
    - CreateTurnToolWithProbingHolder
- Custom axial shape tool functions:
    - CreateCustomAxialShapeTool.

Helpful links:

- How to work with cutting tools in the CAM interface
- Preparing the machining tools import api environment
- GitHub: Machining tool import API examples;

Continue to read:

- Working with Milling Tools
- Working with Turning Tools
- Working with Custom Axial Shaped Tools
- Working with the Tool Holder

Working with Milling Tools

Milling tools are divided into various subtypes, where the geometry of each subtype is set by its own unique set of parameters. Common tool attributes, such as tool name, tool number, etc., are the same for all subtypes of cutting tools. For example, below is the code for creating a Cylindrical Mill tool:

```
//C#
//importer is an instance of IMTI_MachiningToolsImportLibrary
var tool = importer.CreateCylindricalMill();
tool.OverallLength = 100;
tool.CuttingDiameter = 10;
tool.WorkingLength = 30;
tool.ShankDiameter = 12;
tool.ShoulderLength = 60;
tool.ShankTaperAngle = 40;

tool.SetName("My Cylindrical mill");
tool.SetIdentifier("m001");
tool.SetTeethsCount(3);
tool.SetMagazineNumber(1);
tool.SetToolNumber(1);
tool.SetUnits(TMTI_LinearUnits.luMillimeter);
tool.SetOverhang(100);
tool.SetDurability(70);
tool.SetMaxPlungeAngle(90);

var toolings = tool.GetToolingPoints();
toolings.ToolContactPointType = TMTI_AxialToolContactPointType.cpCustomPoint;
toolings.ToolContactPointsCount = 1;

toolings.ToolingPointCount = 1;
toolings.ToolingPointType[0] = TMTI_AxialToolToolingPointType.tpCustomPoint;
toolings.ToolingPointShift[0] = 1.5;
toolings.ToolingPointLengthCorrectorNumber[0] = 1;
toolings.ToolingPointRadiusCorrectorNumber[0] = 1;

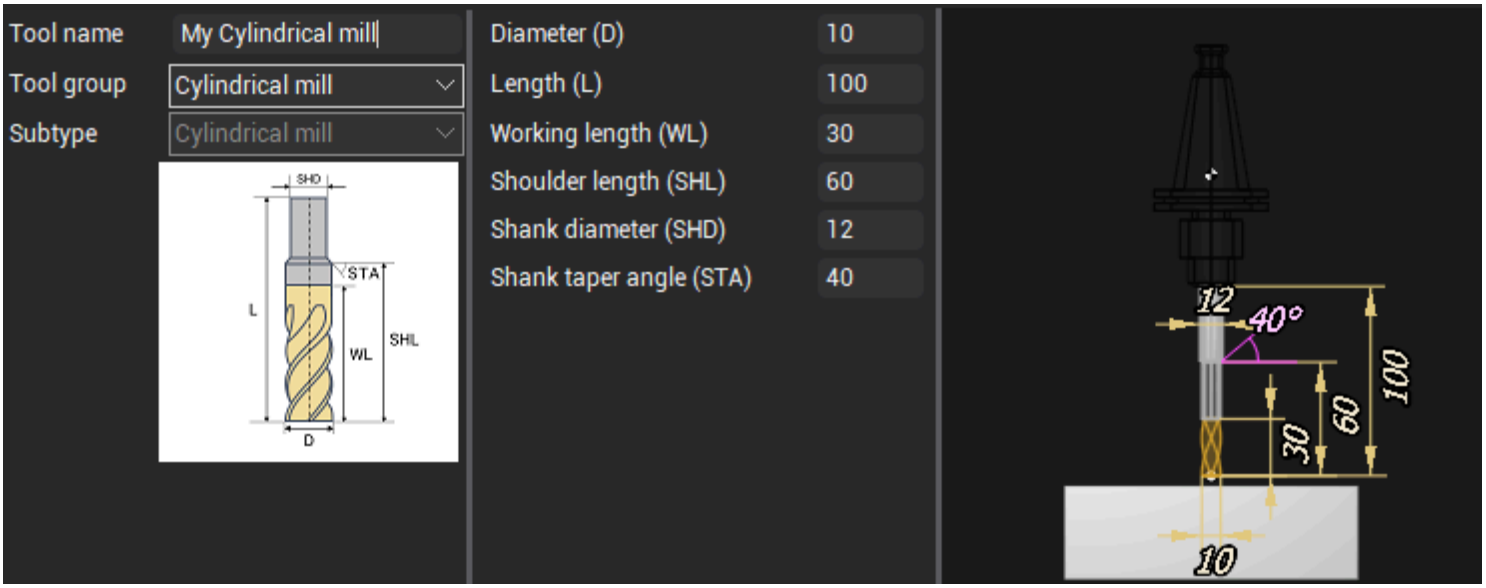
var cond = tool.GetCuttingConditions();
cond.CuttingSpeedMode = TMTI_CuttingSpeedMode.csmRPM;
cond.RotationsPerMinute = 100;
cond.SpindleGearRange = 1;
cond.RotationDirection = TMTI_RotationDirection.rdCW;
cond.FeedUnits = TMTI_FeedUnits.perMinute;
cond.FeedValue = 100;

int tubesCount = cond.Coolant.TubeCount; //20
```

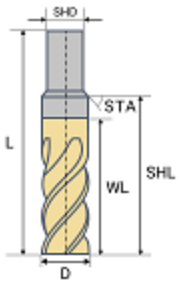
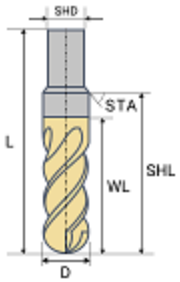
```
cond.Coolant.TubeIsOn[0] = true;
cond.Coolant.TubeIsOn[1] = false;
cond.Coolant.TubeIsOn[2] = true;

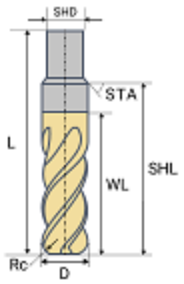
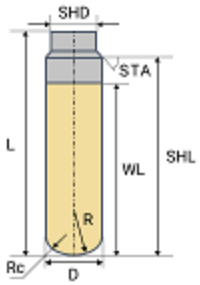
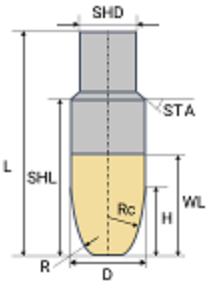
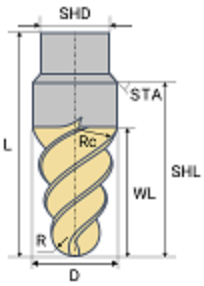
//after setting the properties, add this tool to your tool library instance
```

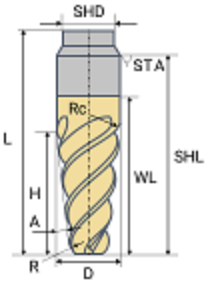
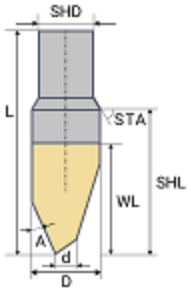
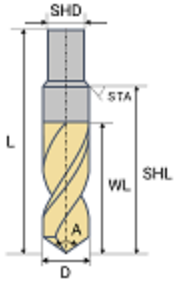
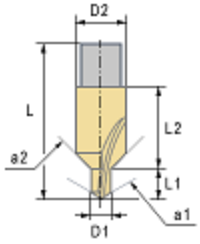
Code execution result:

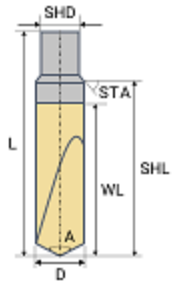
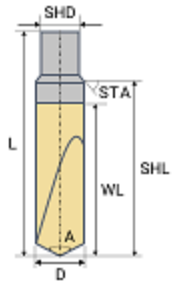
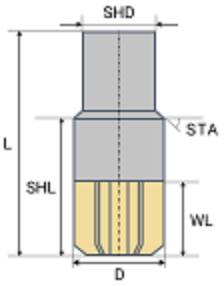
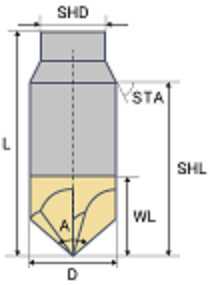


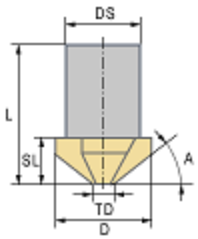
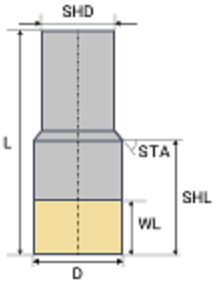
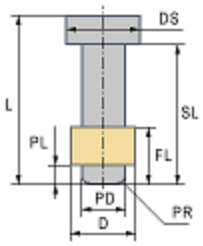
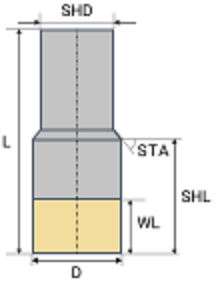
Milling Tool Geometry Properties

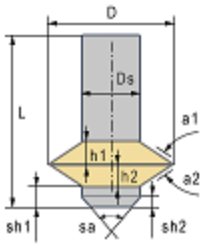
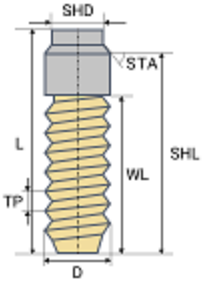
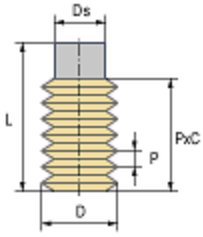
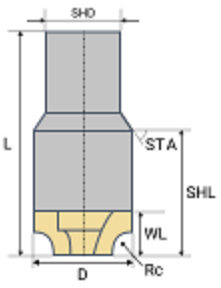
Tool type	View
Cylindrical Mill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA)	
Spherical Mill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA)	

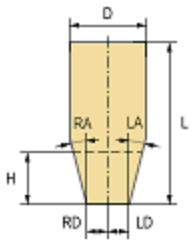
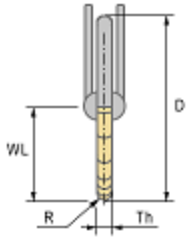
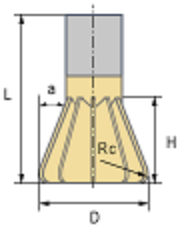
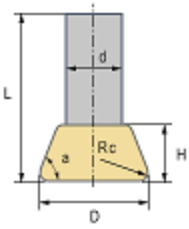
Tool type	View
Torus Mill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Radius (Rc)	
Double Radial Mill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Radius (Rc) TipRadius (R)	
Limited Double Radial Mill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Radius (Rc) TipRadius (R) ShapeLength (H)	
Conical Mill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Radius (Rc)	

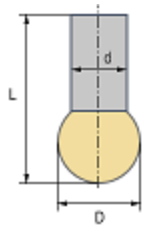
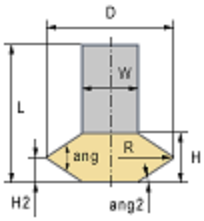
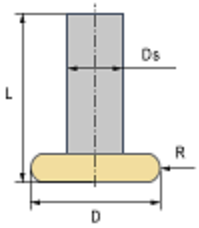
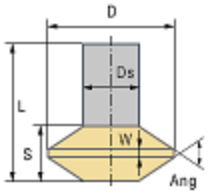
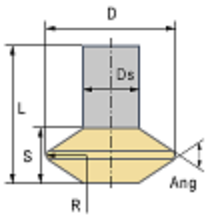
Tool type	View
TipRadius (R) Angle (A)	
Limited Conical Mill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Radius (Rc) TipRadius (R) Angle (A) ShapeLength (H)	
Engraver: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Angle (A) EndDiameter (d)	
Drill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Angle (A)	
Two Stage Drill: OverallLength (L) WorkingLength (WL) Diameter1 (D1) Diameter2 (D2) Length1 (L1)	

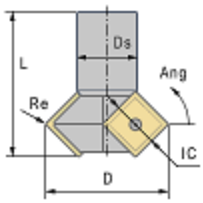
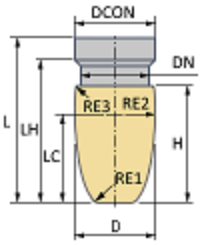
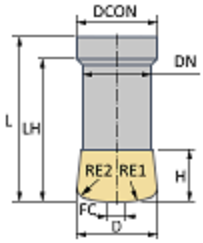
Tool type	View
Length2 (L2) Angle1 (a1) Angle2 (a2)	
Spot Drill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Angle (A)	
Center Drill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Angle (A)	
Reamer: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA)	
Counter Sink Drill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Angle (A)	

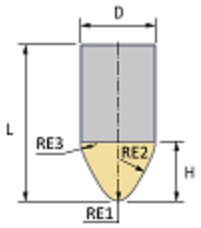
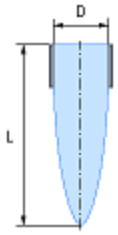
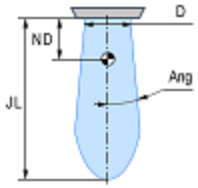
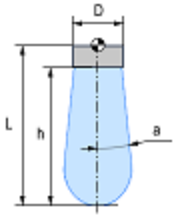
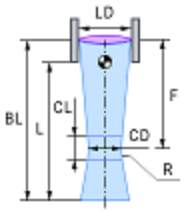
Tool type	View
Flat Tip Countersink: OverallLength (L) Diameter (D) WorkingLength (WL) ShaftDiameter (DS) TipDiameter (TD) ShoulderLength (SL) TipAngle (A)	
Counter Bore Drill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Angle (A)	
Pilot Counterbore: OverallLength (L) Diameter (D) WorkingLength (WL) ShaftDiameter (DS) PilotDiameter (PD) ShoulderLength (SL) FluteLength (FL) PilotLength (PL) PilotRadius (PR)	
Boring Bar: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA)	

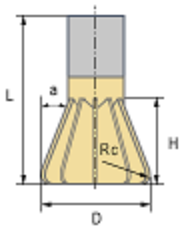
Tool type	View
Back Boring Bar: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) Height1 (h1) Height2 (h2) Angle1 (a1) Angle2 (a2) StemDiameter (Ds) StemAngle (Sa) StemHeight1 (sh1) StemHeight2 (sh2)	
Tap: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA) Angle (A) ThreadPitch (TP)	
Thread Mill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) StemDiameter (Ds) Pitch (P) PitchCount (c) OD_ID (OD/ID)	
Negative Radius Mill: OverallLength (L) CuttingDiameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL)	

Tool type	View
ShankTaperAngle (STA) Radius (Rc)	
Knife: OverallLength (L) Diameter (D) FrontDistance (RD) BackDistance (LD) FrontAngle (RA) BackAngle (LA) AutoDepth (LD) InclinationAngle (IA)	
Saw Blade: WorkingLength (WL) Diameter (D) Thickness (Th) RoundingRadius (R)	
Undercut Dove Mill: OverallLength (L) Diameter (D) WorkingLength (WL) Angle (a) CornerRadius (Rc) ShapeLength (H)	
Undercut Slot Mill: OverallLength (L) Diameter (D) WorkingLength (WL) ShaftDiameter (d) SideAngle (a) CornerRadius (Rc) ShapeLength (H)	

Tool type	View
Undercut Lollipop Mill: OverallLength (L) Diameter (D) WorkingLength (WL) ShaftDiameter (d)	
Undercut Two Angle Mill: OverallLength (L) Diameter (D) WorkingLength (WL) Height (H) HeadHeight (H2) Angle1 (ang) Angle2 (ang2) SmoothRadius (R) ShaftDiameter (W)	
Undercut Round Groove Mill: OverallLength (L) Diameter (D) WorkingLength (WL) Radius (R) StemDiameter (Ds)	
Undercut Sharp Chamfer Mill: OverallLength (L) Diameter (D) WorkingLength (WL) Height (S) Angle (Ang) Width (W) StemDiameter (Ds)	
Undercut Rounded Chamfer Mill: OverallLength (L) Diameter (D) WorkingLength (WL) Height (S) Angle (Ang)	

Tool type	View
Radius (R) StemDiameter (Ds)	
Undercut Indexable Chamfer Mill: OverallLength (L) Diameter (D) WorkingLength (WL) StemDiameter (Ds) InsertAngle (Ang) InsertInnerCircleDiameter (IC) InsertCornersCount (CNC) CornerRadius (Re)	
Undercut Barrel Mill: OverallLength (L) Diameter (D) WorkingLength (WL) CenterLength (LC) Height (H) StemLength (LH) EdgeR1 (RE1) EdgeR2 (RE2) EdgeR3 (RE3) StemDiameter (DN) ConnectDiameter (DCON)	
Undercut Lens Barrel Mill: OverallLength (L) Diameter (D) WorkingLength (WL) FacetDiameter (FC) EdgeR1 (RE1) EdgeR2 (RE2) Height (H) StemLength (LH) StemDiameter (DN) ConnectDiameter (DCON) Angle (a)	

Tool type	View
Undercut Taper Barrel Mill: OverallLength (L) Diameter (D) WorkingLength (WL) Height (H) EdgeR1 (RE1) EdgeR2 (RE2) EdgeR3 (RE3)	
Jet Cutter: OverallLength (L) Diameter (D) WorkingLength (WL) ShankDiameter (SHD) ShoulderLength (SHL) ShankTaperAngle (STA)	
Taper Jet Cutter: OverallLength (L) Diameter (D) WorkingLength (WL) JetLength (JL) NozzleDistance (ND) TaperAngle (Ang)	
Plazma Jet Cutter: OverallLength (L) Diameter (D) WorkingLength (WL) Length (h) Angle (a)	
Laser Beam Cutter: OverallLength (L) WorkingLength (WL) LensDiameter (LD) FocalDistance (F) CylinderZoneDiameter (CD) CylinderZoneLength (CL)	

Tool type	View
CurvatureRadius (R) LaserBeamLength (BL)	
Single Vacuum Gripper: OverallLength (L) WorkingLength (WL) Diameter (D) ShaftDiameter (shd) CornerRadius (Rc) ShapeLength (H)	

Helpful links:

- How to work with milling tools in the CAM interface
- Preparing the machining tools import api environment
- GitHub: Machining tool import API examples;

Continue to read:

- Working with the Tool Holder

Working with Turning Tools

Turning tools are formed from two parts: the insert part of the cutting edge and the tool holder type. To create a turning tool, the first thing you need to do is select a group of turning tools. Next, you need to choose a compatible type of tool holder and insert part of the cutting edge. You also need to choose tool hand direction. Common tool attributes, such as tool name, tool number, etc., are the same for all subtypes of cutting tools. For example, below is the code for creating a turn tool with an external holder:

```
//C#
using THDT = MachinigToolsImportTypes.TMTI_TurnToolHolderDimensionTypes;
using TIDT = MachinigToolsImportTypes.TMTI_TurnToolInsertDimensionTypes;
------------------------------------------------------------
//importer is an instance of IMTI_MachiningToolsImportLibrary
var tool = importer.CreateTurnToolWithExternalHolder();
tool.SetName("My Turn tool");
tool.SetIdentifier("t001");
tool.SetMagazineNumber(1);
tool.SetToolNumber(1);
tool.SetUnits(TMTI_LinearUnits.luMillimeter);
tool.SetDurability(70);

tool.GetHandType().Hand = TMTI_Hand.hRight;

bool isCompatable = tool.SetHolderAndInsertType(
    TMTI_ExternalToolHolderTypes.htA_90deg,
    TMTI_ExternalToolHolderInsertTypes.itC_80degRhombic);
if (isCompatable) {
    var tdim = tool.GetDimensions();
    tdim.Holder[THDT.hdL1] = 130;
    tdim.Holder[THDT.hdB] = 20;
    tdim.Holder[THDT.hdL3] = 30;
    tdim.Holder[THDT.hdF1] = 30;

    tdim.Insert[TIDT.idL] = 17;
    tdim.Insert[TIDT.idTi] = 2;
    tdim.Insert[TIDT.idRe] = 0.1;
    //if you have changed this property, the holder type is automatically set
to undefined
    //tdim.Insert[TIDT.idKr] = 90;
}

var tdir = tool.GetDirections();
tdir.FixingDirection = TMTI_FixingDirection.fdDirect;
tdir.CuttingDirection[TMTI_CuttingDirections.cdS] = true;
tdir.CuttingDirection[TMTI_CuttingDirections.cdE] = true;
```

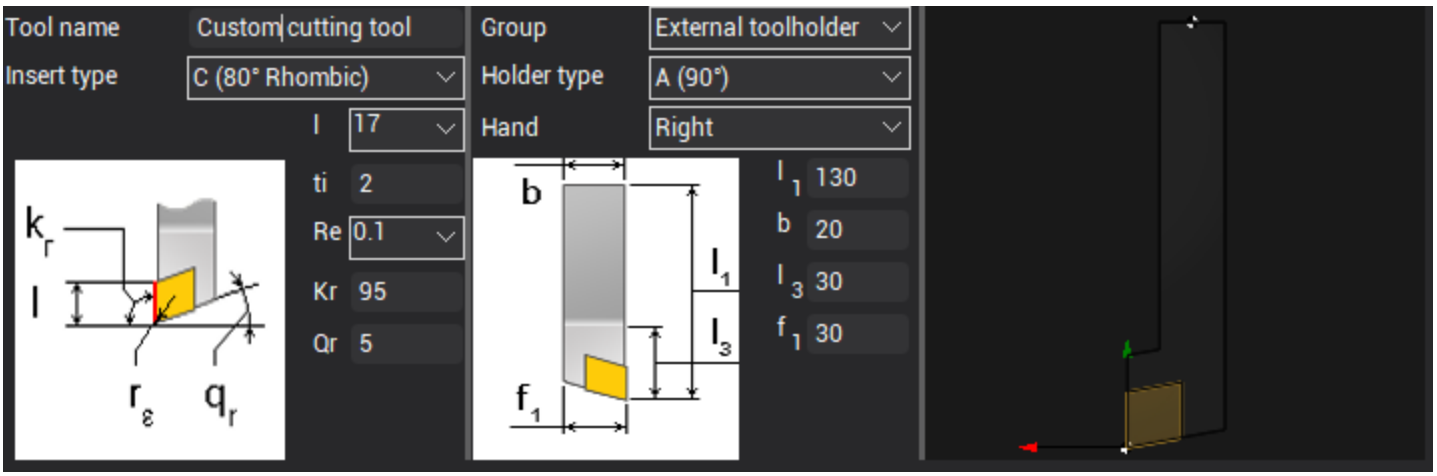
```
tdir.CuttingDirection[TMTI_CuttingDirections.cdSE] = true;

var overh = tool.GetOverhang();
overh.isAutoCalc = false;
overh.Axial = -10;
overh.Radial = 121;
overh.Matching = 1.5;

var tpoints = tool.GetToolingPoints();
tpoints.Corrector[0] = 1;
tpoints.Corrector[1] = 99;
tpoints.TPointX[0] = -2.22;
tpoints.TPointX[1] = 1.11;
tpoints.TPointY[0] = -1.55;
tpoints.TPointY[1] = -0.29;

var cond = tool.GetCuttingConditions();
cond.RotationDirection = TMTI_RotationDirection.rdCCW;
cond.CuttingSpeedMode = TMTI_CuttingSpeedMode.csmCSS;
cond.Coolant.TubeIsOn[0] = false;
cond.Coolant.TubeIsOn[1] = true;
cond.Coolant.TubeIsOn[2] = true;
cond.FeedValue = 2;
//after setting the properties, add this tool to your tool library instance
```

Code execution result:



Possible variants of compatible parts of the tool and their geometric dimensions can be viewed in the CAM interface.

Helpful links:

- How to work with turning tools in the CAM interface
- Preparing the machining tools import api environment

- GitHub: Machining tool import API examples;

Continue to read:

- Working with the Tool Holder

Working with custom Axial Shape Tools

Custom axial shape tools are generated from a curve drawn from points on a 2D plane. Common tool attributes, such as tool name, tool number, etc., are the same for all subtypes of cutting tools. For example, below is the code for creating a custom axial shape tool in a spherical group of milling tools.

```
//C#
//importer is an instance of IMTI_MachiningToolsImportLibrary
var tool = importer.CreateCustomAxialShapeTool();
tool.ToolGroup = TMTI_AxialToolGroup.tgSpherical;
tool.OverallLength = 100;
tool.SetName("My Custom Spherical Mill Tool");
tool.SetIdentifier("s001");
tool.SetTeethsCount(2);
tool.SetMagazineNumber(1);
tool.SetToolNumber(1);
tool.SetUnits(TMTI_LinearUnits.luMillimeter);
tool.SetDurability(70);
tool.SetMaxPlungeAngle(90);

var rcv = tool.BeginGeneratrix();
TST2DPoint p = new TST2DPoint() {X = 1, Y = 1};
TST2DPoint pc = new TST2DPoint() {X = 0.0359, Y = 7.3301};
rcv.SetSpanType(TMTI_CurveSpanType.csfWorking);
rcv.StartCurve(p);
p.X = 6; p.Y = 5;
rcv.ArcTo(pc, p, 6.4031);
p.X = 6; p.Y = 25;
rcv.CutTo(p);
rcv.SetSpanType(TMTI_CurveSpanType.csfNonWorking);
p.X = 6; p.Y = 30;
rcv.CutTo(p);
p.X = 7.5; p.Y = 40;
rcv.CutTo(p);

var toolings = tool.GetToolingPoints();
toolings.ToolContactPointType = TMTI_AxialToolContactPointType.cpCustomPoint;
toolings.ToolContactPointsCount = 1;

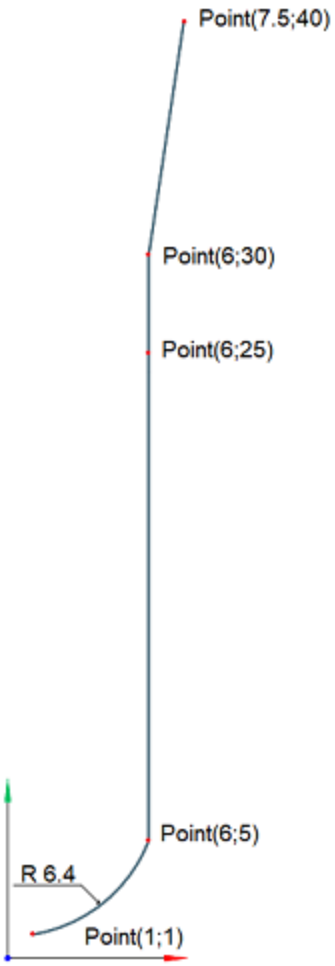
toolings.ToolingPointCount = 1;
toolings.ToolingPointType[0] = TMTI_AxialToolToolingPointType.tpCustomPoint;
toolings.ToolingPointShift[0] = 1.5;
toolings.ToolingPointLengthCorrectorNumber[0] = 1;
toolings.ToolingPointRadiusCorrectorNumber[0] = 1;
```

```
var cond = tool.GetCuttingConditions();
cond.CuttingSpeedMode = TMTI_CuttingSpeedMode.csmRPM;
cond.RotationsPerMinute = 100;
cond.SpindleGearRange = 1;
cond.RotationDirection = TMTI_RotationDirection.rdCW;
cond.FeedUnits = TMTI_FeedUnits.perMinute;
cond.FeedValue = 100;

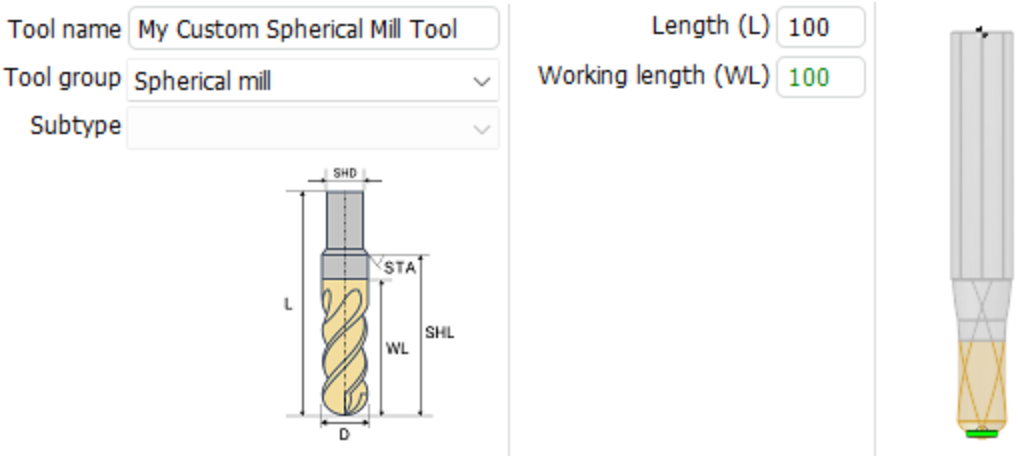
int tubesCount = cond.Coolant.TubeCount; //20
cond.Coolant.TubeIsOn[0] = true;
cond.Coolant.TubeIsOn[1] = false;
cond.Coolant.TubeIsOn[2] = true;

//after setting the properties, add this tool to your tool library instance
```

Curve result view:



Code execution result:



Helpful links:

- How to work with shaped tools in the CAM interface
- Preparing the machining tools import api environment
- GitHub: Machining tool import API examples;

Continue to read:

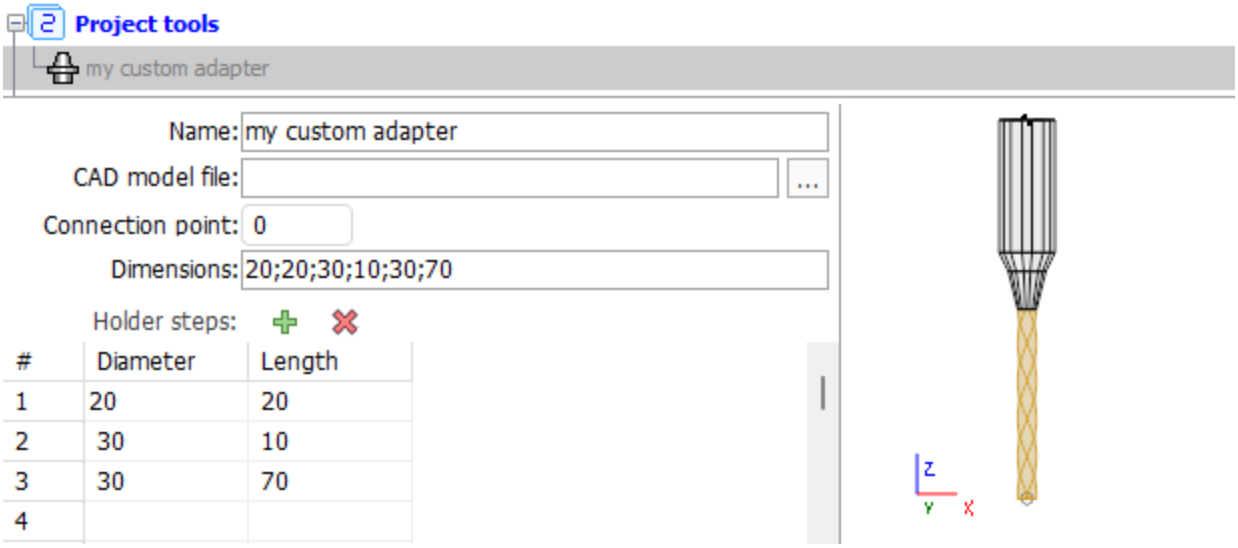
- Working with the Tool Holder

Tool Holder

The tool holder can be defined in two ways: by installing an existing CAD model tool holder and using the CAM's builder. The CAD model allows you to specify files in *.stl and *.osd formats. The builder allows you to create custom tool holders with a set of dimensions, defining the diameter and length for each of the steps. Below is an example of how to form a tool holder using a CAM's builder:

```
//C#
//importer is an instance of IMTI_MachiningToolsImportLibrary
var tool = importer.CreateCylindricalMill();
tool.SetName("My Cylindrical Mill Tool with custom tool holder");
var adapter = tool.GetStepsAdapter();
adapter.HolderName = "my custom adapter";
adapter.HolderStepCount=3;
adapter.HolderStepDiameter[0] = 20;
adapter.HolderStepHeight[0] = 20;
adapter.HolderStepDiameter[1] = 30;
adapter.HolderStepHeight[1] = 10;
adapter.HolderStepDiameter[2] = 30;
adapter.HolderStepHeight[2] = 70;
```

Code execution result:



For turning tools, you can install the holder using the finished CAD model only.

Helpful links:

- How to work with tool holder in the CAM interface
- Preparing the machining tools import api environment
- GitHub: Machining tool import API examples;

Return to Home Page

Interaction with external applications

Using CAMIPC, you can write an external application which:

- can run ENCY and then connect to it;
- can manage ENCY (open new project / calculate NC / edit operations list / etc);
- can open ENCY without any windows, so it will work faster;
- can open more than one ENCY and read information from all of them.
- can mimic the behaviour of user, for a example, it will press some buttons.

Right now, we are hardly working on expanding this list.

Connecting

ENCY provides CAMIPC.Helper.Cam.dll, which can be used to connect to application. Connection methods:

- In C#;
- In Delphi;
- In C++.

Connecting to main application in a C# application

The basic package of installed program includes:

- `CAMIPC.Helper.Cam.dll`, which contains the export function `CreateHelper`. This function returns an object of the IIpcHelper interface. The methods of this interface allow you to connect to an already running instance or start a new one.
- `NativeLibUtils.dll` - to simplify the loading and unloading of native DLLs.

File .csproj should contain:

- reference with path to NativeLibUtils.dll;
- package reference to CAMIPC SDK;

```
<ItemGroup>
    <PackageReference Include="EncySoftware.CAMIPC.SDK.Net" Version="1.1.5" />
    <Reference Include="C:\Program Files\ENCY
Software\ENCY\Bin64\NativeLibUtils.dll" />
</ItemGroup>
```

Support singleton class

This class provides public method `GetApplication()`, which connects to running instance.

```
using System;
using System.IO;
using System.Runtime.InteropServices;
using CAMAPI.ResultStatus;
using CAMIPC.Application;
using CAMIPC.ExecuteContext;
using CAMIPC.Helper;
using EncySoftware.NativeLibUtils;

namespace ApplicationEmptyNet;

public static class CamHelper
{
    private static IIpcHelper? _helper;
    private static ICamIpcApplication? _application;
    private static IntPtr _helperObjectPtr;
    private static IntPtr _helperDllPtr;
    private delegate IntPtr CreateHelperDelegate();

    /// <summary>
```

```
    /// Connect to CAM application
    /// </summary>
    private static IIpcHelper GetHelper()
    {
        if (_helper != null)
            return _helper;

        const string camFolder = @"C:\Program Files\ENCY Software\ENCY\Bin64";
        var helperPath = Path.Combine(camFolder, "CAMIPC.Helper.Cam.dll");
        if (!File.Exists(helperPath))
            throw new Exception($"{helperPath} not found");

        _helper = NativeLibLoader.CreateComObject<IIpcHelper, CreateHelperDelegate>(
                        helperPath, "CreateHelper", out _helperObjectPtr,
out _helperDllPtr)
                    ?? throw new Exception("Can't create helper");

        return _helper;
    }

    /// <summary>
    /// Get instance of running CAM application to interact with it
    /// </summary>
    public static ICamIpcApplication GetApplication()
    {
        if (_application != null)
            return _application;

        var executeContext = new TExecuteContext();
        var instances = GetHelper().GetRunningCamAppList(ref executeContext);
        if (executeContext.ResultStatus.Code == TResultStatusCode.rsError)
            throw new Exception(executeContext.ResultStatus.Description);
        if (instances.Count == 0)
            throw new Exception("ENCY running instance not found");

        _application = instances.Get(0, executeContext);
        if (executeContext.ResultStatus.Code == TResultStatusCode.rsError)
            throw new Exception(executeContext.ResultStatus.Description);

        return _application;
    }

    /// <summary>
    /// Destructor. Release COM objects
    /// </summary>
    public static void Clean()
```

```
    {
        if (_helper != null)
        {
            if (_application != null)
                Marshal.ReleaseComObject(_application);
            NativeLibLoader.FreeDll(_helper, _helperObjectPtr, _helperDllPtr);
        }
    }
}
```

Modificating support singleton class

IIpcHelper interface provides methods to:

- Connect to the first found instance (as mentioned above);
- Connect to ENCY by specifying the process PID;
- Start new ENCY and connect to it.

Connecting to main application in a Delphi application

Connecting to main application in a C++ application

Using ComWrapper

The CAM system core is written in a programming language where memory is released manually — as opposed to languages like C#, which rely on a Garbage Collector that frees memory automatically.

Therefore, when extensions written in .NET interact with the CAM system core, it is necessary to have a mechanism for explicitly releasing COM objects on demand. For this reason, the ComWrapper class was introduced to encapsulate all memory management logic for COM objects.

Problems solved by using the ComWrapper:

1. Accessing a COM object from multiple threads;
2. Proper memory cleanup after use, ensuring that the CAM system core can continue functioning correctly.

Examples:

Get another COM-object:

```
using var ncMakerCom = activeProjectCom.InvokeAndWrap(project => project.NCMaker);
```

Execute method of COM-object:

```
applicationCom.Invoke(application =>
{
    application.MainWorkMode = TMainWorkMode.mwmModel;
});
```

Transfering ownership:

```
var curveCom2 = curveCom1.TransferOwnership();
```

Conclusion

Using ComWrapper is mandatory, if your extension is written on .NET.

Developer guide

Developer guide for ENCY plugins (CAMAPI) and external applications (CAMIPC): concepts, domain reference with C# examples, and macros. Full example projects are in the cam-api-examples repository.

Start here

What are you building?

Task	Where to start
In-process plugin (DLL loaded by ENCY)	CAMAPI overview → Entry points guide
Standalone external app communicating with ENCY	CAMIPC overview → IPC connection guide
Not sure which API to use	See comparison below
ENCY shows an error or crashes when closing after my plugin ran — undestroyed-objects dialog, `LostObjects_*.txt`, AV on exit	COM lifetime → Symptoms — almost always a missing `using`

CAMAPI vs CAMIPC

	CAMAPI	CAMIPC
Deployment	DLL plugin, loaded by ENCY	Standalone .exe, connects via IPC
Trigger	User clicks button / lifecycle event	Initiated by external app
Performance	In-process, direct COM calls	IPC overhead per call
Typical use	Custom operations, UI extensions, solvers	Automation, integration, testing
Entry point	`IExtension*` interfaces	`IIpcHelper` → `ICamIpcApplication`

Documentation map

Concepts (read these first)

- Extension entry points — all 7 plugin types, when to use each, full C# boilerplate
- COM lifetime management — ComWrapper, Invoke vs InvokeAndWrap, ListComWrapper, MTA, event-handler arguments, and what a forgotten `using` looks like at application shutdown
- Error handling — TResultStatus, no-exceptions rule, IExtensionLogger

- UI patterns — Inspector dialog, modal WPF (STA thread), non-modal WPF + IExtensionLazyUnloadable
- Theming plugin windows — applying the host palette to your own window; WinForms visual-styles pitfalls, scrollbars, title bar

Writing macros

- Macros — authoring a macro body (`ICamApiMacro.Run`, run context, `MacroParams`, `NotifyMacroStep`). Managing/building/running macros from a host is in api/application.md (in-process) and ipc/application.md (IPC).

CAMAPI domain reference

- Application & singletons — ICamApiApplication, utilities, macros, events, collections
- Entry points — IExtensionManager, IExtensionStorage, IExtensionLogger
- Project — ICamApiProject, technologist, tech operations, parts, stages, snapshots
- Geometry — model tree, entities, faces, curves, mesh, coordinate systems, GeomPicker, sketcher, point snapper
- CAD — CAD modelling: sketches, extrude, revolve, cut / union / intersect, fillet, chamfer, draft, bodies
- Feature finder — automatic recognition of holes, pockets, fillets, chamfers, planes, edges
- Tools & machine — machining tools, tool lists, machines, workpiece setup, live machine state
- NC & simulation — NCMaker, Simulator, CLDReceiver, ModelFormer, TechOperationSolver
- UI — main form, viewport, view cube, dialogs, SimplePropIterator, CamApiInspectorWindow

CAMIPC domain reference

- Connection — DLL loading, IIpcHelper, TExecuteContext, IAsyncResult, event listeners
- Application — ICamIpcApplication, paths, extension manager, logger, XmlProp
- Project — differences from CAMAPI project domain
- Geometry — differences from CAMAPI geometry domain
- CAD — differences from CAMAPI CAD domain
- Feature finder — differences from CAMAPI feature-finder domain
- Tools & machine — differences from CAMAPI tools domain
- NC & simulation — differences from CAMAPI NC domain
- UI — main form, viewport, view cube, PrimeView, PrimeViewModel
- Extra — CloudsApp, FunctionalTest, Extension.PLM
- Macros — record & save over IPC — MacroManager → builder → commands, recording lifecycle, command schema, guard cases

Key patterns (quick reference)

Minimal CAMAPI plugin

```
// 1. Implement IExtension + chosen entry point interface
public class MyPlugin : IExtension, IExtensionUtility {
    public IExtensionInfo? Info { get; set; }

    public void Run(IExtensionUtilityContext context, out TResultStatus
resultStatus) {
        resultStatus = default;
        try {
            using var appCom = ComWrapper.Create(context.CamApplication);
            using var projectCom = appCom.GetActiveProject();
            // your logic here
        } catch (Exception e) {
            resultStatus.Code = TResultStatusCode.rsError;
            resultStatus.Description = e.Message;
        }
    }
}

// 2. Register in extension.json (see general/extension-entry-points.md for
full format)
```

Minimal CAMIPC connection

```
// Load helper DLL and connect (see ipc/connection.md for full details)
var helper = IpcHelperLoader.Load("CAMIPC.Helper.Cam.dll");
var context = new TExecuteContext { Timeout = 5000 };
using var appCom = helper.ConnectToRunningInstance(context);
```

COM lifetime rule

```
// Always use 'using' — never store COM objects in fields without explicit
lifetime control
using var projectCom = appCom.GetActiveProject();   // correct
var project = appCom.GetActiveProject();             // WRONG — leaks COM reference
```

`ComWrapper<T>` has no finalizer: an undisposed wrapper is never released, and the host reports it (or crashes) when ENCY closes — see Symptoms.

ENCY CAM API — Extension Entry Points

ENCY loads plugins from managed .NET DLLs. Each DLL exposes a single factory class (`CAMAPI.ExtensionFactory`) that ENCY finds by reflection. When ENCY needs an extension instance it calls `ExtensionFactory.Create(extensionIdent)`, where `extensionIdent` is the unique string registered in the extension's JSON config file.

Every plugin class must implement **`IExtension`** (to carry metadata) plus exactly one role interface that determines when and how ENCY invokes it. The role interface is called the *entry point*.

Choosing an Entry Point

Goal	Entry point
Run code on CAM startup / shutdown	`IExtensionGlobal`
Add a button to the toolbar / utilities menu	`IExtensionUtility`
Intercept or customise how a utility is executed	`IExtensionUtilityRunner`
Add items to the right-click menu on a tech operation	`IExtensionOperationPopup`
Add items to the right-click menu on a geometry tree node	`IExtensionGeomModelNodePopup`
Implement a custom toolpath calculation algorithm	`ICamApiTechOperationSolver`
Post-process (transform) CLD toolpath data after calculation	`IExtensionGeomCLDataConverter`

Entry Points Reference

IExtensionTypeInfoGlobal → IExtensionGlobal

Use when: you need to execute code once when ENCY starts (`OnSCInitializing`) and once when it shuts down (`OnSCFinalizing`). Typical uses: register background services, subscribe to application-level events, initialise caches.

Context received: none passed directly. Use `ICamApiApplicationSingleton` (obtained via the singleton mechanism) to reach `ICamApiApplication` and all its children.

What you can access: the full application object — active project, extension manager, tools manager, machines library, utility manager, macro manager.

Implementation:

```
using CAMAPI.Application;
using CAMAPI.Extensions;
using CAMAPI.ResultStatus;

namespace MyPlugin;

public class MyGlobalExtension : IExtension, IExtensionGlobal
{
    public IExtensionInfo? Info { get; set; }

    public TResultStatus OnSCInitializing()
    {
        TResultStatus status = default;
        try
        {
            // Initialisation logic here.
            // Reach the application via ICamApiApplicationSingleton if needed.
        }
        catch (Exception e)
        {
            status.Code = TResultStatusCode.rsError;
            status.Description = e.Message;
        }
        return status;
    }

    public TResultStatus OnSCFinalizing()
    {
        TResultStatus status = default;
        try
        {
            // Cleanup logic here.
        }
        catch (Exception e)
        {
            status.Code = TResultStatusCode.rsError;
            status.Description = e.Message;
        }
        return status;
    }
}
```

Example: ExtensionGlobal/ExtensionGlobalNet/project/main/ExtensionGlobal.cs

IExtensionTypeInfoUtility → IExtensionUtility

Use when: you want to add a button to the ENCY utilities menu. When the user clicks the button, `Run` is called with the full application context.

Context received: `IExtensionUtilityContext` — contains `CamApplication` (`ICamApiApplication`).

What you can access: active project (`GetActiveProject`), extension manager (`GetExtensionManager`), main window handle, tools manager, machines library, utility manager.

JSON type info fields: `Caption` (button label), `HintText`, `IconPath` (relative to DLL).

Implementation:

```
using CAMAPI.Application;
using CAMAPI.DotnetHelper;
using CAMAPI.Extensions;
using CAMAPI.Project;
using CAMAPI.ResultStatus;

namespace MyPlugin;

public class MyUtilityExtension : IExtension, IExtensionUtility
{
    public IExtensionInfo? Info { get; set; }

    public void Run(IExtensionUtilityContext context, out
TResultStatus resultStatus)
    {
        resultStatus = default;
        try
        {
            using var appCom = ComWrapper.Create(context.CamApplication);
            using var projectCom = appCom.GetActiveProject();
            if (projectCom.IsNull)
                throw new Exception("No active project");

            // Work with the project through helper extensions / Invoke.
            Console.WriteLine(projectCom.FilePath());
        }
        catch (Exception e)
        {
            resultStatus.Code = TResultStatusCode.rsError;
            resultStatus.Description = e.Message;
        }
```

```
    }
}
```

> Direct access (IDL): `IExtensionUtility.Run(IExtensionUtilityContext* Context, out TResultStatus)`

Getting the currently selected operation: `IExtensionUtilityContext` has no `SelectedOperation` — use `technologist.CurrentOperation` instead:

```
using var appCom = ComWrapper.Create(context.CamApplication);
using var projectCom = appCom.InvokeAndWrap(app => app.GetActiveProject(out _))
    ?? throw new Exception("No active project.");
using var technologistCom = projectCom.InvokeAndWrap(p => p.Technologist);
using var operationCom = technologistCom.InvokeAndWrap(t => t.CurrentOperation)
    ?? throw new Exception("No operation is selected.");
```

Returns `null` if nothing is selected in the technology tree.

Example: [ExtensionEmpty/ExtensionEmptyNet/project/main/ExtensionTest.cs]

IExtensionTypeInfoUtilityRunner → IExtensionUtilityRunner

Use when: you need to intercept or wrap the execution of another utility — for example, to provide a custom file selector, add pre- or post-processing, or redirect execution to a different handler.

Context received: `IExtensionUtilityRunnerContext` — contains:

- `UtilButtonContext` (`IUtilButtonContext`) — properties of the utility button being run.
- `UtilityContext` (`IExtensionUtilityContext`) — the standard utility context (application instance).

What you can access: everything available through `IExtensionUtilityContext.CamApplication`, plus button metadata.

JSON type info fields: `FilterTypes` (file filter string for the file selector dialog), `VisibleInEditor` (whether the runner appears in the utility editor).

Implementation:

```
using CAMAPI.Application;
using CAMAPI.Extensions;
using CAMAPI.ResultStatus;

namespace MyPlugin;
```

```
public class MyUtilityRunner : IExtension, IExtensionUtilityRunner
{
    public IExtensionInfo? Info { get; set; }

    public void Execute(IExtensionUtilityRunnerContext context, out
TResultStatus resultStatus)
    {
        resultStatus = default;
        try
        {
            var app = context.UtilityContext.CamApplication;
            // Intercept or augment the utility run here.
        }
        catch (Exception e)
        {
            resultStatus.Code = TResultStatusCode.rsError;
            resultStatus.Description = e.Message;
        }
    }
}
```

> Direct access (IDL): `IExtensionUtilityRunner.Execute(IExtensionUtilityRunnerContext* Context, out TResultStatus)`

IExtensionTypeInfoOperationPopup → IExtensionOperationPopup

Use when: you want to add custom items to the right-click context menu that appears on a technology operation in the ENCY technology tree.

Context received: `IExtensionOperationPopupBuildContext` — contains:

- `SelectedOperation` (`ICamApiTechOperation*`) — the operation that was right-clicked.
- `ActiveProject` (`ICamApiProject*`) — the current project.
- `OperationPopup` (`ICamApiTechnologyFormOperationPopup*`) — the popup builder; call `AddItem` to add entries.

What you can access: operation properties (name, type, XML params, tool, machine, LCS, flags), the active project tree, and the popup menu builder.

Implementation:

```
using CAMAPI.Extensions;
using CAMAPI.ResultStatus;
using CAMAPI.TechnologyForm;

namespace MyPlugin;

public class MyOperationPopup : IExtension, IExtensionOperationPopup
{
    public IExtensionInfo? Info { get; set; }

    public TResultStatus Build(IExtensionOperationPopupBuildContext context)
    {
        TResultStatus status = default;
        try
        {
            var popup = context.OperationPopup;
            var op = context.SelectedOperation;

            popup.AddItem(
                name: "my_action",
                caption: "My Action",
                enabled: true,
                onClicked: new MyClickHandler(),
                resultStatus: out _);
        }
        catch (Exception e)
        {
            status.Code = TResultStatusCode.rsError;
            status.Description = e.Message;
        }
        return status;
    }
}
```

> Direct access (IDL):
> `IExtensionOperationPopup.Build(IExtensionOperationPopupBuildContext* Context, out TResultStatus)`

Example:
ExtensionOperationPopup/ExtensionOperationPopupNet/project/main/ExtensionOperationPopup.cs

---

IExtensionTypeInfoGeomModelNodePopup → IExtensionGeomModelNodePopup

Use when: you want to add custom items to the right-click context menu on a node in the 3D geometry model tree.

Context received: `IExtensionGeomModelNodePopupBuildContext` — contains:

- `SelectedNode` (`ICAMAPIGeometryTreeNode*`) — the geometry tree node that was right-clicked.
- `ActiveProject` (`ICamApiProject*`) — the current project.
- `NodePopup` (`ICamApiGeomModelNodePopup*`) — the popup builder; call `AddItem` to add entries.

What you can access: geometry tree node properties, the active project, and the popup menu builder.

Implementation:

```
using CAMAPI.Extensions;
using CAMAPI.GeometryModelForm;
using CAMAPI.ResultStatus;

namespace MyPlugin;

public class MyGeomNodePopup : IExtension, IExtensionGeomModelNodePopup
{
    public IExtensionInfo? Info { get; set; }

    public TResultStatus Build(IExtensionGeomModelNodePopupBuildContext context)
    {
        TResultStatus status = default;
        try
        {
            context.NodePopup.AddItem(
                name: "show_full_name",
                caption: "Show Full Name",
                enabled: true,
                onClicked: new MyNodeClickHandler(),
                resultStatus: out _);
        }
        catch (Exception e)
        {
            status.Code = TResultStatusCode.rsError;
            status.Description = e.Message;
        }
        return status;
```

```
    }
}
```

> Direct access (IDL):
> `IExtensionGeomModelNodePopup.Build(IExtensionGeomModelNodePopupBuildContext* Context, out TResultStatus)`

Example:
GeometryModelNodePopup/NodeFullNameAlert/project/main/ExtensionGeomModelNodePopup.cs

---

IExtensionTypeInfoOperationSolver → ICamApiTechOperationSolver

Use when: you are implementing a custom toolpath calculation algorithm for one or more operation types. ENCY calls `MakeWorkPath` to generate the toolpath and `GetPropIterator` to expose operation parameters in the UI.

Context received (MakeWorkPath):

- `cldFormer` (`ICamApiCLDReceiver*`) — the CLD output sink; call `CutTo`, `ArcTo2d`, `OutStandardFeed`, `AddComment`, etc. to emit toolpath commands.
- `TechOperation` (`ICamApiTechOperation*`) — the operation being calculated; read parameters from `XMLProp`.

Context received (InitSolver):

- `ICamApiTechOperationSolverInitializeContext` — carries the operation and a progress update handler.

What you can access: all operation properties via `ICamApiTechOperation`, the full XML parameter tree, the machine, tool, LCS, and model formers. Log through `ExtensionManagerHelper.GetInstance()`.

Implementation:

```
using CAMAPI.DotnetHelper;
using CAMAPI.Extensions;
using CAMAPI.MCDFormerTypes;
using CAMAPI.ResultStatus;
using CAMAPI.TechOperation;
using STCustomPropTypes;

namespace MyPlugin;

public class MyOperationSolver : IExtension, ICamApiTechOperationSolver
```

```
{
    public IExtensionInfo? Info { get; set; }

    public void InitSolver(ICamApiTechOperationSolverInitializeContext context,
        out TResultStatus resultStatus)
    {
        resultStatus = default;
        // Store context.TechOperation if needed for GetPropIterator.
    }

    public void FinalizeSolver()
    {
        // Release any stored references.
    }

    public bool GetPropIterator(string pageId,
        out IST_CustomPropIterator? iterator,
        out TResultStatus resultStatus)
    {
        resultStatus = default;
        iterator = null;
        return false; // Return true and populate iterator to expose UI parameters.
    }

    public void OnPropFilterChanged(string parameterName, string value) { }

    public void MakeWorkPath(ICamApiCLDReceiver cldFormer,
        ICamApiTechOperation techOperation,
        out TResultStatus resultStatus)
    {
        resultStatus = default;
        try
        {
            using var xmlPropCom = ComWrapper.Create(techOperation.XMLProp);
            double depth = xmlPropCom.Invoke(xp => xp.Flt["MyParams.Depth"]);

            // Emit toolpath commands:
            cldFormer.OutStandardFeed((int)TFeedTypeFlag.affRapid);
            cldFormer.CutTo(new TST3DPoint { X = 0, Y = 0, Z = 10 });
            cldFormer.OutStandardFeed((int)TFeedTypeFlag.affWorking);
            cldFormer.CutTo(new TST3DPoint { X = 0, Y = 0, Z = -depth });
            // ... more moves ...
        }
        catch (Exception e)
        {
            resultStatus.Code = TResultStatusCode.rsError;
```

```
            resultStatus.Description = e.Message;
        }
    }
}
```

> Direct access (IDL): `ICamApiTechOperationSolver.MakeWorkPath(ICamApiCLDReceiver*, ICamApiTechOperation*, out TResultStatus)`

Example: Operation/ExtensionOperationSimpleNet/project/main/ExtensionOperationSimpleNet.cs

---

IExtensionTypeInfoGeomCLDataConverter → IExtensionGeomCLDataConverter

Use when: you want to intercept and transform CLD toolpath data after it has been calculated by the solver, before it reaches the next stage of the pipeline. Common uses: add custom CLData records, filter moves, apply coordinate transformations.

Context received (GetCLDReceiverWrapper):

- `TechOperation` (`ICamApiTechOperation*`) — the operation whose toolpath is being converted.
- `Receiver` (`ICamApiCLDReceiver*`) — the downstream CLD receiver that your wrapper must delegate to.

Return value: a new `ICamApiCLDReceiver` implementation that wraps the supplied `Receiver`. ENCY feeds all CLD commands to the returned wrapper.

JSON type info fields (IExtensionTypeInfoGeomCLDataConverter):

- `FilterType` — `"all"` to apply to every operation, `"exact"` to apply only to operations listed in `OperationTypes`.
- `OperationTypes` — list of operation type identifiers (used when `FilterType` is `"exact"`).

Implementation:

```
using CAMAPI.Extensions;
using CAMAPI.MCDFormerTypes;
using CAMAPI.ResultStatus;
using CAMAPI.TechOperation;

namespace MyPlugin;

public class MyCLDataConverter : IExtension, IExtensionGeomCLDataConverter
```

```
{
    public IExtensionInfo? Info { get; set; }

    private ICamApiCLDReceiver? _wrapper;

    public ICamApiCLDReceiver? GetCLDReceiverWrapper(
        ICamApiTechOperation operation,
        ICamApiCLDReceiver receiver,
        out TResultStatus ret)
    {
        ret = default;
        try
        {
            if (receiver == null)
                throw new Exception("Receiver is null");
            _wrapper = new MyCLDReceiverWrapper(receiver); // your wrapper class
            return _wrapper;
        }
        catch (Exception e)
        {
            ret.Code = TResultStatusCode.rsError;
            ret.Description = e.Message;
            return null;
        }
    }

    public void FinalizeConverter()
    {
        if (_wrapper is IDisposable d)
        {
            d.Dispose();
            _wrapper = null;
        }
    }
}
```

> Direct access (IDL):
> `IExtensionGeomCLDataConverter.GetCLDReceiverWrapper(ICamApiTechOperation*, ICamApiCLDReceiver*, out TResultStatus, return ICamApiCLDReceiver*)`

Example:
[CLData/ExtensionGeomCLDataConverterNet/project/main/ExtensionGeomCLDataConverter.cs]

IExtensionLazyUnloadable

If your extension opens a window or holds an external resource that must remain alive until explicitly closed, implement `IExtensionLazyUnloadable` in addition to the role interface.

```
public class MyWindowExtension : IExtension, IExtensionUtility,
IExtensionLazyUnloadable
{
    public IExtensionInfo? Info { get; set; }

    // Set to false while a window is open; ENCY will not unload the extension.
    public bool CanUnload { get; set; } = true;

    public void Run(IExtensionUtilityContext context, out
TResultStatus resultStatus)
    {
        resultStatus = default;
        CanUnload = false;
        // Open window; set CanUnload = true when it closes.
    }
}
```

ENCY checks `CanUnload` before unloading the extension library. When `CanUnload` is `false`, unloading is deferred until `CanUnload` becomes `true` or the application terminates.

ExtensionFactory Boilerplate

Every library must expose a class named exactly `ExtensionFactory` in namespace `CAMAPI`. This is a hard requirement — ENCY locates the factory by this fully-qualified name.

```
// Namespace and class name must be exactly CAMAPI.ExtensionFactory.
namespace CAMAPI;

using Extensions;
using ResultStatus;

public class ExtensionFactory : IExtensionFactory
{
    public void OnLibraryRegistered(IExtensionFactoryContext context, out
TResultStatus ret)
    {
        ret = default;
        // Optional: read context.Constants or context.Paths at registration time.
    }
```

```
    public void OnLibraryUnRegistered(IExtensionFactoryContext context, out
TResultStatus ret)
    {
        ret = default;
    }

    public IExtension? Create(string extensionIdent, out TResultStatus ret)
    {
        try
        {
            ret = default;
            return extensionIdent switch
            {
                "My.Extension.Ident.One" => new MyFirstExtension(),
                "My.Extension.Ident.Two" => new MySecondExtension(),
                _ => throw new Exception("Unknown extension identifier: "
+ extensionIdent)
            };
        }
        catch (Exception e)
        {
            ret.Code = TResultStatusCode.rsError;
            ret.Description = e.Message;
            return null;
        }
    }
}
```

A single DLL can contain multiple extensions. Each gets its own unique `extensionIdent` matched by the `switch` above.

Extension JSON Registration

Each extension library is described by a JSON config file (`<AssemblyName>.settings.json`) placed next to the DLL. It specifies the DLL path and metadata for every extension inside.

Minimal utility example

```
{
    "name": "My plugin",
    "description": "What it does",
    "module_path": "${extensionJsonFolder}/MyPlugin.dll",
    "extensions": [
```

```
        {
            "utility": {
                "name": "My plugin",
                "caption": "My plugin",
                "id": "Extension.Utility.My.Plugin"
            }
        }
    ]
}
```

- `${extensionJsonFolder}` — resolved to the folder containing the JSON file at runtime.
- `caption` — the label shown in the ENCY utilities menu.
- `id` — must match exactly the string checked in `IExtensionFactory.Create`.

Copying the JSON to the build output

Add this to the `.csproj` so the file lands next to the DLL on every build:

```
<ItemGroup>
  <None Update="MyPlugin.settings.json">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

COM Object Lifetime Management in .NET

This document is critical reading for all .NET developers working with the ENCY CAM API. Mishandling COM lifetime is the most common source of crashes and subtle memory corruption in plugin and application code.

The one rule: every COM object the API hands you is a resource. Wrap it in a `ComWrapper<T>` and dispose it — `using` — before the method that received it returns. There is no garbage-collector safety net: see Symptoms for what happens when you skip it.

Symptoms: what a missing `using` looks like

A leaked COM object almost never fails where the bug is. Your plugin runs fine, does its job, returns — and ENCY breaks minutes later, when the user closes it. If you arrived here from one of the symptoms below, a missing `using` is the first thing to check.

1. Undestroyed-objects dialog on exit

An error dialog with OK/Cancel buttons appears while ENCY shuts down. It reports:

- the name of the running executable;
- the module being unloaded at that moment, e.g. `CAMAPI.ExtensionManager.dll`;
- how many objects of that module were still alive;
- the full path of the report file — `LostObjects_<module>.txt`, described below;
- the first 1000 characters of that report.

Pressing OK opens the report in Notepad. Every Delphi module counts its own objects, so one shutdown can produce several dialogs and several report files.

This dialog only exists in a Debug build of ENCY. The object counter is compiled out of Release entirely — no dialog, no file, no overhead. A release ENCY does not mean your plugin is clean; it means nobody is counting. The same leak then surfaces as symptom 4, or not at all until a user hits it.

2. `LostObjects_<module>.txt`

- Where: the folder of the running `.exe` (`…\EXE\Bin64\`) — *not* your plugin folder and not `%TEMP%`.
- Name: one file per Delphi module — `LostObjects_SCKernel.txt`, `LostObjects_CAMAPI.ExtensionManager.txt`, …
- Written only when something leaked; the previous file is deleted first.

```
Application Name: <full path to the exe>
Instance Name:    <full path to the module that reported>
File write date: <timestamp>
```

```
Lost objects count: 5

TExtensionInfo       ID:1043    Thread:29104
TInstanceInfo        ID:1044    Thread:29104
...
```

`ID` is the object's creation sequence number, `Thread` is the thread it was created on. There are no reference counts and no creation stacks in the file — stack capture exists in the host but is disabled in shipped builds.

How to read it: the names are the Delphi classes implementing the interfaces you obtained — `TExtensionInfo` is the object behind `IExtensionInfo`, and so on. Take each class name, find every call in your plugin that returns the matching interface, and check that each one is `using`-wrapped or explicitly disposed. Repeated identical names mean a leak inside a loop.

3. "Destroy called while references remain" dialog

Another Debug-only dialog, raised the moment the host destroys an object whose reference count is still non-zero. It names the Delphi class of the object and the number of references left on it, and blames a wrong destruction order of related objects.

For a plugin author the meaning is narrower: the host destroyed the object while your code still held a reference to it. In a Release build the same situation is silent, and the next call made through that now-dangling reference is an access violation.

4. Access violation when closing ENCY (any build)

Three ways a lifetime bug turns into a crash instead of a message:

- GC-timed release. `ComWrapper<T>` has no finalizer. The pointer it takes with `Marshal.GetIUnknownForObject` is released by `Dispose()` and by nothing else — an undisposed wrapper is a permanent leak, not a delayed one. What the GC *does* eventually collect is the per-thread RCW the wrapper holds, and the CLR then calls `Release` on it late, from the finalizer thread. If that lands after the Delphi module has finalized, the call goes into unloaded code → AV.
- Over-releasing a host singleton. Disposing a wrapper around a long-lived host object (`IExtensionManager` and friends) from an assembly-unload or finalizer path can drop its last reference while ENCY is still using it; every subsequent access then AVs. Dispose where you acquired, at the end of the method — not from teardown hooks.
- Reverse order. The host frees an object, your code then calls through the reference it kept. In Debug you get dialog 3 first; in Release you get the AV directly.

5. "ENCY did not close correctly" on the next start

Unrelated mechanism, worth knowing so you do not chase it: ENCY flags an unclean shutdown from pid files it deletes on a normal exit. A leak alone does not set that flag — a normal exit stays a normal exit even with objects left over. If you see this report, the process actually died, i.e. you are in symptom 4.

Checking your own plugin

Symptoms 1–3 require a Debug build of ENCY. If you have one, close it after exercising your plugin and look in the exe folder for `LostObjects_*.txt` — that is the whole test. With a release build the host reports nothing, so the check has to be the code review below.

For an unattended run, a debug ENCY started in job mode (`/JOB_MODE`) does not show the dialog: it sets process exit code 1 instead. That makes "did my plugin leak?" a usable CI check — non-zero exit plus a `LostObjects_*.txt` next to the exe.

Leak checklist:

Pattern	Verdict
`using var xCom = ...InvokeAndWrap(...)`	correct
`var xCom = ...InvokeAndWrap(...)` without `using`	leak
`Invoke(x => x.SomeComProperty)` assigned to a plain variable	leak — use `InvokeAndWrap`
`AsInstanceOf<T>()` / `As*()` helper inside a loop without `using`	leak on every iteration
`break` out of a `foreach` over a COM iterator	leaks the iteration variable — `Dispose()` it before `break`
wrapper stored in a field	must be disposed in the owner's `Dispose()`
wrapper disposed from an unload hook / finalizer	over-release — dispose it where you took it
handler still registered when the application closes	later events reach objects already being destroyed — unregister first

---

Why ComWrapper<T> Exists

The ENCY host application is written in Delphi and implements a classic COM reference-counting model. Every COM object has an internal reference count. When that count reaches zero the object is destroyed.

In .NET, the garbage collector manages memory non-deterministically — it may release a Runtime Callable Wrapper (RCW) seconds or minutes after the last managed reference disappears, or never during a short-lived process. If the GC finalizes an RCW at the wrong moment, the underlying COM `Release` call arrives after the Delphi object has already been invalidated by other means, causing access violations or use-after-free crashes.

`ComWrapper<T>` solves this by:

1. Taking explicit ownership of a COM object at construction time via `Marshal.GetIUnknownForObject` (incrementing the ref count).
2. Implementing `IDisposable` — calling `Dispose()` deterministically releases all held pointers in the correct order.
3. Providing cross-thread access: ENCY's COM objects live on MTA threads. `ComWrapper<T>` marshals the pointer safely so that code on any thread (including a WPF STA thread) can call COM methods without COM apartment violations.

`ComWrapper<T>` deliberately has no finalizer. The reference it takes is released by `Dispose()` and by nothing else, so a wrapper you forget to dispose is not released late — it is never released at all. Disposal is not an optimisation you can leave to the runtime; it is the only thing that ends the object's life.

The rule is simple: every COM object you receive from the API must be wrapped in a `ComWrapper<T>` and disposed — either with `using` or an explicit `Dispose()` call — before the method that received it returns.

ComWrapper<T> API

Constructors and Factory Method

```
// Wrap a COM object obtained directly from an API call.
// The wrapper takes ownership (increments ref count internally).
using var nodeCom = new ComWrapper<IMyInterface>(someComObject);

// Preferred factory shorthand — identical semantics, infers the type.
using var nodeCom = ComWrapper.Create(someComObject);
```

Both accept `null`/`IntPtr.Zero` gracefully and produce an "empty" wrapper whose `IsNull` property returns `true`.

Calling Methods That Return No COM Object

Use `Invoke(Action<T>)` when the COM method has no return value, or when the return value is a plain .NET type (string, int, bool, struct):

```
using var technologistCom = ComWrapper.Create(context.CamApplication.Technologist);

// No return value
technologistCom.Invoke(t => t.ResetAllOperationsToolpath());

// Scalar return value — captured via closure
string rootId = string.Empty;
technologistCom.Invoke(t => rootId = t.RootOperation.Id);
```

There is also a typed overload that returns a value directly:

```
string execPath = applicationCom.Invoke(app => app.ExecutablePath);
```

If the COM method returns a `TResultStatus` alongside its result you can use the status-checking overload, which throws automatically on error:

```
// Overload: Func<T, (TReturn Result, TResultStatus Status)>
// Throws if status.Code == rsError
string id = technologistCom.Invoke(t =>
    (t.CreateOperation("TSTFaceMillingOp", prevId, "", out var status), status));
```

Calling Methods That Return Another COM Object

Use `InvokeAndWrap<TResult>` when the COM method returns another COM interface. The result is automatically wrapped in a new `ComWrapper<TResult>` that takes ownership.

```
using var activeProjectCom = applicationCom.InvokeAndWrap(app =>
    app.GetActiveProject(ref executeContext));

using var technologistCom = activeProjectCom.InvokeAndWrap(project =>
    project.Technologist);

// With status checking — throws on error
using var operationCom = technologistCom.InvokeAndWrap(t =>
    (t.CreateOperation("TSTFaceMillingOp", prevId, "", out var status), status));
```

Never return a raw COM interface pointer out of `Invoke` and assign it to a plain variable. Always use `InvokeAndWrap` for COM-returning calls.

Accessing the Raw COM Object

`ComWrapper<T>` exposes two deprecated raw-access properties — `Instance` (nullable) and `It` (throws on empty). Do not use them in new code. Both are marked `[Obsolete]` and give warning CS0618; they bypass MTA marshalling and can fail unpredictably when the RCW is accessed off its apartment thread.

Use instead:

Need	Use
Read a property / call a method	`wrapperCom.Invoke(x => x.Property)` / `wrapperCom.Invoke(x => x.Method(...))` or the `*Helper.cs` extension method
Call a method that returns another COM object	`wrapperCom.InvokeAndWrap(x => x.Method(...))`
Check whether the wrapper is empty	`wrapperCom.IsNull`
Pass a wrapped COM object as argument to another COM method	Nest the calls so the argument is read inside the outer `Invoke`: `outerCom.Invoke(outer => innerCom.Invoke(inner => outer.DoWith(inner)))`

If a scenario seems to require `.It` / `.Instance`, the fix is usually to add a missing extension method to `CAMAPI.DotnetHelper` — not to reach for the raw pointer.

The `using` Pattern

Always use `using` for `ComWrapper<T>` instances. Treat every COM-returning call as a resource allocation:

```
public void Run(IExtensionUtilityContext context, out TResultStatus resultStatus)
{
    resultStatus = default;
    try
    {
        using var applicationCom = ComWrapper.Create(context.CamApplication);
        using var projectCom = applicationCom.InvokeAndWrap(app =>
            app.GetActiveProject(ref executeContext));
        using var technologistCom = projectCom.InvokeAndWrap(p => p.Technologist);

        // work with technologistCom ...
    }
    catch (Exception e)
    {
        resultStatus.Code = TResultStatusCode.rsError;
        resultStatus.Description = e.Message;
```

```
    }
}
// All COM objects released deterministically when the using blocks exit
```

For long-lived wrappers (fields), call `Dispose()` explicitly in the owning object's `Dispose()` method.

Event-Handler Arguments

A handler method receives its COM objects from the host instead of from a getter you called, but they follow the same rule as everything else: wrap them, `using` them for the duration of the call, and never keep one in a field.

```
public void ToolChanged(string handlerIdent, ICamApiMachiningTool tool)
{
    if (_stopped)
        return;

    using var toolCom = ComWrapper.Create(tool);
    // work through toolCom; released when the method returns
}
```

What `using` cannot do is unsubscribe you. The application close event does not remove your handler, so later events — active project changed, operation added and the rest — can still arrive after it, and by then the host may already be destroying the objects they carry. A handler that re-attaches at that point is attaching to a dying project.

So unregister in the close handler: call `UnregisterHandler`, set a "stopped" flag, and have every handler method return immediately while that flag is set. Stopping timers and detaching the current project is not enough on its own — a controller that stays subscribed is still live.

Checking for Null

```
using var nodeCom = geometryModelCom.InvokeAndWrap(m =>
    m.FindByFullName("Part\\Part1.igs\\Face11", out var status));

if (nodeCom.IsNull)
    throw new Exception("Geometry node not found");
```

Interface Casting

To query whether a COM object also implements another interface, perform the cast inside `Invoke` (or `InvokeAndWrap`) — never against `.It` / `.Instance` outside. The RCW lives in the MTA context, and casting the raw instance from the caller's thread can fail unpredictably.

When you only need the narrow interface for one block of work — cast inside `Invoke` and use the variable locally:

```
mfCom.Invoke(mf =>
{
    if (mf is not ICamApiModelFormerWithHoles mfWithHoles) return;
    mfWithHoles.AddHolesSelected();
});
```

> Warning — **`AsInstanceOf`** and **`InvokeAndWrap(f => f as T)`** load transitive assemblies: the C# cast `f as ITarget` inside a generic lambda causes the .NET runtime to resolve every assembly that `ITarget` transitively depends on. If those assemblies are not in your csproj, the build fails with "assembly not found" errors. The same applies to extension helpers like `featureCom.AsHoleFeature()` that call `InvokeAndWrap(f => f as ICamApiHoleFeature)` internally.
>
> Preferred alternative — inline **`is`** check inside **`Invoke`**: keep the cast inside the lambda. This is resolved at the Delphi/COM level only, with no .NET type resolution beyond the immediate interface:
>
> ```
> // CORRECT — cast stays inside Invoke; no transitive assembly loading
> var result = "";
> featureCom.Invoke(f =>
> {
>     if (f is ICamApiHoleFeature hole)
>     {
>         result = $"Ø{hole.Diameter:F2}";
>         return;
>     }
>     if (f is ICamApiFilletFeature fillet)
>     {
>         result = $"R{fillet.Size:F2}";
>         return;
>     }
> });
>
> // WRONG — InvokeAndWrap(f => f as T) triggers transitive assembly resolution
> using var holeCom = featureCom.AsHoleFeature(); // may fail to build
> ```
>
> Use the inline `is` pattern whenever you need to branch on a sub-interface of a COM object and do not want to pull in extra assembly references.

> Pitfall — **`AsInstanceOf` inside `foreach`**: each call to `AsInstanceOf` creates a new `ComWrapper`. The iterator that produced the original item does not know about this new wrapper and will never dispose it. You must use `using var` explicitly:

```
// CORRECT — AsInstanceOf result wrapped with using
foreach (var itemCom in mf.EnumerateProbingItems())
{
    using var asCycle = itemCom.AsInstanceOf<ICamApiProbingCycle>();
    using var asSurf  = itemCom.AsInstanceOf<ICamApiSurfaceProbingCycle>();
    if (asCycle != null) { /* ... */ }
}

// WRONG — leaks two COM objects per iteration, produces LostObjects_*.txt
foreach (var itemCom in mf.EnumerateProbingItems())
{
    var asCycle = itemCom.AsInstanceOf<ICamApiProbingCycle>(); // no using!
    var asSurf  = itemCom.AsInstanceOf<ICamApiSurfaceProbingCycle>(); //
no using!
}
```

> Pitfall — **`break` inside `foreach`** with a side-allocated wrapper: if you call a method on the iteration variable before `break` and that method returns a new `ComWrapper` you intend to keep, make sure to `Dispose()` the iteration variable itself — the iterator will not reach the `yield` cleanup after `break`:

```
// WRONG — both mf and opCom leak when we break
foreach (var opCom in techCom.EnumerateOperations(rmDesigned))
{
    var mf = opCom.ModelFormerJobAssignment(); // no using — already wrong
    if (mf != null) { foundMf = mf; break; }  // opCom also leaks
    mf?.Dispose();
}

// CORRECT — every COM-returning call gets using var; dispose opCom before break
foreach (var opCom in techCom.EnumerateOperations(rmDesigned))
{
    using var mf = opCom.ModelFormerJobAssignment(); // always using var for COM
    if (mf != null) { foundMf = mf.Transfer(); opCom.Dispose(); break; }
}
```

ListComWrapper<T>

`ListComWrapper<T>` is an `IList<ComWrapper<T>>` that manages the lifetime of a collection of COM wrappers. It is used when you need to build up or hold onto a set of COM objects across multiple statements.

Ownership Transfer on Add

`Add(item)` always calls `item.TransferOwnership()` internally. After the call, the original `item` variable is invalidated (marked disposed); ownership belongs to the list. This prevents double-release.

```
using var curves = new ListComWrapper<IMyGeomCurve>();

foreach (var faceId in faceIds)
{
    // InvokeAndWrap creates a local wrapper — Add transfers ownership into the list
    var curveCom = geometryModelCom.InvokeAndWrap(m =>
        m.FindByFullName(faceId, out _));
    curves.Add(curveCom);
    // curveCom is now empty/disposed — do not use it after this line
}

// Use the list...
foreach (var curveCom in curves)
    curveCom.Invoke(c => c.Selected = true);

// Dispose releases all items
```

Disposal

`Dispose()` iterates every item in the list and calls `Dispose()` on each, then clears the list. Always use `using` on a `ListComWrapper<T>`:

```
using var items = new ListComWrapper<ISomeInterface>();
// ... fill items ...
// items.Dispose() called automatically here
```

`Clear()` also disposes all items and empties the list (useful for reuse within a `using` block).

`Remove(item)` and `RemoveAt(index)` each dispose the removed item.

MTA Threading

ENCY's COM objects are created on MTA (Multi-Threaded Apartment) threads. By default, `ComWrapper<T>` assumes the host process is MTA (`ComWrapperSettings.ApplicationApartmentState ==`

`ApartmentState.MTA`) — this is the correct setting for pure console apps and for in-process plugins.

When you need to show a WPF window (which requires STA), you must set:

```
ComWrapperSettings.ApplicationApartmentState = ApartmentState.STA;
```

This must be set before creating any `ComWrapper<T>` instance (typically at application startup, as in `MainWindow` constructor). When STA mode is active, `ComWrapper<T>` internally marshals COM pointers into an MTA-thread pool via `MtaTaskScheduler`.

MtaTaskScheduler

`MtaTaskScheduler` is a static pool of 8 background threads, all with `ApartmentState.MTA`. It is used internally by `ComWrapper<T>` when the calling thread is STA. You rarely need to use it directly, but it is important to understand:

- All `Invoke` and `InvokeAndWrap` calls on a `ComWrapper<T>` that was created on an STA thread automatically dispatch through `MtaTaskScheduler`.
- The scheduler starts automatically on first use and shuts down cleanly on `AppDomain.ProcessExit`.

If you ever need to execute arbitrary code on an MTA thread (for example, to create a COM object that must be created on MTA), you can use it directly:

```
await MtaTaskScheduler.Run(() =>
{
    // this lambda runs on an MTA thread
    var comObject = CreateSomeComObject();
    // wrap immediately
});
```

Important: never access raw COM interface references inside a WPF event handler or any STA-thread callback directly. Always go through `ComWrapper<T>.Invoke` or `ComWrapper<T>.InvokeAndWrap`, which will dispatch to MTA automatically.

IteratorCom Helpers

ENCY iterators expose a cursor-based API: `Current()`, `MoveToSibling()`, optionally `MoveToChild()` and `MoveToParent()`. The `IteratorHelper` extension methods wrap these into standard .NET `IEnumerable<T>`.

`AsEnumerable` — for scalar (non-COM) node values

Use when `Current()` returns a plain value (string, int, struct) — one that does not itself need disposal:

```
using var opTypesCom = technologistCom.InvokeAndWrap(t => t.OperationTypes);

var ids = opTypesCom.AsEnumerable(
    current:       iter => iter.Current()?.Id,   // returns string
    moveToSibling: iter => iter.Next(),
    moveToChild:   null,
    moveToParent:  null,
    reset:         null
).ToList();
```

The iteration runs on the raw iterator instance without COM wrapping, so it is efficient but requires that the iterator not be disposed during iteration.

`AsComEnumerable` — for COM object nodes

Use when `Current()` returns a COM interface pointer that must itself be wrapped and disposed. The constraint `where TNode : IDisposable` enforces this. Each yielded item is a `ComWrapper<TNode>` (or any `IDisposable` COM wrapper), and each is disposed after its `foreach` body executes:

```
using var iterCom = technologistCom.InvokeAndWrap(t =>
t.GetOperations(TReorderMode.rmReordered, out _));

foreach (using var operationCom in iterCom.AsComEnumerable(
    current:       iter => ComWrapper.Create(iter.Current()),
    moveToSibling: iter => iter.Next(),
    moveToChild:   iter => iter.MoveToChild(),
    moveToParent:  iter => iter.MoveToParent(),
    reset:         null))
{
    var id = operationCom.Invoke(op => op.Id);
    Console.WriteLine(id);
    // operationCom.Dispose() called automatically here
}
```

The difference between the two helpers is ownership: `AsEnumerable` yields raw values and does no COM cleanup; `AsComEnumerable` yields `IDisposable` objects and disposes each one after the loop body.

Tree traversal: both helpers perform depth-first traversal when `moveToChild` and `moveToParent` are non-null. Pass `null` for both to get flat (sibling-only) iteration.

Error Handling in ENCY CAM API .NET Code

TResultStatus

`TResultStatus` is the fundamental error type used throughout the ENCY CAM API. It is a plain struct with two fields:

```
public struct TResultStatus
{
    public TResultStatusCode Code;        // rsOk (0) or rsError (1)
    public string Description;            // human-readable error message; empty
on success
}

public enum TResultStatusCode
{
    rsOk    = 0,
    rsError = 1
}
```

Every method that can fail returns a `TResultStatus` either as an `out` parameter or as the return value. There are no exceptions thrown across the COM boundary — .NET exceptions cannot be marshalled to Delphi COM callers.

The Fundamental Rule: No Exceptions Across COM Boundaries

ENCY calls into your plugin through COM. If your code lets an unhandled exception escape from a COM-exported method, the behaviour is undefined — at best the exception is silently swallowed, at worst it corrupts the host process.

Every COM entry point must catch all exceptions and encode them into the **`out TResultStatus`** parameter.

The standard pattern for a `Run` method:

```
public void Run(IExtensionUtilityContext context, out TResultStatus resultStatus)
{
    resultStatus = default; // Code = rsOk, Description = ""
    try
    {
```

```
            RunInternal(context);
        }
        catch (Exception e)
        {
            resultStatus.Code = TResultStatusCode.rsError;
            resultStatus.Description = e.Message;
        }
    }

    private void RunInternal(IExtensionUtilityContext context)
    {
        // All real logic goes here. Exceptions are free to propagate within this scope
        // because they will be caught by the outer try/catch.
        using var applicationCom = ComWrapper.Create(context.CamApplication);
        // ...
    }
```

This pattern appears in every example:

- `FullWorkflow/FullWorkflow3DProject/project/main/ExtensionFullWorkflow3DProject.cs`
- `UI/ExtensionManageViewNet/project/main/ExtensionManageView.cs`
- `UI/ExtensionUtilityNotifyNet/project/main/ExtensionUtilityNotify.cs`
- `UI/ExtensionUtilityDialogWindowNet/project/main/ExtensionUtilityDialogWindow.cs`

The same rule applies to `IExtensionFactory.Create`, `OnLibraryRegistered`, and `OnLibraryUnRegistered`:

```
    public IExtension? Create(string extensionIdent, out TResultStatus ret)
    {
        try
        {
            ret = default;
            if (extensionIdent == "Extension.Utility.MyPlugin")
                return new MyExtension();
            return null;
        }
        catch (Exception e)
        {
            ret.Code = TResultStatusCode.rsError;
            ret.Description = e.Message;
            return null;
        }
    }
```

Checking Results from API Calls

When calling ENCY API methods that take `ref TExecuteContext` (IPC calls) or `out TResultStatus` parameters, always check the result immediately after the call:

Pattern A: Inline check with exception

```
var ctx = new TExecuteContext();

using var projectCom = applicationCom.InvokeAndWrap(app =>
    app.GetActiveProject(ref ctx));

if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception("GetActiveProject: " + ctx.ResultStatus.Description);
```

Pattern B: out parameter with status return

Some API methods use `out TResultStatus` instead of `ref TExecuteContext`:

```
projectCom.Invoke(project =>
{
    project.SetOperationTool(operationId, toolNumber, out var ret);
    if (ret.Code == TResultStatusCode.rsError)
        throw new Exception("SetOperationTool: " + ret.Description);
});
```

Pattern C: InvokeAndWrap with status tuple

`ComWrapper<T>.InvokeAndWrap` has an overload that accepts a function returning `(TResult, TResultStatus)`. It throws automatically if the status is an error:

```
// Throws if status.Code == rsError — no manual check needed
using var operationCom = technologistCom.InvokeAndWrap(t =>
    (t.CreateOperation("TSTFaceMillingOp", prevId, "", out var status), status));
```

Pattern D: Invoke with status return

Similarly, `Invoke` has an overload for `Func<T, TResultStatus>` that throws on error:

```
// Throws automatically if ret.Code == rsError
technologistCom.Invoke(t =>
    t.CalculateAllOperationsToolpath(true, out var ret));
```

Throwing vs Setting the Status

Inside `RunInternal` (or any helper method called from it), use regular C# exceptions freely. The outer `catch` converts them to `TResultStatus`. The exception message becomes `Description`, so write meaningful messages:

```
private static void SetupOperationTool(
    ComWrapper<ICamApiProject> projectCom,
    string operationId,
    string toolNumber)
{
    projectCom.Invoke(project =>
    {
        project.SetOperationTool(operationId, toolNumber, out var ret);
        if (ret.Code == TResultStatusCode.rsError)
            throw new Exception($"SetOperationTool({operationId},
{toolNumber}): {ret.Description}");
    });
}
```

IExtensionLogger

For non-fatal conditions that do not stop execution, use `IExtensionLogger` (available through extension context) instead of throwing. This writes to the ENCY application log, which is visible in the ENCY log window.

```
public void Run(IExtensionUtilityContext context, out TResultStatus resultStatus)
{
    resultStatus = default;
    try
    {
        using var applicationCom = ComWrapper.Create(context.CamApplication);

        // Log informational messages
        context.Logger?.Log(TLogEventType.leInfo, "MyPlugin", "Starting workflow");

        // Log a warning without stopping
        if (someCondition)
            context.Logger?.Log(TLogEventType.leWarning, "MyPlugin", "Unexpected
state detected");

        RunInternal(context);
    }
```

```
    catch (Exception e)
    {
        // Fatal error — set result status so ENCY shows it to the user
        resultStatus.Code = TResultStatusCode.rsError;
        resultStatus.Description = e.Message;
    }
}
```

`UIDialogs.Notify` is available for showing a popup notification to the user for important informational or warning messages (see `UI/ExtensionUtilityNotifyNet`):

```
UIDialogs.Notify(TLogEventType.leInfo, "My Plugin", "Operation
completed successfully.");
UIDialogs.Notify(TLogEventType.leWarning, "My Plugin", "No geometry selected.");
```

How ENCY Handles Errors Returned by Plugins

When a plugin's `Run` method sets `resultStatus.Code = TResultStatusCode.rsError`, ENCY:

1. Displays the `Description` string in a system notification or error dialog (depending on how the extension was invoked).
2. Logs the error to the application log.
3. Does not crash — the error is treated as a graceful plugin failure.

This means the `Description` field is user-visible. Write it in a language your users understand and include enough context (operation name, file path, etc.) to diagnose the problem without access to source code.

Errors That Appear After Your Plugin Has Returned

`TResultStatus` covers failures *inside* your call. A whole class of failures shows up later — most often when the user closes ENCY: a dialog reporting objects that were not destroyed, a `LostObjects_*.txt` file next to the exe, or an access violation on exit. None of these are error-handling bugs; they are COM lifetime bugs, and the cause is nearly always a COM object obtained without a `using`.

See COM lifetime → Symptoms for the exact messages and how to map a report back to the call that leaked.

Note also that `try/catch` alone does not protect lifetime: an exception thrown between acquiring a COM object and disposing it leaks that object unless the wrapper is held by a `using` (or disposed in a `finally`). This is one more reason the `using var` form is mandatory rather than stylistic.

Helper: Centralised Status Check

For projects with many API calls, a small helper method reduces repetition:

```
private static void Check(TExecuteContext ctx, string operationName)
{
    if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
        throw new Exception($"{operationName} failed:
{ctx.ResultStatus.Description}");
}

private static void Check(TResultStatus status, string operationName)
{
    if (status.Code == TResultStatusCode.rsError)
        throw new Exception($"{operationName} failed: {status.Description}");
}

// Usage:
var ctx = new TExecuteContext();
using var techCom = projectCom.InvokeAndWrap(p => p.GetTechnologist(ref ctx));
Check(ctx, nameof(ICamApiProject.GetTechnologist));

projectCom.Invoke(p => { p.SaveClData(path, iter, out var s); Check(s,
"SaveClData"); });
```

UI Patterns for ENCY .NET Plugins

ENCY plugins run inside a Delphi host that uses VCL for its own UI and COM for the .NET bridge. This creates threading constraints that determine which UI pattern is appropriate. There are three distinct patterns, each suited to different use cases.

> Patterns 2 and 3 open a window you own, and ENCY does not paint it. To match the active host theme, see theming-plugin-windows.md. Pattern 1 needs nothing — the inspector dialog is rendered by the host.

Pattern 1: Inspector Dialog (Recommended for Parameter Input)

The inspector dialog uses `CamApiInspectorWindow` and `SimplePropIterator` from the `CAMAPI.UIDialogs` package. It renders a native ENCY-style property grid that integrates visually with the host application. No WPF or XAML is required.

Reference example:
`UI/ExtensionUtilityDialogWindowNet/project/main/ExtensionUtilityDialogWindow.cs`

When to Use

- Collecting parameters from the user before running an operation (operation setup, export settings, etc.)
- Any scenario where you want the dialog to look and feel like a native ENCY dialog

Complete Example

```
using CAMAPI.UIDialogs;
using CAMAPI.UIDialogs.DotnetHelper;

public void Run(IExtensionUtilityContext context, out TResultStatus resultStatus)
{
    resultStatus = default;
    try
    {
        // Container for the values you want to collect
        var values = new MyDialogValues();

        // Create the inspector window and property iterator
        using var window = new CamApiInspectorWindow();
        using var props = new SimplePropIterator();
```

```
// --- String property ---
props.AddStringProp(
    "Output file name",
    () => values.FileName,
    v => values.FileName = v);

// --- Enum with integer index ---
props.AddEnumIndexedProp(
    "Tolerance",
    () => values.ToleranceIndex,
    v => values.ToleranceIndex = v,
    list =>
    {
        list.Add("Coarse",  "");
        list.Add("Normal",  "");
        list.Add("Fine",    "");
    });

// --- Enum with string ID ---
props.AddEnumIdProp(
    "Output format",
    () => values.FormatId,
    v => values.FormatId = v,
    list =>
    {
        list.Add("step", "STEP (*.step)", "");
        list.Add("igs",  "IGES (*.igs)",  "");
        list.Add("stl",  "STL (*.stl)",   "");
    });

// --- Nested group ---
var advancedGroup = props.AddComplexProp("Advanced");
advancedGroup.AddStringProp(
    "Custom postprocessor path",
    () => values.PostProcessorPath,
    v => values.PostProcessorPath = v);

// Attach properties to the window
window.SetPropIterator(props);

// Configure buttons
window.SetButtons(MessageBoxHelper.BuildButtons(
    TUIButtonType.btOk,
    TUIButtonType.btCancel));
```

```
        // Show and handle the result
        switch (window.Show())
        {
            case TUIButtonType.btOk:
                RunWithValues(context, values);
                break;

            case TUIButtonType.btCancel:
                // User cancelled — not an error; just return rsOk with no action
                break;
        }
    }
    catch (Exception e)
    {
        resultStatus.Code = TResultStatusCode.rsError;
        resultStatus.Description = e.Message;
    }
}

private class MyDialogValues
{
    public string FileName        { get; set; } = "output";
    public int    ToleranceIndex  { get; set; } = 1;   // "Normal"
    public string FormatId        { get; set; } = "step";
    public string PostProcessorPath { get; set; } = "";
}
```

SimplePropIterator API Reference

Method	Purpose
`AddStringProp(label, getter, setter)`	Text input field
`AddEnumIndexedProp(label, getter, setter, fillList)`	Drop-down by integer index (0-based)
`AddEnumIdProp(label, getter, setter, fillList)`	Drop-down by string ID
`AddComplexProp(label)`	Returns a sub-iterator for grouped/nested properties

For `AddEnumIndexedProp`, `fillList` receives a list object and you call `list.Add(displayName, hint)`. For `AddEnumIdProp`, `fillList` receives `list.Add(id, displayName, hint)`.

The `getter` and `setter` lambdas are called live as the user edits the dialog — you can read the current selection at any time from the values object after `Show()` returns.

Pattern 2: Modal WPF or WinForms Window

Use a dedicated STA thread to host a WPF/WinForms dialog that the caller waits for before continuing. This is appropriate when you need a rich custom UI (forms, charts, file pickers, etc.) and the result must be available synchronously.

Reference example: `UI/ExtensionUtilityNotifyNet/project/main/ExtensionUtilityNotify.cs`

The Threading Constraint

ENCY's .NET bridge runs on an MTA thread. WPF requires STA. You cannot `ShowDialog()` from an MTA thread. The solution is to create a dedicated STA thread, block on it, and capture the result via a closure.

```
private static string? AskUserForText()
{
    string? result = null;

    var thread = new Thread(() =>
    {
        var window = new TextInputWindow();
        if (window.ShowDialog() == true)
            result = window.UserInput;
        // Window is destroyed and STA dispatcher stops here
    });

    thread.SetApartmentState(ApartmentState.STA);
    thread.Start();
    thread.Join(); // Block until the window closes

    return result; // Safe to read after Join()
}
```

Full usage inside a `Run` method:

```
public void Run(IExtensionUtilityContext context, out TResultStatus resultStatus)
{
    resultStatus = default;
    try
    {
        var userText = AskUserForText();
        if (userText == null)
```

```
            return; // User cancelled

        UIDialogs.Notify(TLogEventType.leInfo, "My Plugin", $"You
entered: {userText}");
    }
    catch (Exception e)
    {
        resultStatus.Code = TResultStatusCode.rsError;
        resultStatus.Description = e.Message;
    }
}
```

Critical Caveat: No COM Access Inside the STA Thread

COM objects obtained from the ENCY API cannot be accessed directly inside the STA thread's lambda. COM apartment rules prohibit cross-apartment access without proper marshalling. If you need to display data retrieved from COM in the dialog, read it before spawning the STA thread and pass it as plain .NET values:

```
// Read COM data on the calling (MTA) thread
var operationNames = GetOperationNames(context); // returns List<string>

// Pass plain data to the STA thread
string? selected = null;
var thread = new Thread(() =>
{
    var window = new PickOperationWindow(operationNames); // receives List<string>
    if (window.ShowDialog() == true)
        selected = window.SelectedName;
});
thread.SetApartmentState(ApartmentState.STA);
thread.Start();
thread.Join();
```

Pattern 3: Non-Modal WPF Window

Use a non-modal window when you want to display a persistent panel (viewport controller, live data viewer, etc.) that stays open while the user continues working in ENCY.

> Dedicated guide: non-modal-window.md — step-by-step with the canonical `ExtensionManageViewNet` example. Start there; the summary below is the short reference.

Reference example: `UI/ExtensionManageViewNet/project/main/ExtensionManageView.cs`

This pattern requires implementing `IExtensionLazyUnloadable` so ENCY knows not to unload the plugin DLL while the window is open.

IExtensionLazyUnloadable

When ENCY wants to unload an extension, it checks `IExtensionLazyUnloadable.CanUnload`. Return `false` to defer unloading. Set it to `true` (from the window-closed callback) when the window closes.

Using ExtensionWindowLazyUnloadable (Recommended)

`ExtensionWindowLazyUnloadable` from `CAMAPI.DotnetHelper` implements `IExtensionLazyUnloadable` and wraps `WindowHelper.ShowStaWindow`, handling the `CanUnload` flag automatically:

```
using CAMAPI.DotnetHelper;
using CAMAPI.Extensions;
using CAMAPI.ResultStatus;

public class MyExtension : IExtension, IExtensionUtility, IExtensionLazyUnloadable
{
    private readonly ExtensionWindowLazyUnloadable _windowManager = new();

    public IExtensionInfo? Info { get; set; }

    // Forward IExtensionLazyUnloadable.CanUnload to the helper
    public bool CanUnload
    {
        get => _windowManager.CanUnload;
        set { }
    }

    public void Run(IExtensionUtilityContext context, out
TResultStatus resultStatus)
    {
        resultStatus = default;
        try
        {
            using var applicationCom = ComWrapper.Create(context.CamApplication);

            // Get the owner window handle for parenting
            _windowManager.SetOwnerHandle(applicationCom);

            // Obtain a COM object to pass into the window.
            // Note: DO NOT wrap in 'using' here — transfer ownership to the window
            var viewPortCom = applicationCom.InvokeAndWrap(app =>
                app.MainForm).InvokeAndWrap(f => f.MainViewPort);
```

```
            // Show the window on a background STA thread.
            // The window's constructor receives the COM wrapper and owns
its lifetime.
            // The window must implement IDisposable and call
viewPortCom.Dispose() there.
            _windowManager.ShowWindow(
                () => new MyViewWindow(viewPortCom));
            // Run returns immediately; the window lives on its own thread
        }
        catch (Exception e)
        {
            resultStatus.Code = TResultStatusCode.rsError;
            resultStatus.Description = e.Message;
        }
    }
}
```

Using WindowHelper Directly

If you need more control, use `WindowHelper.ShowStaWindow` directly:

```
public class ExtensionManageView : IExtension, IExtensionUtility,
IExtensionLazyUnloadable
{
    private bool _canUnload;

    public IExtensionInfo? Info { get; set; }
    public bool CanUnload { get => _canUnload; set { } }

    public void Run(IExtensionUtilityContext context, out
TResultStatus resultStatus)
    {
        resultStatus = default;
        try
        {
            using var applicationCom = ComWrapper.Create(context.CamApplication);
            using var mainFormCom = applicationCom.InvokeAndWrap(app
=> app.MainForm);
            var ownerHandle = mainFormCom.MainWindowHandle();

            // Transfer ownership of the view port COM wrapper to the window
            var viewPortCom = mainFormCom.InvokeAndWrap(f => f.MainViewPort);

            WindowHelper.ShowStaWindow(
                ownerHandle,
```

```
                createWindow: () => new ViewControlWindow(viewPortCom),
                onClosed:     () => _canUnload = true);
            // Run returns here; window is alive on its STA thread
        }
        catch (Exception e)
        {
            resultStatus.Code = TResultStatusCode.rsError;
            resultStatus.Description = e.Message;
        }
    }
}
```

`WindowHelper.ShowStaWindow` signature:

```
public static void ShowStaWindow<TWindow>(
    long ownerHandle,        // Win32 HWND of the ENCY main window (for parenting)
    Func<TWindow> createWindow,     // called on the STA thread to construct
the window
    Action? onClosed = null,        // called after window.Dispose() and
dispatcher shutdown
    Action<Exception>? onException = null) // called if an exception occurs on the
STA thread
    where TWindow : Window, IDisposable
```

`WindowHelper.ShowStaWindow` also sets `ComWrapperSettings.ApplicationApartmentState = ApartmentState.STA` automatically.

Window Lifetime and COM Disposal

The window class must implement `IDisposable`. Any COM wrappers that were passed to the window and are not owned elsewhere must be disposed there:

```
public class ViewControlWindow : Window, IDisposable
{
    private readonly ComWrapper<ICamApiViewPort> _viewPortCom;

    public ViewControlWindow(ComWrapper<ICamApiViewPort> viewPortCom)
    {
        _viewPortCom = viewPortCom; // takes ownership
        InitializeComponent();
    }

    public void Dispose()
    {
```

```
            _viewPortCom.Dispose();
        }
    }
```

`WindowHelper.ShowStaWindow` calls `window.Dispose()` automatically when the window's `Closed` event fires.

CanUnload Behaviour

- While the window is open: `CanUnload` returns `false`. ENCY defers plugin unloading.
- After the window closes (the `onClosed` callback fires): `CanUnload` returns `true`. ENCY may now unload the plugin DLL.

If `IExtensionLazyUnloadable` is not implemented, ENCY may unload the DLL while the window is still on screen, causing an immediate crash.

UIDialogs Static Helpers

`UIDialogs` (in `CAMAPI.UIDialogs.DotnetHelper`) provides lightweight helper methods for common UI interactions without requiring a full window:

```
using CAMAPI.UIDialogs.DotnetHelper;
using CAMAPI.Logger;

// Show a notification popup (info, warning, or error)
UIDialogs.Notify(TLogEventType.leInfo,    "My Plugin", "Toolpath
calculated successfully.");
UIDialogs.Notify(TLogEventType.leWarning, "My Plugin", "No geometry was selected.");
UIDialogs.Notify(TLogEventType.leError,   "My Plugin", "File not found: output.nc");
```

For message boxes with button choices and for file/folder selection dialogs, obtain a helper instance:

```
using var helper = UIDialogs.CreateHelper();

// Message box with OK / Cancel
var result = helper.ShowMessageBox(
    "Confirm",
    "This will overwrite existing toolpaths. Continue?",
    MessageBoxHelper.BuildButtons(TUIButtonType.btOk, TUIButtonType.btCancel));

if (result == TUIButtonType.btCancel)
    return;
```

```
    // Open file dialog
    var filePath = helper.ShowOpenFileDialog(
        "Select postprocessor",
        "ENCY postprocessor (*.sppx)|*.sppx|All files (*.*)|*.*");

    if (string.IsNullOrEmpty(filePath))
        return;
```

`UIDialogs.CreateHelper()` returns an `IDisposable`; always use `using`.

Non-Modal WPF Window in an ENCY Plugin

A non-modal window is a WPF panel that opens when the user clicks a toolbar button and stays on screen while the user continues working in ENCY. The panel can read and drive ENCY live — switching views, editing parameters, reacting to selection.

This is the pattern ENCY users ask about most often. The canonical example is `UI/ExtensionManageViewNet/project/main/ExtensionManageView.cs` — a viewport controller with Face / Edge / Corner buttons that rotate the ViewCube while the user keeps editing the part. Copy from it as a starting point.

The four moving parts

A non-modal plugin is made of four files:

File	Role
`ExtensionFactory.cs`	Creates the extension instance when ENCY loads the DLL. Standard boilerplate — no non-modal logic here.
`ExtensionXxx.cs`	`IExtension, IExtensionUtility, IExtensionLazyUnloadable`. Handles `Run()` and `CanUnload`.
`XxxWindow.xaml` / `XxxWindow.xaml.cs`	A WPF `Window` that also implements `IDisposable` and owns any COM wrappers it uses.
`*.settings.json`	Declares the extension to ENCY.

`IExtensionLazyUnloadable` is the crucial piece. Without it ENCY may unload the plugin DLL while the window is still on screen — the next click crashes the host.

What has to work correctly

1. STA thread — WPF requires STA; ENCY's .NET bridge runs MTA. The helper `WindowHelper.ShowStaWindow` spins up a dedicated STA thread and runs a `Dispatcher` loop on it.
2. Window parenting — the window should sit on top of the ENCY main window, and minimize/restore with it. That requires a `WindowInteropHelper` with the ENCY main window's HWND as owner. `ShowStaWindow` does this for you if you give it the HWND.
3. COM wrapper ownership transfer — any `ComWrapper<T>` you need inside the window must not be `using`-scoped in `Run()`. Construct it, hand it to the window's constructor, and dispose it in the window's `Dispose()`. `Run()` returns immediately — if you `using` the wrapper, it gets disposed before the user clicks anything.

4. **`CanUnload`** flag — return `false` while the window lives, flip to `true` in the `onClosed` callback.
5. **`window.Dispose()`** on close — `ShowStaWindow` calls it for you. Dispose every COM wrapper the window holds.

Minimal skeleton — manual approach

Follows `ExtensionManageView.cs` verbatim. Copy this if you want to see every moving piece.

```
using CAMAPI.Application;
using CAMAPI.DotnetHelper;
using CAMAPI.Extensions;
using CAMAPI.ResultStatus;

public class MyExtension : IExtension, IExtensionUtility, IExtensionLazyUnloadable
{
    private bool _canUnload;

    public IExtensionInfo? Info { get; set; }

    public void Run(IExtensionUtilityContext context, out
TResultStatus resultStatus)
    {
        resultStatus = default;
        try
        {
            using var applicationCom         =
ComWrapper.Create(context.CamApplication);
            using var applicationMainFormCom = applicationCom.MainForm();
            var mainWindowHandle             =
applicationMainFormCom.MainWindowHandle();

            // NOTE: no 'using' — ownership transfers to the window.
            var viewPortCom = applicationMainFormCom.MainViewPort();

            WindowHelper.ShowStaWindow(
                mainWindowHandle,
                createWindow: () => new MyWindow(viewPortCom),
                onClosed:     () => _canUnload = true);
            // Run returns immediately; window lives on its own STA thread.
        }
        catch (Exception e)
        {
            resultStatus.Code        = TResultStatusCode.rsError;
            resultStatus.Description = e.Message;
        }
```

```
    }

    /// <summary>CanUnload returns true only after the window has been
closed.</summary>
    public bool CanUnload
    {
        get => _canUnload;
        set { }
    }
}
```

The window must implement `IDisposable` and dispose every COM wrapper it was handed:

```
public partial class MyWindow : Window, IDisposable
{
    private readonly ComWrapper<ICamApiViewPort> _viewPortCom;

    public MyWindow(ComWrapper<ICamApiViewPort> viewPortCom)
    {
        _viewPortCom = viewPortCom; // takes ownership — do NOT dispose twice
        InitializeComponent();
    }

    private void Face_Click(object sender, RoutedEventArgs e)
    {
        if (sender is not Button { Tag: string tagStr }) return;
        if (!Enum.TryParse<TViewCubeRotateMode>(tagStr, out var mode)) return;

        try
        {
            _viewPortCom.ZoomAll(true);          // helper extension
            using var cubeCom = _viewPortCom.GetCube();
            cubeCom.Rotate(mode);
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message, "Error", MessageBoxButton.OK,
MessageBoxImage.Error);
        }
    }

    public void Dispose() => _viewPortCom.Dispose();
}
```

`WindowHelper.ShowStaWindow` calls `window.Dispose()` automatically when the window's `Closed` event fires, then invokes `onClosed`.

Minimal skeleton — helper approach

`ExtensionWindowLazyUnloadable` from `CAMAPI.DotnetHelper` wraps the HWND + `ShowStaWindow` + `_canUnload` bookkeeping into one object. Same result, less boilerplate:

```
public class MyExtension : IExtension, IExtensionUtility, IExtensionLazyUnloadable
{
    private readonly ExtensionWindowLazyUnloadable _windowManager = new();

    public IExtensionInfo? Info { get; set; }

    public bool CanUnload
    {
        get => _windowManager.CanUnload;
        set { }
    }

    public void Run(IExtensionUtilityContext context, out
TResultStatus resultStatus)
    {
        resultStatus = default;
        try
        {
            using var applicationCom = ComWrapper.Create(context.CamApplication);
            _windowManager.SetOwnerHandle(applicationCom);

            using var mainFormCom = applicationCom.MainForm();
            var viewPortCom = mainFormCom.MainViewPort();   // again, no 'using'

            _windowManager.ShowWindow(() => new MyWindow(viewPortCom));
        }
        catch (Exception e)
        {
            resultStatus.Code        = TResultStatusCode.rsError;
            resultStatus.Description = e.Message;
        }
    }
}
```

Prefer this form when the extension has only one non-modal window.

`WindowHelper.ShowStaWindow` signature

```
public static void ShowStaWindow<TWindow>(
    long ownerHandle,                          // HWND of the ENCY main window
    Func<TWindow> createWindow,                // called on the STA thread
    Action? onClosed = null,                   // called after window.Dispose() +
dispatcher shutdown
    Action<Exception>? onException = null)   // fires if an exception is raised on
the STA thread
    where TWindow : Window, IDisposable
```

- Sets `ComWrapperSettings.ApplicationApartmentState = ApartmentState.STA` on entry.
- Starts a new background STA thread and runs `Dispatcher.Run()` inside it.
- On the `Closed` event: `window.Dispose()`, then `Dispatcher.InvokeShutdown()`, then `onClosed?.Invoke()`.
- Exceptions thrown on the STA thread are routed to `onException` (or shown in a `MessageBox` if no handler was supplied).

Common mistakes

- Wrapping the transferred COM in **`using`** — it gets disposed the moment `Run()` returns, and the window clicks onto a dead reference. Build it without `using`; the window's `Dispose()` is the single point of release.
- Not implementing **`IExtensionLazyUnloadable`** — ENCY may unload the DLL while the window is on screen; the next event crashes the host.
- Returning **`CanUnload = true`** before the window is closed — same outcome.
- Calling COM from the UI thread without **`Invoke`** — the COM objects were created in MTA; direct access from the STA thread is unsafe. Always go through `ComWrapper.Invoke` / extension methods, never through `.Instance` / `.It`.
- Disposing **`viewPortCom`** both in **`Run()`** (with **`using`**) and in the window's **`Dispose()`** — double-release. Keep disposal in the window only.

Settings JSON and factory

The plugin registration is identical to any other utility extension. `*.settings.json`:

```
{
    "name": "Utility to manage main view window by C#",
    "description": "Utility to manage main view window by C#",
    "module_path": "${extensionJsonFolder}\\MyExtension.dll",
    "extensions": [
```

```
        {
            "Utility": {
                "name": "Utility to manage main view window by C#",
                "id": "Extension.UI.MyView.Net"
            }
        }
    ]
}
```

`ExtensionFactory.cs` — standard boilerplate, no non-modal-specific code:

```
namespace CAMAPI;

using Extensions;
using ResultStatus;

public class ExtensionFactory : IExtensionFactory
{
    public void OnLibraryRegistered(IExtensionFactoryContext context, out
TResultStatus ret)   { ret = default; }
    public void OnLibraryUnRegistered(IExtensionFactoryContext context, out
TResultStatus ret) { ret = default; }

    public IExtension? Create(string extensionIdent, out TResultStatus ret)
    {
        ret = default;
        try
        {
            if (extensionIdent == "Extension.UI.MyView.Net")
                return new MyExtension();
            throw new Exception("Unknown extension identifier: " + extensionIdent);
        }
        catch (Exception e)
        {
            ret.Code        = TResultStatusCode.rsError;
            ret.Description = e.Message;
            return null;
        }
    }
}
```

Where to read next

- Full example project: UI/ExtensionManageViewNet/ — `.xaml` + `.xaml.cs` + `Extension…View.cs` + `ExtensionFactory.cs` + `*.settings.json`.
- Threading rules and COM lifetime: com-lifetime.md — why `Invoke` / `InvokeAndWrap` / `.IsNull` are mandatory and why `.Instance` / `.It` / `AsInstanceOf` are not used.
- Other UI patterns (inspector dialog, modal WPF window): ui-patterns.md.

Applying the Host Theme to a Plugin Window

The host application exposes its active UI palette read-only through `ICamApiTheme`. It does not paint plugin windows — a plugin that opens its own window is responsible for matching the host look itself.

This page is about applying the palette. For reading it — `Name` / `Kind` / `IsDark` / `GetColor` and the `TCamApiColorKind` slot list — see api/ui.md § ICamApiTheme.

Rules

1. Read the palette once, when the window opens, and treat it as best-effort: if the host does not expose a theme, keep the framework defaults and never throw.
2. WPF — put the palette into resource brushes referenced with `DynamicResource`. That is the whole job.
3. WinForms — assigning `BackColor` is not enough. Controls painted by Windows visual styles silently ignore it; each one needs its own opt-out (table below).
4. Never try to strip visual styles with `SetWindowTheme(handle, "", "")`. It crashes some controls.
5. Prefer controls that honour `BackColor`. `NumericUpDown` and `TabControl` cannot be fully themed — pick a different control instead of fighting them.
6. Scrollbars are themed by *name*, not by colour, and often only after their handle exists.
7. The window title bar is a DWM call — cosmetic, wrap in `try/catch`.

---

1. Read the palette once, best-effort

Read every colour inside a single `Invoke` on the application wrapper and release the theme object there: the host creates a fresh theme object per getter call, so releasing it on its own apartment frees it deterministically.

```
private static Palette ReadPalette(ICamApiApplication app)
{
    var theme = app.Theme;
    if (theme is null)
        return default;                // headless / kernel-only host build

    try
    {
        return new Palette(true,
            FromTColor(theme.GetColor(TCamApiColorKind.ckColorWindowBackground)),
            FromTColor(theme.GetColor(TCamApiColorKind.ckColorPanelBackground)),
            FromTColor(theme.GetColor(TCamApiColorKind.ckColorText)),
            FromTColor(theme.GetColor(TCamApiColorKind.ckColorBorder)),
            FromTColor(theme.GetColor(TCamApiColorKind.ckColorBtnBackground)),
            FromTColor(theme.GetColor(TCamApiColorKind.ckColorAccent)));
```

```
    }
    finally
    {
        if (Marshal.IsComObject(theme))
            Marshal.ReleaseComObject(theme);
    }
}
```

The call site swallows everything — a missing theme is a normal condition, not an error:

```
Palette palette;
try
{
    using var appCom = ComWrapper.Create(application);
    palette = appCom.Invoke(ReadPalette);
}
catch
{
    return;                              // keep the framework's default colours
}
if (!palette.Ok)
    return;
```

Colour conversion

`GetColor` returns a Delphi `TColor` (`0x00BBGGRR`) — the low byte is red. Win32 `COLORREF` (needed for the DWM calls below) uses the same layout, so it is rebuilt from the parsed colour:

```
private static Color FromTColor(int tcolor)
    => Color.FromArgb(tcolor & 0xFF, (tcolor >> 8) & 0xFF, (tcolor >> 16) & 0xFF);

private static int ToColorRef(Color c)
    => c.R | (c.G << 8) | (c.B << 16);
```

Dark or light

`ICamApiTheme.IsDark` answers this directly. If you already have the window background colour, a luminance test on it avoids a second COM call and stays correct for host themes the SDK does not know yet:

```
private static bool IsDark(Color c)
    => 0.299 * c.R + 0.587 * c.G + 0.114 * c.B < 128;
```

2. WPF: brushes and `DynamicResource`

WPF has none of the problems below. Declare the palette as brushes in a resource dictionary, style your controls against those brushes with `DynamicResource`, and at runtime replace only the brush *values* — every reference repaints itself:

```
<SolidColorBrush x:Key="ThemeWindowBrush" Color="#1E1E1E"/>
<SolidColorBrush x:Key="ThemeTextBrush"   Color="#F1F1F1"/>

<Style TargetType="TextBlock">
    <Setter Property="Foreground" Value="{DynamicResource ThemeTextBrush}"/>
</Style>
```

```
var res = window.Resources;
res["ThemeWindowBrush"] = Brush(palette.Window);
res["ThemeTextBrush"]   = Brush(palette.Text);
```

The dictionary values double as the fallback palette when the host exposes no theme, so one style set covers dark, light and any future host theme.

One WPF caveat: the stock `Button` and `ListViewItem` templates paint their own chrome and ignore `Background` in the hover/selected states. Retemplate them (a `Border` bound to `{TemplateBinding Background}`) if you need those states themed.

In-repo reference: `UI/SelectedFeaturesViewerNet/project/main/Themes/PluginTheme.xaml` with `Service/ThemeService.cs`, and `FullWorkflow/PartCalibrationWorkflowNet/project/main/Service/ThemeService.cs`.

Everything from here on is WinForms / native visual-styles specific.

---

3. WinForms: visual styles override `BackColor`

The failure mode is deceptive: the assignment succeeds and the property reads back correctly (`BackColor` really is `#1E2028`), but the control still renders light, because it is painted by the Windows visual-styles renderer rather than from its own colour properties. There is no exception and no warning.

Control	Does **`BackColor`** apply?	What it takes
`Panel`, `Label`, `GroupBox`, `TextBox`, `CheckBox`, `ListBox`	yes	set `BackColor` / `ForeColor`

Control	Does `BackColor` apply?	What it takes
`Button`	no	`FlatStyle = FlatStyle.Flat` + `FlatAppearance.BorderColor`
`ComboBox` with `DropDownStyle = DropDownList`	no	`FlatStyle = FlatStyle.Flat`
`DataGridView`	headers no, body yes	`EnableHeadersVisualStyles = false` + explicit cell styles
`NumericUpDown`	spin buttons never	use a different control (see rule 4)
`TabControl`	tab strip never	use a different control (see rule 4)

Walk the tree recursively and special-case the controls that need the opt-out:

```
private static void ApplyToChildren(Control parent, Palette p, bool dark)
{
    foreach (Control control in parent.Controls)
    {
        switch (control)
        {
            case Button button:
                button.FlatStyle = FlatStyle.Flat;
                button.FlatAppearance.BorderColor = p.Border;
                button.BackColor = p.Button;
                button.ForeColor = p.Text;
                break;

            // a themed drop-down list ignores BackColor until its style is flat
            case ComboBox combo:
                combo.FlatStyle = FlatStyle.Flat;
                combo.BackColor = p.Panel;
                combo.ForeColor = p.Text;
                break;

            case DataGridView grid:
                ApplyToGrid(grid, p, dark);
                break;

            default:
```

```
                control.BackColor = p.Panel;
                control.ForeColor = p.Text;
                break;
        }

        ApplyToChildren(control, p, dark);
    }
}
```

`DataGridView` needs five separate assignments — the four style objects do not inherit from each other, and `BackgroundColor` is only the empty area to the right of the last column:

```
private static void ApplyToGrid(DataGridView grid, Palette p, bool dark)
{
    grid.EnableHeadersVisualStyles = false;     // otherwise headers stay
system-coloured
    grid.BackgroundColor = p.Panel;
    grid.GridColor = p.Border;
    grid.ColumnHeadersDefaultCellStyle.BackColor = p.Button;
    grid.ColumnHeadersDefaultCellStyle.ForeColor = p.Text;
    grid.DefaultCellStyle.BackColor = p.Panel;
    grid.DefaultCellStyle.ForeColor = p.Text;
    grid.DefaultCellStyle.SelectionBackColor = p.Accent;
    grid.DefaultCellStyle.SelectionForeColor = p.Text;
}
```

---

4. Anti-pattern: do not strip visual styles

`SetWindowTheme(handle, "", "")` detaches a control from the visual-styles engine. It looks like the universal fix for anything that ignores `BackColor`. Do not use it. WinForms controls that own their painting still go through `VisualStyleRenderer`, which now has nothing to render with:

```
InvalidOperationException: Visual Style handle creation operation did not succeed
   at UpDownBase.UpDownButtons.OnPaint
```

The crash happens on the first repaint, not at the call — so it surfaces as a random failure later.

The correct fix is control selection, not force. When designing a plugin window, pick controls that honour **`BackColor`**:

- `NumericUpDown` → a plain `TextBox`, parsing the value when the user presses the action button.
- `TabControl` → a `SplitContainer` (or your own header buttons).

The empty-string form is the problem. Passing a real theme *name* (next section) is the supported use of the same API.

---

5. Scrollbars

Scrollbars are not coloured — they are re-themed by name, `"DarkMode_Explorer"` or `"Explorer"`:

```
[DllImport("uxtheme.dll", CharSet = CharSet.Unicode)]
private static extern int SetWindowTheme(IntPtr hwnd, string? subAppName,
string? subIdList);
```

Three details:

(a) The handle may not exist yet. At the moment the window is shown, a grid's scrollbars usually have no native handle (a typical run finds four scrollbars and can theme one). Subscribe for the rest:

```
foreach (var scrollBar in FindScrollBars(form))
{
    if (scrollBar.IsHandleCreated)
    {
        SetWindowTheme(scrollBar.Handle, theme, null);
        continue;
    }

    scrollBar.HandleCreated += (sender, _) =>
SetWindowTheme(((Control)sender!).Handle, theme, null);
}
```

(b) **`DataGridView`** scrollbars are private fields — reach them by reflection:

```
foreach (var field in grid.GetType().GetFields(BindingFlags.NonPublic
| BindingFlags.Instance))
{
    if (typeof(ScrollBar).IsAssignableFrom(field.FieldType) && field.GetValue(grid)
is ScrollBar sb)
        yield return sb;
}
```

(c) Theme the grid handle itself as well — that is what removes the white square in the corner between the horizontal and vertical scrollbars:

```
SetWindowTheme(grid.Handle, dark ? "DarkMode_Explorer" : "Explorer", null);
```

Run this pass after the window is realized (`form.BeginInvoke(...)` from the theming entry point) and wrap it in `try`/`catch` — themed scrollbars are optional chrome.

---

6. Window title bar (Windows)

The caption is owned by the desktop window manager, not by the framework:

```
private const int DWMWA_USE_IMMERSIVE_DARK_MODE = 20;
private const int DWMWA_CAPTION_COLOR = 35;
private const int DWMWA_TEXT_COLOR = 36;

[DllImport("dwmapi.dll")]
private static extern int DwmSetWindowAttribute(IntPtr hwnd, int attribute, ref int
value, int size);


var darkFlag = dark ? 1 : 0;
DwmSetWindowAttribute(form.Handle, DWMWA_USE_IMMERSIVE_DARK_MODE, ref
darkFlag, sizeof(int));

var caption = ToColorRef(p.Window);
DwmSetWindowAttribute(form.Handle, DWMWA_CAPTION_COLOR, ref caption, sizeof(int));
var text = ToColorRef(p.Text);
DwmSetWindowAttribute(form.Handle, DWMWA_TEXT_COLOR, ref text, sizeof(int));
```

`DWMWA_USE_IMMERSIVE_DARK_MODE` gives a dark caption. `DWMWA_CAPTION_COLOR` / `DWMWA_TEXT_COLOR` paint it in the exact theme colour on Windows 11 build 22000+ and are silently ignored on Windows 10. All three are cosmetic: wrap them so an unsupported OS build cannot take the window down.

---

7. Excluding controls from the walk

Some controls must keep their own colours — legend swatches, status indicators, anything whose colour carries meaning. Mark them and skip them in the recursive walk:

```
public const string SkipTag = "no-theme";

// at construction
var swatch = new Panel { BackColor = Color.OrangeRed, Tag = FormTheme.SkipTag };
```

```
// in the walk
if (control.Tag as string == SkipTag)
    continue;
```

Note that `continue` skips the subtree, not just the control — which is normally what you want for a self-coloured container.

Checklist

- [ ] Palette read once on window load, inside one `Invoke`, wrapped in `try`/`catch`
- [ ] Missing theme leaves the window on framework defaults without an error
- [ ] `TColor` decoded with the low byte as red
- [ ] WinForms: `FlatStyle.Flat` on every `Button` and `ComboBox`
- [ ] WinForms: `EnableHeadersVisualStyles = false` and all cell styles on every `DataGridView`
- [ ] No `NumericUpDown` / `TabControl` in the window
- [ ] No `SetWindowTheme(handle, "", "")` anywhere
- [ ] Scrollbars themed after realization, including the deferred `HandleCreated` ones
- [ ] Title-bar DWM calls wrapped in `try`/`catch`

Writing a macro

A macro is a small extension that ENCY builds and runs on demand. The default language is .NET (a compiled DLL); a script language is also available. Either way the macro implements one method, `ICamApiMacro.Run`, and receives a live application context to do its work.

This folder is only about authoring the macro body — the code that runs *inside* ENCY. It does not cover the actual CAM operations you call from that code: for those (operations, geometry, tools, NC, …) use the regular API reference.

If you want to…	Read
Understand the macro class and run context	this file
Write the `Run` body and call CAM commands	writing-macro.md
Let the user override values before running	macro-params.md
Report progress / enable step highlighting	notify-step.md
Call actual CAM commands (operations, geometry, …)	../api/ — the regular API docs (start at project.md, geometry.md, …)
Manage / build / run macros from a host plugin or external app	../api/application.md (in-process) · ../ipc/application.md (IPC)
COM object lifetime (`ComWrapper`, MTA)	../general/com-lifetime.md
Returning errors across the COM boundary	../general/error-handling.md

Anatomy of a macro

A built macro is a normal extension DLL with two classes:

- **`ExtensionFactory`** — must be in the `CAMAPI` namespace and named `ExtensionFactory`; ENCY finds it by convention and asks it to create the macro instance.
- The macro class — implements `IExtension` + `ICamApiMacro`. Its `Run` method is the entry point.

```
namespace CAMAPI;

public class ExtensionFactory : IExtensionFactory
{
    public void OnLibraryRegistered(IExtensionFactoryContext context, out
TResultStatus ret) => ret = default;
```

```
    public void OnLibraryUnRegistered(IExtensionFactoryContext context, out
TResultStatus ret) => ret = default;

    public IExtension? Create(string extensionIdent, out TResultStatus ret)
    {
        ret = default;
        if (extensionIdent == "ExtensionDummy")
            return new ExtensionDummy();
        throw new Exception("Unknown extension identifier: " + extensionIdent);
    }
}
```

The macro itself:

```
public class ExtensionDummy : IExtension, ICamApiMacro
{
    public IExtensionInfo? Info { get; set; }

    public void Run(ICamApiMacroRunContext context, out TResultStatus ret)
    {
        ret = default;
        try
        {
            if (context.CamApplication is not ICamApiApplication application)
                throw new Exception("Invalid application");

            MacroParams.Initialize(context.Params);                // see
macro-params.md
            using var applicationCom = ComWrapper.Create(application);
            // ... your work here — call the regular API (see ../api/) ...
        }
        catch (Exception e)
        {
            ret.Code = TResultStatusCode.rsError;
            ret.Description = e.Message;
        }
    }
}
```

The run context — `ICamApiMacroRunContext`

Member	Description
`CamApplication` (`IUnknown`)	The live `ICamApiApplication`. Wrap it in `ComWrapper.Create(...)` — this is your gateway to everything (project, technologist, geometry, tools, NC).
`Params` (`string`)	JSON of user-supplied per-step value overrides. Pass it to `MacroParams.Initialize` — see macro-params.md.

Generated project layout (when you edit a built macro)

When a macro is created via the builder, ENCY generates a small project:

- `Extension.cs` — the macro class (`Run` body).
- `ExtensionFactory.cs` — the factory above.
- a global-usings file — brings in `CAMAPI.Extensions`, `CAMAPI.ResultStatus`, `CAMAPI.Macros`, `CAMAPI.Application`, `CAMAPI.DotnetHelper`, etc., so the body needs no explicit `using`s.
- a `.csproj` — its `TargetFramework`, `SDKVersion` and reference assemblies are filled by the builder. Do not hand-edit the references — if they are wrong the project will not resolve CAMAPI types. (Builders set them from the extension manager; see ../api/application.md.)

If you edit `Extension.cs` by hand, the two macro-specific things to keep in mind are **`MacroParams`** (macro-params.md) and **`NotifyMacroStep`** (notify-step.md); for everything else (the commands you call) consult the regular API docs in ../api/.

Writing the macro body

Everything a macro does happens inside `ICamApiMacro.Run`. You get the live application; from there you call the regular CAM API exactly as any in-process extension would.

```
public void Run(ICamApiMacroRunContext context, out TResultStatus ret)
{
    ret = default;
    try
    {
        if (context.CamApplication is not ICamApiApplication application)
            throw new Exception("Invalid application");

        MacroParams.Initialize(context.Params);                // overridable
values — macro-params.md
        using var applicationCom = ComWrapper.Create(application);

        // --- your work: call the regular API ---
        using var projectCom      = applicationCom.GetActiveProject();
        using var technologistCom = projectCom.Technologist();
        // ...
    }
    catch (Exception e)
    {
        // Never let an exception cross the COM boundary — report via TResultStatus.
        ret.Code = TResultStatusCode.rsError;
        ret.Description = e.Message;
    }
}
```

Rules

- Wrap the application in `ComWrapper.Create(application)` and use `using` for every COM object you obtain. See ../general/com-lifetime.md.
- Never throw across the COM boundary. Catch everything and set `ret` (`TResultStatus`). See ../general/error-handling.md.
- The actual commands (create operations, edit geometry, assign tools, run NC, …) are the regular API — they are not documented here. Find them in ../api/: project.md, geometry.md, tools-machine.md, nc-simulation.md.

Worked example — create an operation

```
public void Run(ICamApiMacroRunContext context, out TResultStatus ret)
{
    ret = default;
    try
    {
        if (context.CamApplication is not ICamApiApplication application)
            throw new Exception("Invalid application");
        MacroParams.Initialize(context.Params);

        using var applicationCom  = ComWrapper.Create(application);
        using var macroManagerCom = applicationCom.MacroManager();
        using var projectCom      = applicationCom.GetActiveProject();
        using var technologistCom = projectCom.Technologist();

        macroManagerCom.NotifyMacroStep(0);                    // report progress
— notify-step.md
        using var operationCom = technologistCom.CreateOperation("HoleMachiningOp");

        // Read an overridable value (falls back to the default if the user did not
change it):
        var safeZ = MacroParams.GetFlt(0, "value", 5.0);
        // ... apply safeZ to the operation via its XML properties (see
../api/project.md) ...
    }
    catch (Exception e)
    {
        ret.Code = TResultStatusCode.rsError;
        ret.Description = e.Message;
    }
}
```

> `CreateOperation`, `Technologist`, `GetActiveProject` and the operation-property APIs are documented in ../api/project.md. This page only shows the macro scaffolding around them.

Overridable parameters — `MacroParams`

A macro can expose values that the user (or a caller of `Execute`) overrides without editing the code. The macro reads them through the static `MacroParams` helper; if no override was supplied, the recorded default is used.

In the macro body

Call `Initialize` once at the start of `Run`, then read values per step:

```
MacroParams.Initialize(context.Params);     // context.Params is the overrides JSON

double v = MacroParams.GetFlt(stepIndex, "value", 10.0);   // default 10.0 if
not overridden
string  s = MacroParams.GetStr(stepIndex, "fullName", "");
int     i = MacroParams.GetInt(stepIndex, "mode", 0);
bool    b = MacroParams.GetBol(stepIndex, "enabled", false);
```

- `stepIndex` is the 0-based index of the step within the macro (the same index used by `NotifyMacroStep` and by the macro's `commands.json`).
- The last argument is the default — returned when the user did not override that key. So a macro always runs correctly even with empty params.

The overrides format

`context.Params` is a JSON dictionary `{ "stepIndex": { "key": "value" }, ... }` — exactly what `ICamApiMacroManager.Execute(id, paramsJson)` accepts (see ../api/application.md). Empty string = no overrides (all `GetXxx` return their defaults).

```
{ "0": { "value": "7.5" }, "3": { "fullName": "Part\\Face1" } }
```

Values are always strings in the JSON; `GetInt`/`GetFlt`/`GetBol` parse them (invariant culture for floats).

Making a parameter editable in the UI

The macro's `commands.json` (written next to the built macro) drives the editing UI. Each step's entry may carry `editable_params`:

```
{
  "display_text": "Set parameter: SafeZ = {0}",
  "editable_params": [
    { "key": "value", "label": "Safe Z", "type": "flt", "default_value": "5" }
```

```
  ]
}
```

- `type` is `str` / `bol` / `int` / `flt`.
- `default_value: null` (or absent) marks the parameter required — the UI refuses to run until the user supplies a value.
- `display_text` uses `{N}` placeholders that the UI replaces with editable fields.

A host can discover these programmatically via `ICamApiMacroInfo.Step[i].Param[j]` (`Key` / `LabelText` / `ParamType` / `Required` / `DefaultValue` / `ValuesString`) — see ../api/application.md.

Reporting progress — `NotifyMacroStep`

`NotifyMacroStep` is macro-specific: a running macro tells ENCY which step it has reached so the UI can highlight the current step and honour breakpoints. It does no CAM work — it is purely a progress signal.

How to call it

Get the macro manager from the application and notify before doing each logical step's work:

```
using var applicationCom  = ComWrapper.Create(application);
using var macroManagerCom = applicationCom.MacroManager();

macroManagerCom.NotifyMacroStep(0);
// ... step 0 work ...
macroManagerCom.NotifyMacroStep(1);
// ... step 1 work ...
```

(`MacroManagerHelper.NotifyMacroStep(int)` wraps the call; the raw method is `ICamApiMacroManager.NotifyMacroStep(stepIndex, out ret)`.)

What the step index means

- `stepIndex` is the 0-based position of the step within this macro, matching the order in the macro's `commands.json` and the `stepIndex` used by `MacroParams`.
- Builder-generated macros emit a `NotifyMacroStep(n)` at the start of every action step, with `n` incrementing across all steps (not just operation creation).

Why it matters

- UI feedback: the macros window highlights the step currently executing.
- Breakpoints: the user can pause a macro on a step; the pause is evaluated when that step is notified.

If you hand-write a macro, calling `NotifyMacroStep` is optional but recommended — without it the UI cannot show progress or stop on a step. Keep the indices consistent with the values you pass to `MacroParams.GetXxx`.

Recording and saving a macro over IPC (CAMIPC)

A macro is recorded as a sequence of commands, then generated into a runnable macro. This is the full flow, top to bottom.

Objects

```
application.MacroManager → GetMacroBuilder → builder
builder.GetCommandsManager → commands
```

Lifecycle

1. Start the recording

   ```
   commands.Start()
   ```

2. Record commands — repeat for each action. A command is a bag of key/value fields; which keys to fill (and which are required) comes from the schema, not from guessing — see *Discovering a command's parameters* below.

   ```
   data := commands.CreateCommandData(commandType)   // commandType
   ∈ TMacroCommandType
   data.SetStr(key, value)                            // setter per field type
   (see below)
   data.SetInt(key, value)
   ...
   commands.RegisterCommand(data)
   ```

3. Stop the recording

   ```
   commands.Stop()
   ```

4. Set the name and language

   ```
   main := builder.CreateMainSettings()
   main.SetCaption('MyMacro')   // display name in the macro list
   main.SetId('MyMacro')        // identifier / folder name
   // optional: main.SetDescription(...) / SetOutputFolder(...)
   / SetCreateOperations(...)
   ```

```
lang := builder.CreateLanguageSettings('dotnet')   // 'dotnet' (C#) or 'spr'
```

Unknown language id fails with `rsError`. Any settings you leave unset are filled with host defaults in the next step (output folder, .NET references, and an auto-generated id when `Id` is empty).

5. Generate the macro

```
builder.CreateMacro(main, lang)
```

6. Save it

```
path := builder.Save()   // writes the macro to disk, registers it, returns the path
```

Capturing current project state

Besides recording individual commands, you can snapshot the current project state into the recording. Call these between **`Start`** and **`Stop`** — each one appends command(s) to the active recording, indistinguishable from user-recorded ones (replay applies them as-is).

Method	Effect
`commands.AddMachineState()`	capture the current machine selection
`commands.AddWorkpieceState()`	capture the current workpiece primitive
`commands.AddStrategyState()`	capture the current operations strategy
`commands.AddRecognizeFeature()`	append a blank "recognize feature by type" step; the feature type and target item are chosen later in the macro row (no pick required at record time)

Each method requires an active recording session; called before `Start` or after `Stop` it fails with `rsError` ("Macro recording is not started").

> Replaces the former single `CaptureProjectState(captureMachine, captureWorkpiece, captureStrategy)` call — the three `Add*State` methods map 1:1 to its flags, and `AddRecognizeFeature` is new.

Discovering a command's parameters

Do not hardcode keys. Ask the schema what a command expects.

```
schema := commands.GetCommandSchema()
fields := schema.GetFlatCommand(commandType)
```

`commandType` is a value of the `TMacroCommandType` enum — enumerate it to learn the available command types.

`fields` describes one command:

- `GetCount()` — number of fields.
- `GetKey(i)` — the field key (the string you pass to a setter).
- `GetFieldType(i)` — value type, which dictates the setter to use:
  - `0 = Str` → `SetStr`
  - `1 = Bool` → `SetBool`
  - `2 = Int` → `SetInt`
  - `3 = Float` → `SetFloat`
- `GetRequired(i)` — `true` ⇒ the field must be set, otherwise the macro is malformed.

Rules when filling:

- Required fields: always set them. Omitting one produces a macro that may fail on replay.
- Optional fields: set them only when you have a value. An empty value is not recorded and will not overwrite the target's own default.

Variant (model-item) commands

Some commands have variant-specific keys selected by a discriminator (e.g. a model item whose keys differ per class). For those, `GetFlatCommand` returns the common keys including the discriminator key; read its value, then request the variant keys:

```
variantFields := schema.GetClassItemCommand(commandType, discriminator)
```

Guard cases

Action	Condition	Result
`Start`	a recording is already active	error — "Macro recording is already started"
`Stop`	not currently recording	no-op (not an error)

Action	Condition	Result
`RegisterCommand`	called before `Start` or after `Stop`	command is silently dropped (no error)
`CreateMacro`	recording not stopped yet	error — "Stop the recording before building the macro."
`CreateMacro`	nothing was recorded	error — "No recorded macro to build…"
`CreateMacro`	called again without a new `Start`	error — "Macro record wasn't started yet"
`Save`	no preceding `CreateMacro`	error — "Macro code file is not created. Call CreateMacro first."
`AddMachineState` / `AddWorkpieceState` / `AddStrategyState` / `AddRecognizeFeature`	called before `Start` or after `Stop`	error — "Macro recording is not started"

Instead of tracking the session yourself, ask: `commands.GetIsStarted()` is `true` between `Start` and `Stop`. Worth checking when the user may also be recording through the UI.

Managing and running macros

`application.MacroManager` (`ICamIpcMacroManager`) is the entry point for everything that is not recording:

Method	Description
`GetCount()`	Number of registered macros
`GetMacro(index)`	Macro by index → `ICamIpcMacroInfo`
`GetMacroById(id)`	Macro by id
`CreateMacroInstance()`	New empty `ICamIpcMacroInfo` to fill and register
`AddMacro(macro)`	Register a macro
`RemoveMacro(id, deleteSources)`	Unregister; `deleteSources` also deletes its files
`GetMacroBuilder()`	The builder used for recording (above)

Method	Description
`Execute(id, params)`	Run a macro — see below
`NotifyMacroStep(stepIndex)`	Report replay progress for the given step
`OpenInEditor(macro)`	Open the macro source in the editor for its language
`ExportMacro(id, targetPath)`	Export to a single self-contained `.dmcr` archive
`ImportMacro(sourcePath)`	Import a `.dmcr` archive and register it → `ICamIpcMacroInfo`

`Execute`'s `params` is a JSON dictionary of user-overridden values, keyed by step index:

```
{"0": {"feed": "250"}, "3": {"depth": "12.5"}}
```

Pass an empty string to run with the recorded defaults.

> `ImportMacro` overwrites a macro with the same id, including its previous folder — check `GetMacroById` first if that matters.

Inspecting a macro's steps

`ICamIpcMacroInfo` exposes the recorded steps, which is what a playback UI needs to show editable parameters before running:

- `ICamIpcMacroInfo`: `GetStepCount()`, `GetStep(index)`, plus read/write `Id`, `Caption`, `Description`, `MacroPath`, `ExecutablePath`, `LanguageId`, `ExecuteExtensionId`.
- `ICamIpcMacroStepInfo` (read-only): `GetDisplayText()`, `GetParamCount()`, `GetParam(index)`, `GetGroupCaption()`.
- `ICamIpcMacroStepParam` (read-only): `GetKey()`, `GetLabelText()`, `GetParamType()`, `GetRequired()`, `GetDefaultValue()`, `GetValuesString()`.

`GetParamType()` returns the integer ordinal of `TMacroCommandParamType` — `0 = Str`, `1 = Bool`, `2 = Int`, `3 = Float` — the same numbering as `GetFieldType` above. `GetValuesString()` is a semicolon-separated list of allowed values, empty for free-form. `GetRequired()` means the step was recorded without a default, so a playback UI must refuse to run until the user supplies one.

The keys you collect this way are exactly the keys of the `Execute` params JSON.

> The full CAMAPI-side model — including the composite/pill parameter types and the presentation flags — is documented in [../api/application.md](). The IPC mirror exposes the read-only subset

> listed above.

Language settings and .NET

`ICamIpcMacroBuilderLanguageSettings` folds the .NET-specific properties (`TargetFramework`, `SDKVersion`, `References`) directly into the one interface — the server narrows the wrapped object to the concrete language interface. For SPR settings the .NET getters return empty strings and the setters do nothing, so there is no separate interface to request. `References` marshals by value as a comma-separated list.

ENCY CAM IPC — Standalone Application Development Overview

What is CAMIPC?

CAMIPC is the inter-process API for standalone .NET/C++ applications that communicate with a running ENCY instance via IPC (sockets + JSON).

Quick Start

1. Clone ENCY connection → ipc/connection.md
2. Get ICamIpcApplication → access project, technologist, geometry
3. All calls use TExecuteContext for error checking

How CAMAPI vs CAMIPC differ

Aspect	CAMAPI (plugin)	CAMIPC (standalone)
Process	Same process as ENCY	Separate process
Loading	ENCY loads your DLL	You load CAMIPC.Helper.Cam.dll
Entry	ExtensionFactory.Create()	CreateHelper() → GetRunningCamAppList()
Threading	COM MTA (ENCY's thread)	Your thread, use TExecuteContext
Interface prefix	ICamApi*	ICamIpc*

Documentation Map

Topic	File
Connecting to ENCY	ipc/connection.md
Project, Technologist, Operations	ipc/project.md
Geometry	ipc/geometry.md
CAD modelling	ipc/cad.md
Feature finder	ipc/feature-finder.md
Tools & Machine	ipc/tools-machine.md

Topic	File
NC Generation & Simulation	ipc/nc-simulation.md
Application, Logger, XmlProp	ipc/application.md
UI (viewport, Prime)	ipc/ui.md
Clouds, Tests, PLM	ipc/extra.md
Macros — record & save over IPC	ipc/macros.md
COM lifetime (.NET)	general/com-lifetime.md
Error handling	general/error-handling.md

Connecting a .NET Application to a Running ENCY Instance

This guide covers connecting a standalone .NET process (console app, WPF app, etc.) to a live ENCY instance via IPC. For in-process plugins loaded by ENCY itself, see the extension entry points guide; the IPC layer is only needed for out-of-process automation.

Reference example: `ApplicationNet/CreateOperations/project/main/CamHelper.cs`

Overview

The IPC bridge is provided by `CAMIPC.Helper.Cam.dll`, which ships in the ENCY installation folder. The DLL exposes a single C export, `CreateHelper`, that returns an `IIpcHelper` COM interface. From there you enumerate running ENCY instances, pick one, and receive an `ICamIpcApplication` that gives access to the full CAM API over a socket-based IPC channel.

Every call across the IPC channel takes a `ref TExecuteContext` argument. You must check `executeContext.ResultStatus.Code` after each call.

Step 1: Load the DLL and Create IIpcHelper

```
using System.IO;
using System.Runtime.InteropServices;
using CAMAPI.DotnetHelper;
using CAMIPC.Helper;
using CAMHelper.NativeLibUtils;

// Adjust path to match the installed ENCY version
const string camFolder = @"C:\Program Files\ENCY Software\ENCY 2\Bin64";
var helperPath = Path.Combine(camFolder, "CAMIPC.Helper.Cam.dll");

if (!File.Exists(helperPath))
    throw new FileNotFoundException($"CAMIPC.Helper.Cam.dll not found
at: {helperPath}");

// Load the native DLL
var helperDllPtr = NativeLibLoader.LoadDll(helperPath, out var loadResult);
if (helperDllPtr == IntPtr.Zero || loadResult != 0)
    throw new Exception($"Failed to load CAMIPC.Helper.Cam.dll: error
code {loadResult}");

// Bind the CreateHelper export
```

```
delegate IntPtr CreateHelperDelegate();
var createHelper = NativeLibLoader.GetProc<CreateHelperDelegate>
(helperDllPtr, "CreateHelper");

// Obtain IIpcHelper — wrap immediately for deterministic release
var helperCom = new ComWrapper<IIpcHelper>(createHelper());
```

`NativeLibLoader` is a thin P/Invoke wrapper provided in the ENCY helper NuGet package. `FreeDll` must be called on the pointer when the application exits (see the `CamHelper` dispose chain below).

Step 2: Enumerate Running Instances

`IIpcHelper.GetRunningCamAppList` returns an `ICamIpcListApplication`. Each element is an `ICamIpcApplication` representing one running ENCY process.

```
using CAMIPC.ExecuteContext;
using CAMAPI.ResultStatus;

var executeContext = new TExecuteContext();

using var instancesCom = helperCom.InvokeAndWrap(helper =>
    helper.GetRunningCamAppList(ref executeContext));

if (executeContext.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception(executeContext.ResultStatus.Description);

// Get the first available instance (index 0)
var applicationCom = instancesCom.InvokeAndWrap(instances =>
{
    if (instances.Count == 0)
        throw new Exception("No running ENCY instance found. Start ENCY first.");
    return instances.Get(0, executeContext);
});

if (executeContext.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception(executeContext.ResultStatus.Description);
```

If you need a specific instance (e.g., by process ID), use `IIpcHelper.GetRunningCamApp(processId, ref executeContext)` instead.

Step 3: The CamHelper Wrapper Class

In practice, wrap the entire connection lifecycle in a disposable class. The dispose order is critical: the application COM object must be released before the helper, and the helper before the DLL is unloaded.

```
using System;
using System.IO;
using CAMAPI.DotnetHelper;
using CAMAPI.ResultStatus;
using CAMHelper.NativeLibUtils;
using CAMIPC.Application;
using CAMIPC.ExecuteContext;
using CAMIPC.Helper;

public class CamHelper : IDisposable
{
    private readonly ComWrapper<IIpcHelper> _helper;
    private readonly ComWrapper<ICamIpcApplication>? _application;
    private readonly IntPtr _helperDllPtr;

    private delegate IntPtr CreateHelperDelegate();

    public ComWrapper<ICamIpcApplication> GetApplication() =>
        _application ?? throw new Exception("ENCY application not connected");

    public CamHelper()
    {
        const string camFolder = @"C:\Program Files\ENCY Software\ENCY 2\Bin64";
        var helperPath = Path.Combine(camFolder, "CAMIPC.Helper.Cam.dll");
        if (!File.Exists(helperPath))
            throw new FileNotFoundException($"CAMIPC.Helper.Cam.dll not
found: {helperPath}");

        _helperDllPtr = NativeLibLoader.LoadDll(helperPath, out var loadResult);
        if (_helperDllPtr == IntPtr.Zero || loadResult != 0)
            throw new Exception($"Failed to load helper DLL: error
code {loadResult}");

        var proc = NativeLibLoader.GetProc<CreateHelperDelegate>
(_helperDllPtr, "CreateHelper");
        _helper = new ComWrapper<IIpcHelper>(proc());

        var executeContext = new TExecuteContext();

        using var instancesCom = _helper.InvokeAndWrap(h =>
            h.GetRunningCamAppList(ref executeContext));
        if (executeContext.ResultStatus.Code == TResultStatusCode.rsError)
```

```
            throw new Exception(executeContext.ResultStatus.Description);

        _application = instancesCom.InvokeAndWrap(instances =>
        {
            if (instances.Count == 0)
                throw new Exception("No running ENCY instance found.");
            return instances.Get(0, executeContext);
        });
        if (executeContext.ResultStatus.Code == TResultStatusCode.rsError)
            throw new Exception(executeContext.ResultStatus.Description);
    }

    public void Dispose()
    {
        // Dispose order matters: application before helper, helper before DLL
        _application?.Dispose();
        _helper.Dispose();
        NativeLibLoader.FreeDll(_helperDllPtr);
    }
}
```

Usage:

```
// In a WPF application (STA), set this before constructing CamHelper
ComWrapperSettings.ApplicationApartmentState = ApartmentState.STA;

using var cam = new CamHelper();
var executeContext = new TExecuteContext();

using var projectCom = cam.GetApplication().InvokeAndWrap(app =>
    app.GetActiveProject(ref executeContext));
if (executeContext.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception(executeContext.ResultStatus.Description);
```

TExecuteContext

`TExecuteContext` is a struct that accompanies every IPC call. It carries two fields:

```
TExecuteContext
├── ExecuteSettings: TExecuteSettings
│   ├── ThreadHandle: integer   // 0 = main thread (default)
│   └── Timeout: integer        // 0 = use ENCY default timeout
└── ResultStatus: TResultStatus
```

```
    ├── Code: TResultStatusCode // rsOk | rsError
    └── Description: string     // error message when Code == rsError
```

Always create a fresh **`TExecuteContext`** (or reset it) before a group of related calls, and check **`ResultStatus.Code`** after each call that can fail:

```
var executeContext = new TExecuteContext();
// Optionally configure: executeContext.ExecuteSettings.Timeout = 30000; // 30 s

using var projectCom = applicationCom.InvokeAndWrap(app =>
    app.GetActiveProject(ref executeContext));

if (executeContext.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception("GetActiveProject failed: " +
executeContext.ResultStatus.Description);
```

`ThreadHandle` selects which thread inside ENCY processes the call. The default (0) is the main thread. Most API calls must run on the main thread.

IAsyncResult

Some long-running IPC operations return `IAsyncResult` instead of completing synchronously. This allows you to start the operation and poll or wait for completion.

```
// IAsyncResult interface:
//   WaitFor(timeout: int) -> bool        — waits up to `timeout` ms; returns true
if done
//   GetResult(timeout: int) -> IUnknown* — waits and returns the COM result object
//   GetResultStatus(timeout: int) -> TResultStatus — waits and returns the status
```

Typical pattern:

```
using var asyncResultCom = someOperationCom.InvokeAndWrap(op =>
    op.StartLongRunningOperation(ref executeContext));

// Wait up to 60 seconds
bool completed = asyncResultCom.Invoke(ar => ar.WaitFor(60_000));
if (!completed)
    throw new TimeoutException("Operation did not complete within 60 s");

// Retrieve the status
var status = asyncResultCom.Invoke(ar => ar.GetResultStatus(0));
```

```
if (status.Code == TResultStatusCode.rsError)
    throw new Exception("Async operation failed: " + status.Description);
```

Use async calls for operations that block the ENCY main thread for extended periods (toolpath calculation, simulation, G-code generation). For quick property reads and object traversal, synchronous calls are preferred.

ICamIpcEventListener — Subscribing to ENCY Events

To receive events from a running ENCY instance, register an event listener through `IIpcEventListenerManager`. ENCY sends events over a socket to the port your listener opens.

The general pattern:

1. Obtain `IIpcEventListenerManager` from the interaction layer.
2. Call `RegisterListener(listenerIdent)` — this opens a local socket and returns an `ICamIpcEventListener` whose `Port` property holds the listening port number.
3. Call `RegisterHandler(senderInstanceIdent, eventHeader, handlerIdent, handler, listener)` — `handler` is your implementation of `ICamIpcEventHandler`.
4. Implement `ICamIpcEventHandler.GetTimeOut(interfaceUid)` — return `-1` for async (fire-and-forget) or `0` for infinite wait. Positive values specify a millisecond timeout.

```
// ICamIpcEventHandler implementation
public class MyEventHandler : ICamIpcEventHandler
{
    public int GetTimeOut(string interfaceUid) => -1; // async, ENCY doesn't wait
for us

    public void OnEvent(string eventIdent, string paramsJson)
    {
        Console.WriteLine($"Event received: {eventIdent}, params: {paramsJson}");
    }
}
```

Listener registration and unregistration must be paired. Always call `UnregisterHandler` and `UnregisterListener` when shutting down, otherwise the ENCY process may hold a connection open.

Error Handling in IPC

IPC errors surface through `TExecuteContext.ResultStatus` — the same `TResultStatus` used everywhere in the API. Check it after every call:

```
private static void CheckStatus(TExecuteContext ctx, string callName)
{
    if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
        throw new Exception($"{callName} failed: {ctx.ResultStatus.Description}");
}

// Usage:
var ctx = new TExecuteContext();
using var projCom = applicationCom.InvokeAndWrap(app => app.GetActiveProject(ref
ctx));
CheckStatus(ctx, nameof(ICamIpcApplication.GetActiveProject));
```

Common failure causes over IPC:

- ENCY process exited between `GetRunningCamAppList` and the actual call — the call returns `rsError` with a connection error description.
- Timeout exceeded — increase `ExecuteContext.ExecuteSettings.Timeout`.
- Calling on the wrong thread handle — use the default (0) unless you have a specific reason to target another thread.

Launching a New ENCY Instance

Instead of connecting to an existing instance, you can launch a new one:

```
// Console mode (headless, no window)
using var appCom = helperCom.InvokeAndWrap(h =>
    h.RunNewConsoleApp("", ref executeContext));

// Window mode (with UI, hosted in a Prime window)
using var appCom = helperCom.InvokeAndWrap(h =>
    h.RunNewWinApp("", ref executeContext));
```

Both methods block until the ENCY process has started and is ready to accept IPC calls. The returned `ICamIpcApplication` is ready for use immediately.

To shut down a process you started:

```
int processId = appCom.Invoke(app => app.ProcessId);
helperCom.Invoke(h => h.KillCamApp(processId, ref executeContext));
```

ENCY CAM IPC API — Application Domain

This document covers the out-of-process (IPC) Application interfaces used when controlling a ENCY CAM instance from an external process (e.g., a standalone executable, a Prime shell, or another tool).

IPC interfaces mirror the in-process `CAMAPI` interfaces but add `TExecuteContext` parameters to every call that crosses the process boundary.

Table of Contents

1. ICamIpcApplication — remote application instance
2. ICamIpcListApplication — managing multiple instances
3. IPC application event handlers
4. ICamIpcPaths — system path properties
5. IIpcExtensionManager — managing extensions over IPC
6. ICamIpcUtilityManager / ICamIpcUtilityInfo
7. IIpcLogger — logging from an IPC context
8. ICamIpcXmlPropPointer / ICamIpcXmlPropArray — XML property tree

---

ICamIpcApplication

The remote proxy for a single running ENCY CAM process. All methods that read from or write to the remote application take a `TExecuteContext` so the IPC layer can route the call and track the result.

Connection management properties

Property	Type	Description
`ProcessId`	`integer`	OS process ID of the ENCY instance
`IsAlive`	`boolean`	Whether the instance is running and responding
`IsBusy`	`boolean`	Whether the instance is currently processing an IPC request
`IsWindowed`	`boolean`	Whether the instance has a visible window
`ExecutablePath`	`string`	Path to the ENCY executable

Property	Type	Description
`MainForm`	`ICamIpcApplicationMainForm*`	Proxy for the main window (available immediately without context)
`MachinesLibrary`	`ICamIpcMachinesLibrary*`	Machine library manager
`MacroManager`	`ICamIpcMacroManager*`	Macro system manager
`Paths`	`ICamIpcPaths*`	System path information
`UserTechOperationList`	`ICamIpcUserTechOperationList*`	User tech operation list

Connection lifecycle methods

Method	Description
`WaitForStarted(timeout, out ResultStatus)`	Blocks until the ENCY instance is ready or the timeout (ms) elapses
`Close(asyncMode, out ResultStatus) → IListString*`	Sends a close request; returns any error strings
`StartAsyncWork(out ResultStatus)`	Starts the local IPC server to receive async callbacks from ENCY
`StopAsyncWork(out ResultStatus)`	Stops the local IPC server
`GetPort(out ResultStatus) → integer`	Returns the TCP/named-pipe port the IPC server listens on
`RegisterWorkingThread(threadHandle, out ResultStatus)`	Registers a thread to process async IPC commands
`UnregisterWorkingThread(threadHandle, out ResultStatus)`	Unregisters a working thread

Remote application methods (require TExecuteContext)

Method	Description
`GetLogFilePath(ctx) → string`	Path to the ENCY log file
`GetActiveProject(ctx) → ICamIpcProject*`	Returns the active project proxy

Method	Description
`GetExtensionManager(ctx) → IIpcExtensionManager*`	Returns the extension manager proxy
`GetUtilityManager(ctx) → ICamIpcUtilityManager*`	Returns the utility manager proxy
`GetMachiningToolsManager(ctx) → ICamIpcMachiningToolsManager*`	Returns the tools manager proxy
`GetPLMManager(ctx) → IIpcPLMManager*`	Returns the PLM manager proxy
`GetMainFormTimeOut(timeout) → ICamIpcApplicationMainForm*`	Returns the main form proxy, waiting up to `timeout` ms
`OpenProject(fileName, addToReOpen, ctx)`	Opens a project file
`SaveCurrentProject(fileName, ctx)`	Saves the current project
`ExportCurrentProject(targetFileName, overwrite, ctx)`	Exports the project with all snapshots
`GetMainWorkMode(ctx) → TMainWorkMode`	Returns the current work mode
`SetMainWorkMode(workMode, ctx)`	Sets the work mode
`SetActiveProjectMachine(machineInfo, ctx)`	Assigns a machine to the current project
`OpenProjectFromPLM(plmItemId, connectionId, ctx)`	Opens a PLM project

Event handling over IPC

IPC events require a `listener` object in addition to the handler:

```
CreateListener(ctx) → ICamIpcEventListener*
GetDefaultListener() → ICamIpcEventListener*
RegisterHandler(handlerIdent, handler, listener, ctx)
UnregisterHandler(handlerIdent, ctx)
```

A listener encapsulates the transport channel that delivers events from ENCY back to the external process. Typically you use `GetDefaultListener()` for the default channel and `CreateListener(ctx)` when you need an isolated channel per module.

Usage pattern

```
// Obtain an ICamIpcApplication proxy (from ICamIpcSingletons or Prime):
var ctx = new TExecuteContext();
app.WaitForStarted(10_000, out var status);
if (status.Code == TResultStatusCode.rsError)
    throw new Exception(status.Description);

// Open a project:
app.OpenProject(@"C:\projects\part.encam", addToReOpen: false, ref ctx);

// Read the work mode:
var mode = app.GetMainWorkMode(ref ctx);

// Get active project:
var project = app.GetActiveProject(ref ctx);
```

ICamIpcListApplication

A collection of `ICamIpcApplication` instances. Returned by the singletons or Prime to enumerate all running ENCY processes.

```
Count: integer
Get(index, ctx) → ICamIpcApplication*
GetByProcessId(processId, ctx) → ICamIpcApplication*
Add(item)
RemoveByProcessId(processId)
RemoveAt(index)
Clear()
```

IPC Application Event Handlers

These are the IPC equivalents of the in-process application event handlers. They are registered via `ICamIpcApplication.RegisterHandler`.

Interface	Method
`ICamIpcHandlerApplicationAfterLoad`	`ApplicationAfterLoad(handlerIdent, application)`

Interface	Method
ICamIpcHandlerApplicationBeforeClose	ApplicationBeforeClose(handlerIdent, application)
ICamIpcHandlerApplicationAfterClose	ApplicationAfterClose(handlerIdent)
ICamIpcHandlerApplicationNewProject	ApplicationNewProject(handlerIdent)
ICamIpcHandlerApplicationActiveProjectChanged	ApplicationActiveProjectChanged(handlerI newProject)
ICamIpcHandlerApplicationBeforeLoadProject	ApplicationBeforeLoadProject(handlerIden project)
ICamIpcHandlerApplicationAfterLoadProject	ApplicationAfterLoadProject(handlerIdent project)
ICamIpcHandlerApplicationBeforeSaveProject	ApplicationBeforeSaveProject(handlerIden project)
ICamIpcHandlerApplicationAfterSaveProject	ApplicationAfterSaveProject(handlerIdent project)

Interface	Method
`ICamIpcHandlerApplicationUpdateStartProgress`	`ApplicationUpdateStartProgress(handlerId caption, percent)`
`ICamIpcHandlerApplicationUpdateProcessState`	`ApplicationUpdateProcessState(handlerIde caption, percent)`

ICamIpcPaths

Exposed directly on `ICamIpcApplication.Paths`. Contains the same path properties as the in-process `ICamApiPaths`, plus a unique instance identifier.

Methods and properties

Member	Description
`GetInstanceId() → string`	Unique identifier of this ENCY instance in IPC messages
`MainProgramFolder`	Root installation folder
`InterpretersFolder`	NC interpreter files
`ModelsFolder`	Default models folder
`NCProgramsFolder`	Default NC programs folder
`ExamplesFolder`	Example projects
`PostprocessorsFolder`	Postprocessor files
`OperationsFolder`	XML files with additional operation info
`MachinesFolder`	Machine definition files
`LibrariesFolder`	Help library files

Member	Description
`ConfigFolder`	Configuration files
`UserExtensionsFolder`	User-installed extensions
`TryReducePath(fullPath) → string`	Converts an absolute path to a portable CAM-variable form
`TryUnfoldPath(reducedPath) → string`	Expands a portable CAM-variable path to an absolute path

---

IIpcExtensionManager

The IPC proxy for the ENCY extension manager. All methods take a `TExecuteContext`.

Library and storage management

Method	Description
`RegisterLibrary(storageType, descFilePath, ctx) → IIpcExtensionLibraryInfo*`	Registers an extension library from a JSON description file
`UnRegisterLibrary(libraryPath, ctx)`	Removes a library registration
`RegisterLibraryFromFolder(storageType, folderPath, out needReload, ctx) → IIpcExtensionLibraryInfo*`	Finds the `settings.json` in `folderPath`, copies the library into ENCY's own folder, then registers it. `needReload` reports that a restart is required to pick it up
`GetPendingDeletionLibraries() → IIpcListExtensionLibraryInfo*`	Libraries whose unregistration could not complete because the DLL was locked (a `_del_` cleanup artifact is still on disk). Stale entries are dropped automatically, so a non-empty result means a real leftover. No **`ctx`** parameter
`ReloadStorage(storageType, ctx)`	Re-reads a storage layer
`GetLibrariesInfo() → IIpcListExtensionLibraryInfo*`	Lists all registered libraries
`GetLibraryInfo(extensionTypeId, ctx) → IIpcExtensionLibraryInfo*`	Gets library info by extension type ID

Method	Description
`FreeLibrary(libraryPath, ctx)`	Unloads all extensions from a library

Extension enable/disable

Method	Description
`GetLibraryDisabled(storageType, libraryPath, ctx) → boolean`	Whether the library is disabled for a storage layer
`GetExtensionDisabled(storageType, extensionIdent, ctx) → boolean`	Whether a specific extension is disabled
`SetLibraryDisabled(storageType, libraryPath, disabled, ctx)`	Enables or disables a library for a storage layer
`SetExtensionDisabled(storageType, extensionIdent, disabled, ctx)`	Enables or disables an extension
`GetFieldInherited(storageType, entityIdent, field, ctx) → boolean`	Checks if a field value is inherited
`SetFieldInherited(storageType, entityIdent, field, ctx)`	Resets a field to use the inherited value
`GetLibraryLoaded(libraryPath, ctx) → boolean`	Whether a library is currently loaded

Extension discovery and instantiation

Method	Description
`GetExtensionTypeGroups(ctx) → IListString*`	Lists all available extension type group names
`GetExtensionTypeInfo(extensionTypeId, ctx) → IIpcExtensionTypeInfo*`	Gets type metadata for one extension
`GetExtensionTypeInfos(extensionTypeGroup, ctx) → IIpcListExtensionTypeInfo*`	Lists all extension types in a group
`GetExtensionTypeInfosFromLibrary(libraryPath, ctx) → IIpcListExtensionTypeInfo*`	Lists all extensions declared in a library DLL

Method	Description
`CreateExtension(extensionTypeId, ctx) → IIpcExtension*`	Creates (loads) a new extension instance
`CreateExtensions(extensionTypeIds, ctx) → IIpcListExtension*`	Creates multiple extensions (semicolon-separated IDs)
`GetExtension(extensionInstanceId, ctx) → IIpcExtension*`	Returns an existing extension instance by ID
`GetSingletonExtension(extensionTypeId, ctx) → IIpcExtension*`	Returns the single shared instance of a singleton extension by its type id; narrow to a concrete interface with `AsInstanceOf` on the client
`FreeExtension(extensionInstanceId, ctx)`	Unloads an extension instance

Extension settings

Named key/value settings owned by an extension type and stored by the extension manager, for deployment-provided configuration (server URL, license key, shared folder). Resolution is a cascade, independent of the disabled/inherited machinery above:

```
stCurrentUser → stAllUsers → stDealer → stSystem → manifest default
```

Method	Description
`GetExtensionSetting(extensionTypeId, key, out found, ctx) → string`	Effective value after the whole cascade; `found` is `false` when nothing defines the key
`GetExtensionSettingSourceLevel(extensionTypeId, key, out found, out fromDefault, ctx) → TStorageType`	Where the effective value came from. When `fromDefault` is `true` the value is the manifest default and the returned tier is meaningless
`GetExtensionSettingKeys(extensionTypeId, ctx) → IListString*`	Union of all keys defined across every tier and the defaults

Method	Description
`SetExtensionSetting(storageType, extensionTypeId, key, value, ctx)`	Write a value into one tier and persist it
`RemoveExtensionSetting(storageType, extensionTypeId, key, ctx)`	Drop the key from one tier so the next tier down (or the default) becomes effective again

`ExtensionManagerHelper` wraps all five and throws on `rsError`, so no `TExecuteContext` is passed by the caller. `GetExtensionSetting` folds the `found` flag into the return value — it returns **`null`** when nothing defines the key:

```
using var emCom = appCom.ExtensionManager();
const string extensionTypeId = "My.Company.MyExtension";

var url = emCom.GetExtensionSetting(extensionTypeId, "serverUrl");
if (url == null)
    throw new Exception("The 'serverUrl' setting is not configured for
this installation");

var level = emCom.GetExtensionSettingSourceLevel(
    extensionTypeId, "serverUrl", out var found, out var fromDefault);

if (found && !fromDefault && level == TStorageType.stAllUsers)
{
    // machine-wide value — override it for this user only
    emCom.SetExtensionSetting(TStorageType.stCurrentUser, extensionTypeId,
"serverUrl", url);
}

// back to the inherited value
emCom.RemoveExtensionSetting(TStorageType.stCurrentUser, extensionTypeId,
"serverUrl");

// enumerate everything this extension is configured with
using var keysCom = emCom.GetExtensionSettingKeys(extensionTypeId);
```

Writing to a tier that has no storage in this installation (`stDebugMode` outside a debug session) fails instead of silently doing nothing. A key marked secret is kept in memory for the session only and never written to disk — the calls look identical, but the value is gone after a restart.

Settings are keyed by extension type id alone, so they can be written and read for an id that is not registered — handy for provisioning an extension before it is installed. `TStorageType` comes from the `CAMAPI.Extensions` namespace even in IPC clients.

The in-process equivalents live on `IExtensionManager` and have no helper wrappers — see ../api/entry-points.md.

Storage type enum — TStorageType

The configuration tier a library, flag, or setting belongs to. Tiers are merged in priority order — the highest-priority tier carrying a value wins.

Value	Priority	Scope	Writable
`stSystem`	Lowest	Machine-wide, shipped with the installation	No
`stDealer`	2	Machine-wide, provisioned by the distributor	No
`stAllUsers`	3	Machine-wide	Yes
`stCurrentUser`	4	Current user	Yes
`stDebugMode`	5	Current user, only while a debug session provides the folder	Yes
`stTestMode`	Highest	Automated test runs	Yes

Macro build support

Method	Description
`GetApiVersion(ctx) → string`	Current CAM Open API version — use as the macro .NET SDK version
`GetApiDependencies(ctx) → IListString*`	Reference assemblies a built macro must compile against — use as the macro .NET References

Helper (`ExtensionManagerHelper`): `emCom.GetApiVersion()`, `emCom.GetApiDependencies()`. These feed the macro builder's `.NET` language settings (see `ICamIpcMacroManager` below) — without them the generated macro project cannot resolve CAMAPI types.

ICamIpcMacroManager

IPC mirror of `ICamApiMacroManager` — list and run existing macros, and create new ones. See docs/api/application.md for the full conceptual model; this section gives the IPC helper surface.

(Writing the body of a macro itself is covered in docs/macros.)

```
using var macroMgrCom = appCom.MacroManager();   // ComWrapper<ICamIpcMacroManager>
```

Listing and running

```
int count = macroMgrCom.Count();
using var info = macroMgrCom.GetMacroById("MyMacro");   //
or macroMgrCom.GetMacro(index)
macroMgrCom.Execute("MyMacro", paramsJson);             // paramsJson "" =
recorded defaults
```

`paramsJson` is a per-step override dictionary: `{"stepIndex": {"key": "value"}}`.

Helpers (**`MacroManagerHelper`**):

Helper	Description
`Count()`	Number of macros
`GetMacro(i)` / `GetMacroById(id)` → `ComWrapper<ICamIpcMacroInfo>`	Macro metadata
`Execute(id, paramsJson = "")`	Run a macro
`CreateMacroInstance()` / `AddMacro(info)` / `RemoveMacro(id, deleteSources)`	Manage macros
`GetMacroBuilder()` → `ComWrapper<ICamIpcMacroBuilder>`	Build macros (below)
`NotifyMacroStep(i)` / `OpenInEditor(info)`	Step notify / open source

ICamIpcMacroInfo

`MacroInfoHelper`: getters `Id()`, `Caption()`, `Description()`, `MacroPath()`, `ExecutablePath()`, `LanguageId()`, `ExecuteExtensionId()`, `StepCount()`, `GetStep(i)`; setters `SetId(v)` … `SetExecuteExtensionId(v)` (for the `CreateMacroInstance` → fill → `AddMacro` flow).

Discovering overridable step parameters

`MacroStepInfoHelper`: `DisplayText()`, `ParamCount()`, `GetParam(i)`, `GroupCaption()`.
`MacroStepParamHelper`: `Key()`, `LabelText()`, `ParamType()` (`TMacroCommandParamType`), `Required()`, `DefaultValue()`, `ValuesString()`.

Creating a macro programmatically

```
using var builderCom = macroMgrCom.GetMacroBuilder();
using var cmdsCom    = builderCom.GetCommandsManager();

cmdsCom.Start();
using (var cd = cmdsCom.CreateCommandData(TMacroCommandType.mctCreateOperation))
{
    cd.SetStr("OperationType", "HoleMachiningOp");
    cd.SetStr("TypeCaption",   "Hole machining");
    cd.SetStr("Name",          "Op1");
    cmdsCom.RegisterCommand(cd);
}
cmdsCom.Stop();

using var main = builderCom.CreateMainSettings();
main.SetId("MyMacro");
main.SetCaption("My macro");
main.SetCreateOperations(true);
main.SetOutputFolder(outputFolder);
main.SetExecuteExtensionId("Extension.MacroRunner.Dotnet");

using var lang = builderCom.CreateLanguageSettings("dotnet");
lang.SetTargetFramework("net8.0-windows");
// CRITICAL: SDKVersion + References — without them the generated .csproj
cannot resolve
// CAMAPI types (CS0246). Source both from the extension manager.
using var emCom = appCom.ExtensionManager();
lang.SetSDKVersion(emCom.GetApiVersion());
lang.SetReferences(emCom.GetApiDependencies());

string sourcePath = builderCom.CreateMacro(main, lang);
string builtPath  = builderCom.Save();
```

Alternative — capture the current project instead of authoring commands:

```
cmdsCom.AddStrategyState();   // or AddMachineState() / AddWorkpieceState()
/ AddRecognizeFeature()
```

Command field-key schema (authoritative, at runtime)

Get the schema provider (`ICamIpcMacroCommandSchema`) from the commands manager; `GetFlatCommand` returns an `ICamIpcMacroCommandFieldList` (indexed list of field descriptors). The `Fields()` helper reads it in one pass:

```
using var schemaCom = cmdsCom.GetCommandSchema();
using var fieldsCom =
schemaCom.GetFlatCommand(TMacroCommandType.mctCreateOperation);
foreach (var f in fieldsCom.Fields())
{
    // f.Key (string), f.FieldType (TMacroCommandParamType), f.Required (bool)
}
```

Variant (model-item) commands — some commands accept different key sets depending on a *discriminator* field (e.g. `mctSetWorkpiecePrimitive`, whose keys depend on the item class). `GetFlatCommand` returns the common keys including the discriminator; pass a discriminator value to `GetClassItemCommand` for the variant keys:

```
using var boxCom = schemaCom.GetClassItemCommand(
    TMacroCommandType.mctSetWorkpiecePrimitive, "TBoxLinkItem");
// discriminators: "TBoxLinkItem", "TCylLinkItem", "TStockLinkItem", "TAutoLinkItem"
```

An empty list (`GetCount() == 0`) means the command type / discriminator has no documented schema yet.

Command bag and helpers

- `MacroCommandSchemaHelper`: `GetFlatCommand(type)`, `GetClassItemCommand(type, discriminator)` → `ComWrapper<ICamIpcMacroCommandFieldList>` (on the provider from `GetCommandSchema`).
- `MacroCommandFieldListHelper`: `GetCount()`, `GetKey(i)`, `GetFieldType(i)`, `GetRequired(i)`, and `Fields()` to read the whole list at once.
- `MacroCommandDataHelper`: `CommandType()`, `GetStr/GetInt/GetFloat/GetBool(key)`, `SetStr/SetInt/SetFloat/SetBool(key, value)` (setters work only on a bag from `CreateCommandData`).
- Language `"dotnet"` → .NET macro (`TargetFramework`/`SDKVersion`/`References` via `MacroBuilderLanguageSettingsHelper`); `"script"` → script macro.
- List-bearing commands (which the scalar bag cannot hold) have dedicated typed methods on the commands manager: `RegisterSetDriveFaceItemProperties`, `RegisterAddFixture`, `RegisterSetFixtureItemStock`/`Color`/`Caption`.

ICamIpcUtilityManager / ICamIpcUtilityInfo

ICamIpcUtilityManager

```
GetListInfo(ctx) → ICamIpcListUtilityInfo*
Execute(uid, ctx)
Reload()
GetUtilsUiInfoJson(ctx) → string
```

ICamIpcUtilityInfo — read-only utility descriptor

```
GetUid(ctx) → string
GetUiVisible(ctx) → boolean
GetName(ctx) → string
GetHint(ctx) → string
GetIconPath(ctx) → string
```

Usage example

```
var utilMgr = app.GetUtilityManager(ref ctx);
var list    = utilMgr.GetListInfo(ref ctx);
int count   = list.Count();
for (int i = 0; i < count; i++)
{
    var info = list.Get(i);
    string uid  = info.GetUid(ref ctx);
    string name = info.GetName(ref ctx);
    Console.WriteLine($"{uid}: {name}");
}

// Execute by UID:
utilMgr.Execute("MyUtility.UID", ref ctx);
```

IIpcLogger

Writes log entries into the ENCY log stream from an external process.

Methods

Method	Description
`log(event: LogItem)`	Writes a fully-populated log entry
`debug(message)`	Writes a `leDebug` entry
`verbose(message)`	Writes a `leVerbose` entry

Method	Description
`info(message)`	Writes a `leInfo` entry
`head(message)`	Writes a `leHead` entry
`warning(message)`	Writes a `leWarning` entry
`error(message)`	Writes a `leError` entry
`IsEventTypeActive(eventType) → boolean`	Whether entries of that severity are currently being recorded
`Notify(eventType, message, title)`	Shows a user-visible notification in the host application, in addition to logging

Use `IsEventTypeActive` to skip building an expensive message that would be discarded:

```
if (logger.IsEventTypeActive(TLogEventType.leDebug))
    logger.debug(BuildExpensiveDump());
```

`Notify` is not just a log write — it surfaces a toast/message to the user, so reserve it for things they must actually see.

`LogItem` is the same struct used in the in-process API — see api/application.md.

> Moved and retyped. `IIpcLogger` now lives in `CAMIPC.IpcInteraction` (it was in `CAMIPC.Helper`), and `IIpcInteraction.Logger` is now typed `IIpcLogger*` instead of `IExtensionLogger*`. Client code that assigned an `IExtensionLogger` to that property, or imported the interface from `CAMIPC.Helper`, must be updated.

```
// Obtain via ICamIpcSingletons:
IIpcLogger logger = /* from singletons */;

logger.info("IPC session started");
logger.warning("Connection latency is high");
logger.error("Failed to load project");

logger.log(new LogItem {
    EventType = TLogEventType.leInfo,
    message   = "structured entry",
```

```
    srcModule = "MyTool"
});
```

ICamIpcXmlPropPointer / ICamIpcXmlPropArray

IPC proxies for navigating the ENCY XML-property tree (the internal data model). These are the IPC equivalents of `IST_XMLPropPointer` / `IST_XMLPropArray`.

ICamIpcXmlPropPointer

Member	Type	Description
`GetInstanceId() → string`	—	IPC instance identifier
`Flt[name]`	`double` RW	Read/write a float child property by name
`Int[name]`	`integer` RW	Read/write an integer child property by name
`Bol[name]`	`boolean` RW	Read/write a boolean child property by name
`Str[name]`	`WideString` RW	Read/write a string child property by name
`Ptr[name]`	`ICamIpcXmlPropPointer*` R	Navigate to a child pointer property
`Arr[name]`	`ICamIpcXmlPropArray*` R	Navigate to a child array property
`CStr[name]`	`WideString` R	Calculated (formula-expanded) string value

```
// Example: read cutting speed from a tech-operation property tree
double speed = op.Ptr["CuttingConditions"].Flt["CuttingSpeed"];

// Write a parameter:
op.Ptr["CuttingConditions"].Flt["FeedRate"] = 0.15;
```

ICamIpcXmlPropArray

Member / Method	Description
`TopItem`	RW — index of the last item (count − 1)
`Itm[index]`	R — get item by index
`AddItem(item) → integer`	Appends an item; returns the new index
`Insert(item, index)`	Inserts before `index`
`Delete(index)`	Removes item at `index`
`ExtractItem(index) → ICamIpcXmlPropPointer*`	Removes and returns item at `index`
`Clear()`	Removes all items
`IndexOf(item) → integer`	Returns the index of an item, or −1
`CreateNewItem(id) → ICamIpcXmlPropPointer*`	Creates a new typed item (if allowed by schema)

```
// Iterate over an array property:
var arr = root.Arr["Operations"];
for (int i = 0; i <= arr.TopItem; i++)
{
    var item = arr.Itm[i];
    string name = item.Str["Name"];
    Console.WriteLine(name);
}
```

ENCY CAM IPC — Project / Technologist Domain

This document covers the CAMIPC variants of the Project / Technologist interfaces. CAMIPC is used when an extension communicates with the ENCY CAM process across a process boundary (inter-process communication).

For a full description of each interface's purpose, properties, and usage patterns, see the CAMAPI reference: ../api/project.md. This document focuses exclusively on what differs in the IPC layer.

Key difference: TExecuteContext

Every IPC method that crosses the process boundary takes a `[in, out] struct TExecuteContext*` parameter. The context carries the call's execution state, error information, and cancellation token. In the .NET helper layer this is handled transparently — you do not construct or pass `TExecuteContext` manually.

In raw IDL / COM you pass a pointer to a stack-allocated `TExecuteContext` on every call:

```
// IDL style (direct COM, not recommended for new code)
TExecuteContext ctx = {};
string id = project->GetId(&ctx);
```

With the .NET helper the context is managed inside `ComWrapper.Invoke` / `InvokeAndWrap`, so the call sites look identical to CAMAPI.

Interface mapping

CAMAPI interface	CAMIPC counterpart
`ICamApiProject`	`ICamIpcProject`
`ICamApiTechnologist`	`ICamIpcTechnologist`
`ICamApiTechOperation`	`ICamIpcTechOperation`
`ICamApiTechOperationIterator`	`ICamIpcTechOperationIterator`
`ICamApiTechOperationType`	`ICamIpcTechOperationType`
`ICamApiTechOperationTypeIterator`	`ICamIpcTechOperationTypeIterator`
`ICamApiPart`	`ICamIpcPart`

CAMAPI interface	CAMIPC counterpart
`ICamApiSetupStage`	`ICamIpcSetupStage`
`ICamApiPartStage`	`ICamIpcPartStage`
`ICamApiPartAndStageList`	`ICamIpcPartAndStageList`
`ICamApiSnapshot`	`ICamIpcSnapshot`
`IListCamApiSnapshot`	`IListCamIpcSnapshot`
`ICamApiUserTechOperationInfo`	`ICamIpcUserTechOperationInfo`
`ICamApiUserTechOperationList`	`ICamIpcUserTechOperationList`

ICamIpcProject

Differences from ICamApiProject

1. All data-returning members are functions, not properties. In CAMAPI, `Id`, `FilePath`, and `Technologist` are COM properties. In CAMIPC they are explicit `GetXxx` functions, each taking `TExecuteContext*`.

CAMAPI property	CAMIPC function
`Id` (property)	`GetId(ctx)`
`FilePath` (property)	`GetFilePath(ctx)`
`Technologist` (property)	`GetTechnologist(ctx)`
`Snapshots` (property)	`GetSnapshots(ctx)`
`NCMaker` (property)	`GetNCMaker(ctx)`
`GeomImporter` (property)	`GetGeomImporter(ctx)`
`IsModified` (property)	`GetIsModified(ctx)`

   `GetIsModified` is a live check of the same condition that triggers the save prompt on close.

2. **`ToolsList`**, **`CoordinateSystems`**, **`MachineInformation`**, **`Machine`**, **`Simulator`**, **`GeomModel`**, **`FeatureFinder`** remain as read-only properties (no `ctx`) on `ICamIpcProject` — they return IPC-

variant types (`ICamIpcMachiningToolsList`, `ICamIpcFeatureFinder`, etc.). `FeatureFinder` is the new feature-recognition axis — see [feature-finder.md]. `SetMachine(guid, filePath, typeName, ctx)` assigns a machine to the project.

`LoadMachineFromXmlProp(ctx)` rebuilds the machine from its current XML properties — the apply step after editing `Machine.XMLProp`, for example flipping an `ActiveNode` selector to add or remove a turn table or a tail stock. Until it is called, the XML and the machine in memory disagree. Unlike `Machine.LoadFromOperationXml`, which reloads the machine object alone, it also rebuilds everything derived from the machine: the initial machine state, the operation enability, the toolpath and the simulation. See [../api/tools-machine.md] for the walkthrough.

3. **`SaveClData`** takes `ICamIpcTechOperationIterator*` (the IPC variant) instead of `ICamApiTechOperationIterator*`.

4. **`RegisterHandler`** / **`UnregisterHandler`** accept `ICamIpcEventHandler*` and `ICamIpcEventListener*` (rather than `ICamApiEventHandler*` and `IListString*`).

5. **`GetInstanceId()`** — every IPC object exposes `GetInstanceId()` (no ctx needed), which returns the stable IPC object identifier used internally for message routing. CAMAPI objects have no equivalent.

IPC-specific event handlers

Two save event handler interfaces exist in CAMIPC, mirroring CAMAPI:

- `ICamIpcHandlerProjectBeforeSave` — `ProjectBeforeSave(handlerIdent, project)`
- `ICamIpcHandlerProjectAfterSave` — `ProjectAfterSave(handlerIdent, project)`

ICamIpcTechnologist

Differences from ICamApiTechnologist

1. All mutating and querying methods take **`TExecuteContext*`**.

CAMAPI	CAMIPC
`GetActiveReorderingModeOfTechnology(out status)`	`GetActiveReorderingModeOfTechnology(ctx)`
`GetActiveReorderingModeOfSimulation(out status)`	`GetActiveReorderingModeOfSimulation(ctx)`
`GetOperations(mode, out status)`	`GetOperations(mode, ctx)`
`CalculateToolpath(links, out status)`	`CalculateToolpath(links, ctx)`

CAMAPI	CAMIPC
`CalculateAllOperationsToolpath(links, out status)`	`CalculateAllOperationsToolpath(links, ctx)`
`ResetToolpath()`	`ResetToolpath(ctx)`
`ResetAllOperationsToolpath()`	`ResetAllOperationsToolpath(ctx)`
`SwitchOperationEnability()`	`SwitchOperationEnability(ctx)`
`GetAvailableOperationTypeIds(out status)`	`GetAvailableOperationTypeIds(ctx)`
`CreateOperation(typeId, afterId, protoId, out status)`	`CreateOperation(typeId, afterId, protoId ctx)`
`CreatePart(externalId, out status)`	`CreatePart(externalId, ctx)`
`CreateSetupStage(out status)`	`CreateSetupStage(ctx)`
`DeleteOperation(id, out status)`	`DeleteOperation(id, ctx)`
`DeletePart(partIdx, out status)`	`DeletePart(partIdx, ctx)`
`DeletePartStage(partIdx, stageIdx, incl, out status)`	`DeletePartStage(partIdx, stageIdx, incl, ctx)`
`DeleteSetupStage(stageIdx, out status)`	`DeleteSetupStage(stageIdx, ctx)`
`CreateOperationFromUserTemplate(userOpId, afterId, out status)`	`CreateOperationFromUserTemplate(userOpId afterId, ctx)`
`GetOperationById(opId, out status)`	`GetOperationById(opId, ctx)`
`MoveOperation(opId, afterId, out status)`	`MoveOperation(opId, afterId, ctx)`

2. **`CurrentOperation`** is read-only in CAMIPC (`propertyR`). To change the current operation use the separate procedure:

   ```
   SetCurrentOperation(operation, ctx)
   ```

   In CAMAPI, `CurrentOperation` is `propertyRW` so it can be set directly.

3. **`GetPartAndStageList`** is a function with `ctx` in CAMIPC:

```
GetPartAndStageList(ctx) → ICamIpcPartAndStageList*
```

In CAMAPI it is a read-only property `PartAndStageList`.

4. **`OperationTypes`** remains a read-only property on `ICamIpcTechnologist` (no ctx), returning `ICamIpcTechOperationTypeIterator*`.

5. **`OperationCatalogList`** — read-only property returning `ICamIpcOperationCatalogList*`, the IPC mirror of the "New operation" window catalogs (see below).

Operation catalogs — ICamIpcOperationCatalogList

IPC mirror of the CAMAPI operation catalog tree (../api/project.md). The tree-walking and mutation methods take `ctx`; scalar properties do not:

- `ICamIpcOperationCatalogList` — `Count`, `Catalog[i]`; `BeginUpdate(ctx)`/`EndUpdate(ctx)` batch, `CreateCatalog(name, ctx)`, `DeleteCatalog(catalog, ctx)`, `Import(zip, ctx)`, `Export(catalog, zip, ctx)`.
- `ICamIpcOperationCatalog` — `Id`, `Name` (RW), `IsActive` (RW, set true to activate); `GetItems(ctx)`, `FindOperation(typeId, ctx)`, `AddGroup(parent, index, caption, ctx)`, `MoveItem(item, newParent, index, ctx)`, `DeleteItem(item, ctx)`.
- `ICamIpcOperationCatalogItem` — `Id`, `Kind` (`TCamApiOperationCatalogItemKind`), `Caption` (RW), `Visible` (RW), `Parent`, `Order`; cast to `ICamIpcOperationCatalogGroup` (`IconFile`) or `ICamIpcOperationCatalogOperation` (`TypeId`, `IsUserOperation`) by `Kind` via `AsInstanceOf`.
- `ICamIpcOperationCatalogItemIterator` — `Next`, `Current`, `Reset`.

All add `GetInstanceId()`.

IPC event handler

`ICamIpcHandlerTechnologistOperationAdded` — fires after a new operation is added:

```
OperationAdded(handlerIdent, operation: ICamIpcTechOperation*)
```

The CAMAPI equivalent is `ICamApiHandlerTechnologistOperationAdded`.

ICamIpcTechOperation

Differences from ICamApiTechOperation

1. All status-flag accessors are functions with **`ctx`** (rather than properties):

CAMAPI property	CAMIPC function
`Id`	`GetId(ctx)`
`Name` (RW)	`GetName(ctx)` / `SetName(name, ctx)`
`Enabled`	`GetEnabled(ctx)`
`Calculated`	`GetCalculated(ctx)`
`Simulated`	`GetSimulated(ctx)`
`IsRapidError`	`GetIsRapidError(ctx)`
`IsHolderError`	`GetIsHolderError(ctx)`
`IsCompensationError`	`GetIsCompensationError(ctx)`
`IsPlungeError`	`GetIsPlungeError(ctx)`
`IsTravelError`	`GetIsTravelError(ctx)`
`IsCollisionError`	`GetIsCollisionError(ctx)`
`IsGougeError`	`GetIsGougeError(ctx)`
`IsToolOverloadError`	`GetIsToolOverloadError(ctx)`
`IsTurnDirectionError`	`GetIsTurnDirectionError(ctx)`
`IsMachiningResultCalculated`	`GetIsMachiningResultCalculated(ctx)`
`IsError`	`GetIsError(ctx)`
`NeedDeleteChips`	`GetNeedDeleteChips(ctx)`

2. **`XMLProp`** returns `ICamIpcXmlPropPointer*` instead of `IST_XMLPropPointer*`. The property-accessor surface (`Bol`, `Int`, `Flt`, `Str`, `Arr`) is the same.

3. **`LoadFromXmlProp`** / **`SaveToXmlProp`** accept / return `ICamIpcXmlPropPointer*` and take `ctx`:

```
LoadFromXmlProp(xmlProp: ICamIpcXmlPropPointer*, ctx)
SaveToXmlProp(ctx) → ICamIpcXmlPropPointer*
```

4. **`ToolChanged`** event handler (`ICamIpcHandlerTechOperationToolChanged`) gains an extra parameter compared to CAMAPI:

   ```
   ToolChanged(tool: ICamIpcMachiningTool*, justPropertyChange: boolean)
   ```

   `justPropertyChange = true` means only a property on the existing tool changed (the tool itself was not replaced). The CAMAPI handler does not have this parameter.

5. Properties that remain as properties (no ctx): `XMLProp`, `Machine`, `WorkpieceCoordinateSystem`, `MachineConfiguration`, `ApproachRule` (RW), `ReturnRule` (RW), `ModelFormerJobAssignment`, `ModelFormerPart`, `ModelFormerWorkpiece`, `ModelFormerRestrictions`, `ModelFormerFixtures`.

6. Missing in CAMIPC vs CAMAPI:

   - `OperationType` (type-id string) — not exposed on `ICamIpcTechOperation`
   - `IsGroup`
   - `FullName`
   - `Units`
   - `LCS`
   - `PartIndex` / `SetupStageIndex`
   - `Technologist` (back-reference)
   - `InitMachineEvaluator`
   - `GetPropIterator`
   - `OperationTag`

   These are available in-process only via `ICamApiTechOperation`.

IPC event handlers on ICamIpcTechOperation

Handler interface	Event	Difference from CAMAPI
`ICamIpcHandlerTechOperationInitModelFormers`	Model formers initialised	Parameter type is `IUnknown*` instead of `ICamApiModelFormer*`
`ICamIpcHandlerTechOperationLoadFromXmlProp`	Properties loaded from XML	Uses `IST_XmlPropPointer*` (same type name as CAMAPI)
`ICamIpcHandlerTechOperationSaveToXmlProp`	Properties saved to XML	Uses `IST_XmlPropPointer*`

Handler interface	Event	Difference from CAMAPI
`ICamIpcHandlerTechOperationToolChanged`	Tool changed	Extra `justPropertyChange: boolean` parameter
`ICamIpcHandlerTechOperationToolpathCalculated`	Body toolpath (re)computed	Receives only `operation` — no `handlerIdent` parameter, unlike the CAMAPI handler

`ToolpathCalculated` fires only on a real body recompute — not for link/approach recomputes, and not when a locked operation keeps its frozen toolpath.

Custom approach/return over IPC

The same structural editor as CAMAPI (`../api/project.md`), but wired differently — worth reading even if you know the in-process API:

CAMAPI	CAMIPC
`GetApproachRuleIsCustom(out rule)` — one call returns the flag *and* a seeded rule	`GetApproachRule(ctx)` and `GetApproachIsCustom(ctx)` — two separate calls
`SetApproachRuleIsCustom(rule)` — takes the rule you built	`ApplyApproachRuleAsCustom(ctx)` — no argument; persists the live list

Over IPC the rule object is live over the operation: mutate its commands in place, then call `ApplyApproachRuleAsCustom` / `ApplyReturnRuleAsCustom` to commit. There is nothing to pass back, because the rule you edited *is* the operation's list.

```
rule := operation.GetApproachRule(ctx)
cmd  := rule.AddCommand(0, ctx)          // 0 = aprtGoto
cmd.SetAxisValue(2, 50.0, ctx)           // Z = 50
cmd.SetAxisAuto(0, ctx)                  // X from the operation's final position
operation.ApplyApproachRuleAsCustom(ctx) // commit
```

`ICamIpcApproachReturnRule`: `GetCommandCount`, `GetCommand(index)`, `AddCommand(positionType)`, `DeleteCommand(index)`, `Clear`.

`ICamIpcApproachReturnCommand`: `GetPositionType` / `SetPositionType`, `GetAxisCount`, and per-axis `GetAxisId`, `GetAxisKind`, `GetAxisMode`, `GetAxisValue`, `SetAxisValue`, `SetAxisAuto`, `SetAxisUnset`.

> Position type, axis mode and axis kind are plain **`integer`** over IPC, not enums — pass and compare the ordinals of `TCamApiApproachReturnPositionType`, `TCamApiApproachReturnAxisMode` and `TCamApiApproachReturnAxisKind`. The value tables are in ../api/project.md.

> `SetPositionType` rebuilds the command's axis set, so set it before the axis values and re-read `GetAxisCount` afterwards.

ICamIpcTechOperationIterator

Differences from ICamApiTechOperationIterator

All navigation functions take `ctx`:

```
MoveToChild(ctx)   → boolean
MoveToSibling(ctx) → boolean
MoveToParent(ctx)  → boolean
Current(ctx)       → ICamIpcTechOperation*
Reset(ctx)
```

There is no `OperationsFilter` property on `ICamIpcTechOperationIterator` (the filter mechanism is CAMAPI-only).

ICamIpcSnapshot / IListCamIpcSnapshot

Differences from ICamApiSnapshot

Both accessors take `ctx` in the IPC variant:

CAMAPI property	CAMIPC function
`CreationTime`	`GetCreationTime(ctx)`
`IsAuto`	`GetIsAuto(ctx)`

`IListCamIpcSnapshot` has the same shape as `IListCamApiSnapshot` (`Get`, `Count`, `Add`, `Remove`, `RemoveAt`) but returns `ICamIpcSnapshot*` items.

ICamIpcPart / ICamIpcSetupStage / ICamIpcPartStage / ICamIpcPartAndStageList

Differences from CAMAPI counterparts

All mutating methods on `ICamIpcSetupStage` take `ctx`:

```
SetToolConnector(toolId, connectorId, ctx)
```

`ICamIpcPart`, `ICamIpcPartStage`, and `ICamIpcPartAndStageList` expose the same properties as their CAMAPI equivalents but gain `GetInstanceId()`.

`ICamIpcPart` does not expose `PrototypePartIndex` or `IsPartCopy` (CAMAPI only).

`ICamIpcPartAndStageList.GetPartStage` has no `ctx` parameter (it is a lightweight local look-up).

ICamIpcUserTechOperationInfo / ICamIpcUserTechOperationList

Differences from CAMAPI counterparts

`ICamIpcUserTechOperationInfo.XMLProp` returns `ICamIpcXmlPropPointer*` instead of `IST_XMLPropPointer*`. It also adds the read-only `BaseOperationTypeId` property (same value as `ICamIpcTechOperationType.Id`), mirroring the CAMAPI addition.

Mutating list operations take `ctx`:

CAMAPI	CAMIPC
`Remove(guid, out status)`	`Remove(guid, ctx)`
`AddFromFile(file, out status)`	`AddFromFile(file, ctx)`
`AddFromOp(caption, op, out status)`	`AddFromOp(caption, op, ctx)`

Async vs sync

All CAMIPC methods are synchronous from the caller's perspective — the .NET helper and the underlying IPC transport handle serialisation. The `TExecuteContext` may carry a cancellation signal, but the call does not return until the remote side has completed or faulted.

There is no callback-based async pattern in the current CAMIPC surface for the Project/Technologist domain.

Concise usage example

The code pattern for IPC is identical to CAMAPI at the helper level. The same `ComWrapper<T>` idiom applies; just use the IPC-typed interfaces:

```
// Both lines look the same — the IPC context is managed inside ComWrapper
using var projectCom      = applicationCom.GetActiveProject();
// ICamIpcProject
using var technologistCom = projectCom.InvokeAndWrap(p => p.GetTechnologist(ctx));

// Create an operation
using var opCom = technologistCom.InvokeAndWrap(tech =>
    tech.CreateOperation("TSTWaterlineOp", afterOpId, "", ctx));

// Read a property (uses ctx internally)
string name = opCom.Invoke(op => op.GetName(ctx));

// Write XML properties (XMLProp property has no ctx; accessor calls do)
using var xmlPropCom = opCom.InvokeAndWrap(op => op.XMLProp);
xmlPropCom.Invoke(xp =>
{
    xp.Bol["Roughing"] = true;
    xp.Flt["RoughingStep"] = 10.0;
});
```

For complete working examples using CAMAPI (which share the same helper pattern), see:

- ../api/project.md — shared concepts and full examples
- ../../FullWorkflow/FullWorkflow3DProject/project/main/TechnologyHelper.cs
- ../../Technologist/Operation/CreateUserOperationNet/project/main/ExtensionCreateUserOperation.cs

Geometry — ENCY CAM IPC

This document describes the IPC (inter-process communication) variants of the geometry interfaces. The IPC layer mirrors the CAMAPI geometry domain but is designed for use in out-of-process extensions that communicate with the ENCY CAM host over an IPC channel.

Read the CAMAPI geometry reference first: `../api/geometry.md`. This document focuses exclusively on the differences between the IPC and API layers.

Table of contents

1. Overview of IPC differences
2. ICamIpcGeomLibrary — factory
3. ICamIpcGeometryModel — model root
4. ICamIpcGeometryTreeNode and iterator
5. ICamIpcGeometryEntity and subtypes
6. ICamIpcFace, ICamIpcLoop, ICamIpcCoEdge
7. ICamIpcSurface and ICamIpcNurbsSurface
8. ICamIpcCurve and related interfaces
9. ICamIpcMesh
10. ICamIpcCoordinateSystem
11. ICamIpcGeometryImporter
12. ICamIpcTurnGeneratrixExtractor — lathe-specific

---

1. Overview of IPC differences

Every IPC interface has two systematic additions compared to its CAMAPI counterpart:

GetInstanceId()

All IPC interfaces expose a `GetInstanceId(): string` method. This returns a unique runtime identifier for the COM proxy object. It is used internally by the IPC transport to route calls to the correct host-side object. You do not normally need to call this from extension code, but it appears on every interface.

TExecuteContext parameter

Any method that performs a blocking or stateful operation in the host process carries an extra `[in, out] TExecuteContext*` parameter. In the CAMAPI (in-process) equivalents this parameter is absent.

CAMAPI (in-process):

```
var iterator = model.GetNodes(out var status);
```

CAMIPC (out-of-process):

```
var iterator = model.GetNodes(ref executeContext);
```

`TExecuteContext` is passed by reference. It carries execution state and error information for the IPC round-trip. Always obtain it from your extension execution context and pass it through.

Naming convention

CAMAPI interface	CAMIPC interface
`ICAMAPIGeometryModel`	`ICamIpcGeometryModel`
`ICAMAPIGeometryTreeNode`	`ICamIpcGeometryTreeNode`
`ICAMAPIGeometryTreeNodeIterator`	`ICamIpcGeometryTreeNodeIterator`
`ICAMAPIGeometryEntity`	`ICamIpcGeometryEntity`
`ICAMAPIGeomLibrary`	`ICamIpcGeomLibrary`
`ICAMAPIGeometryImporter`	`ICamIpcGeometryImporter`
`ICamApiFace`	`ICamIpcFace`
`ICamApiLoop`	`ICamIpcLoop`
`ICamApiCoEdge`	`ICamIpcCoEdge`
`ICamApiCoEdgeIterator`	`ICamIpcCoEdgeIterator`
`ICamApiSurface`	`ICamIpcSurface`
`ICamApiNurbsSurface`	`ICamIpcNurbsSurface`
`ICamApiCurve`	`ICamIpcCurve`
`ICamApiCurve5D`	`ICamIpcCurve5D`
`ICamApiAbstractCurve`	`ICamIpcAbstractCurve`
`ICamApiAbstractNurbsCurve`	`ICamIpcAbstractNurbsCurve`
`ICamApiAbstractCurveReceiver`	`ICamIpcAbstractCurveReceiver`

CAMAPI interface	CAMIPC interface
`ICamApiCurveArcsReceiver`	`ICamIpcCurveArcsReceiver`
`ICamApiMesh`	`ICamIpcMesh`
`ICamApiMeshList`	`ICamIpcMeshList`
`ICamApiCoordinateSystem`	`ICamIpcCoordinateSystem`
`ICamApiListCoordinateSystem`	`ICamIpcListCoordinateSystem`
`ICMAPITurnGeneratrixExtractor`	`ICamIpcTurnGeneratrixExtractor`
`ICamApiFaceList`	`ICamIpcFaceList`
`ICamApiSurfaceCurve`	`ICamIpcSurfaceCurve`
`ICamApiAbstractCurveList`	`ICamIpcAbstractCurveList`

---

2. ICamIpcGeomLibrary — factory

Mirrors `ICAMAPIGeomLibrary`. The same properties (`HideUserMessages`, `VisTolerancePercent`, `SearchFontFolder`) are present with identical semantics.

Key differences:

CAMAPI method	CAMIPC equivalent	Change
`CreateGeometryModel(out status)`	`CreateGeometryModel(ref ctx)`	`TResultStatus` replaced by `TExecuteContext`
`CreateGeometryImporter(out status)`	`CreateGeometryImporter(ref ctx)`	Same
`CreateProjectEtalonFormer(out status)`	*(not present)*	Etalon support is CAMAPI-only
`CreateTurnGeneratrixExtractor()`	`CreateTurnGeneratrixExtractor()`	Unchanged — no context needed

---

3. ICamIpcGeometryModel — model root

Mirrors `ICAMAPIGeometryModel`. All tree-mutation and export methods that were synchronous in CAMAPI now carry a `TExecuteContext` parameter.

Method signature differences

CAMAPI	CAMIPC
`GetNodes(out status)`	`GetNodes(ref ctx)`
`FindByFullName(name, out status)`	`FindByFullName(name, ref ctx)`
`Transform(node, matrix)`	`Transform(node, matrix, ref ctx)`
`ExportSelectedToSTL(fileName, out status)`	`ExportSelectedToSTL(fileName, ref ctx)`
`ExportSelectedToOSD(fileName, out status)`	`ExportSelectedToOSD(fileName, ref ctx)`
`ExportSelectedToDXF(fileName, out status)`	`ExportSelectedToDXF(fileName, ref ctx)`
`DeleteNode(node, out status)`	`DeleteNode(node, ref ctx)`
`AddGroup(node, out status)`	`AddGroup(node, ref ctx)`

Methods that are identical between CAMAPI and CAMIPC: `DeselectAll()`.

Properties `RootNode` and `ActiveNode` have the same semantics. `GetFaceListOfSelected` and `SaveToEtalonReceiver` are not present on `ICamIpcGeometryModel`.

Observing selection changes

`ICamIpcGeometryModel` adds `RegisterHandler(handlerIdent, handler, listener, ref ctx)` / `UnregisterHandler(handlerIdent, ref ctx)`. The handler implements `ICamIpcHandlerSelectionChanged` — `SelectionChanged(handlerIdent, addedNodes, removedNodes)` with the added/removed node full names as `IListString`. This mirrors the CAMAPI selection-change subscription (see `../api/geometry.md`); over IPC the callback is delivered through the standard event-listener mechanism (`ICamIpcEventListener`), like other IPC events.

---

4. ICamIpcGeometryTreeNode and iterator

ICamIpcGeometryTreeNode

Identical properties and navigation links (`Name`, `FullName`, `Selected`, `Visible`, `Parent`, `Child`, `Sibling`, `GeometryEntity`) as `ICAMAPIGeometryTreeNode`, plus `GetInstanceId()`.

ICamIpcGeometryTreeNodeIterator

Identical navigation methods (`MoveToChild`, `MoveToSibling`, `MoveToParent`, `Current`, `Reset`) as `ICAMAPIGeometryTreeNodeIterator`, plus `GetInstanceId()`.

The iterator pattern and depth-first traversal described in ../api/geometry.md §5 applies without change. The only difference is that the iterator is obtained from `ICamIpcGeometryModel.GetNodes(ref ctx)`.

---

5. ICamIpcGeometryEntity and subtypes

ICamIpcGeometryEntity

Same properties as `ICAMAPIGeometryEntity` (`EntityClass`, `EntityType`, `Color`, `ParentColor`, `UpdateStamp`), plus `GetInstanceId()`.

Difference in `GetBoundBox`:

CAMAPI	CAMIPC
`GetBoundBox(lcs, out status): TST3DBox`	`GetBoundBox(lcs, ref ctx): TST3DBox`

Specialised entity subtypes

All three specialised entity interfaces exist in IPC form with the same typed properties, plus `GetInstanceId()`:

Interface	Property	Type
`ICamIpcCSGeometryEntity`	`Matrix`	`TST3DMatrix`
`ICamIpcFaceGeometryEntity`	`Face`	`ICamIpcFace`
`ICamIpcCurveGeometryEntity`	`Curve`	`ICamIpcCurve`

---

6. ICamIpcFace, ICamIpcLoop, ICamIpcCoEdge

All members are identical to the CAMAPI equivalents described in ../api/geometry.md §7. All interfaces add `GetInstanceId()`.

`ICamIpcSurfaceCurve.SaveToReceiver` takes an `ICamIpcAbstractCurveReceiver` instead of `ICamApiAbstractCurveReceiver`, but the method signature is otherwise the same (no

`TExecuteContext`).

`ICamIpcFaceList.GetMesh` has the same four parameters (`i`, `tol`, `holeCappingSize`, `fromThread`) as `ICamApiFaceList.GetMesh`.

Analytic surface recognition and edges

Mirrors the CAMAPI additions in ../api/geometry.md:

- `ICamIpcFace` gains `SurfaceKind` (`TCamIpcSurfaceKind`: `skIpcOther`, `skIpcPlanar`, `skIpcCylindrical`, `skIpcConical`, `skIpcSpherical`, `skIpcToroidal`), plus `GetMinCurvatureRadius(tol)`, `GetPlane`, `GetCylinder`, `GetCone` — same out-parameters and boolean returns as CAMAPI.
- `ICamIpcCoEdge` gains `Edge` → `ICamIpcEdge` (`StartPoint`, `EndPoint`, `Length`).

Edge adjacency — ICamIpcEdgeAnalyzer

IPC mirror of `ICamApiEdgeAnalyzer` (../api/geometry.md) — reconstructs which two faces meet at each edge and classifies it. Same singleton lookup (`GetSingletonExtension` on the IPC extension manager, then cast to `ICamIpcEdgeAnalyzer`).

- `ICamIpcFaceListBuilder` — `Add(face)`, `Build()`, `Count`, `Face[i]`.
- `ICamIpcEdgeAnalyzer` — `CreateFaceList()`, `Build(faces, tolerance)` → `ICamIpcEdgeFaceInfoList`.
- `ICamIpcEdgeFaceInfoList` — `Count`, `Item[i]`, `GetByEdge(edge)`.
- `ICamIpcEdgeFaceInfo` — `FaceA`, `FaceB` (nil for a boundary edge), `IsConvex`, `IsConcave`, `IsSmooth`, `IsBoundary`.

All add `GetInstanceId()`.

---

7. ICamIpcSurface and ICamIpcNurbsSurface

`ICamIpcSurface.GetNurbsForm()` returns `ICamIpcNurbsSurface` (no context parameter). `ICamIpcNurbsSurface` has the same indexed properties as `ICamApiNurbsSurface`, plus `GetInstanceId()`.

---

8. ICamIpcCurve and related interfaces

ICamIpcCurve

Same properties and methods as `ICamApiCurve`, with two exceptions:

Method	CAMAPI	CAMIPC
`MakeStepByLen`	`out ResultStatus`	`ref ctx`
`SavePartToReceiver`	takes `ICamApiAbstractCurveReceiver`	takes `ICamIpcAbstractCurveReceiver`, plus `ref ctx`
`FindNearestPoint`	present	not present in IPC
`Inverse()`	present	not present in IPC

ICamIpcAbstractCurveReceiver

Methods `StartCurve3D`, `StopCurve`, `CutTo3D` all take an additional `ref TExecuteContext` because the receiver implementation lives in the out-of-process extension and each call crosses the IPC boundary.

ICamIpcCurveArcsReceiver

`ArcTo3D` and `AddCircle` both take an additional `ref TExecuteContext` for the same reason.

ICamIpcAbstractCurve and ICamIpcAbstractNurbsCurve

`GetNurbsForm` takes `ref ctx`. `ICamIpcAbstractNurbsCurve` has the same indexed properties as `ICamApiAbstractNurbsCurve`, plus `GetInstanceId()`.

ICamIpcCurve5D

Identical to `ICamApiCurve5D`: `Get_Point5D(t)` with no context parameter.

---

9. ICamIpcMesh

Same five methods as `ICamApiMesh` (`GetVertexCount`, `GetTriangleCount`, `GetVertex`, `GetTriangle`, `GetTriangleNormal`), plus `GetInstanceId()`. No context parameter on any method.

---

10. ICamIpcCoordinateSystem

ICamIpcCoordinateSystem

Same four properties as `ICamApiCoordinateSystem` (`Name`, `Parent`, `Matrix`, `Color`), plus `GetInstanceId()`.

ICamIpcListCoordinateSystem

Same management API as `ICamApiListCoordinateSystem`, but the mutating/querying methods take `ref TExecuteContext`:

Method	CAMAPI	CAMIPC
`Add(name, matrix, parent, ...)`	`out ResultStatus`	`ref ctx`
`Remove(name, ...)`	`out ResultStatus`	`ref ctx`
`GetByName(name, ...)`	`out ResultStatus`	`ref ctx`
`SetActive(name, ...)`	`out ResultStatus`	`ref ctx`

Property name difference: CAMAPI uses `GetActive` (property getter named `GetActive`), CAMIPC uses `Active`.

---

11. ICamIpcGeometryImporter

Mirrors `ICAMAPIGeometryImporter` with two differences:

1. `GeometryModel` is not a direct property. Instead, use `GetGeometryModel(ref ctx)` to retrieve the associated `ICamIpcGeometryModel`.
2. `ImportFile` does not return a `TResultStatus`. Error information is conveyed via the `TExecuteContext`:

CAMAPI	CAMIPC
`GeometryModel` (R/W property)	`GetGeometryModel(ref ctx)`
`ImportFile(path, folder, dialog): TResultStatus`	`ImportFile(path, folder, dialog, ref ctx)`

---

12. ICamIpcTurnGeneratrixExtractor — lathe-specific

This is an IPC-only addition (no direct equivalent in CAMAPI at the same interface level, though `ICMAPITurnGeneratrixExtractor` exists in the CAMAPI layer with identical semantics).

The IPC version differs in:

- `Receiver` property type is `ICamIpcAbstractCurveReceiver` instead of `ICamApiAbstractCurveReceiver`.
- `MakeGeneratrixForNode(node, ref ctx)` — the context parameter is required.

All other properties (`Tolerance`, `SewTolerance`, `NeedJoinCurves`, `TurnAxis`, `NeedCloseToAxis`) have identical types and semantics to the CAMAPI version described in ../api/geometry.md §15.

Obtained from `ICamIpcGeomLibrary.CreateTurnGeneratrixExtractor()` (no context parameter on the factory method itself).

---

13. ICamIpcGeometryModelSketcher — creating primitives

IPC mirror of `ICamApiGeometryModelSketcher` (../api/geometry.md §18). Every method takes `ctx` and returns an `ICamIpcGeometryTreeNode`.

Method	Creates
`AddPoint(x, y, z, ctx)`	A single point
`AddLine(p1, p2, ctx)`	A line segment
`AddNormalLine(origin, endPoint, ctx)`	A "normal line"
`StartPolyline(ctx)`	Polyline → `ICamIpcSpatialCurveBuilder`
`StartSpline(ctx)`	Smooth spline → `ICamIpcSpatialCurveBuilder`

Sketching in a local coordinate system

Coordinates are Global by default. Two ways to change that, same model as CAMAPI:

Sticky — `GetTargetCS(ctx)` / `SetTargetCS(matrix, ctx)` set the CS applied to every subsequent `AddPoint`, `AddLine`, `AddNormalLine`, `StartPolyline` and `StartSpline`. The matrix maps sketch-local coordinates to world, the same convention as `ICamIpcCoordinateSystem.Matrix`; identity means Global.

One-shot — these take the CS explicitly and neither read nor modify `TargetCS`:

Method	Creates
`AddPointInCS(x, y, z, lcs, ctx)`	Point in `lcs`
`AddLineInCS(p1, p2, lcs, ctx)`	Line in `lcs`
`AddNormalLineInCS(origin, endPoint, lcs, ctx)`	Normal line in `lcs`
`StartPolylineInCS(lcs, ctx)`	Polyline builder in `lcs`
`StartSplineInCS(lcs, ctx)`	Spline builder in `lcs`

> Unlike the CAMAPI helpers — which expose these as overloads of the plain names — the IPC interface keeps the explicit `…InCS` method names.

> `TargetCS` is sticky server-side state on the geometry model, shared by every IPC client. Prefer the `…InCS` variants, or restore the previous matrix when done.

CAD — ENCY CAM IPC

IPC variant of the CAD modelling API, for out-of-process apps.

Read the CAMAPI CAD reference first: [../api/cad.md](../api/cad.md). Lifecycle, boolean operations, the `Save()` rule and the pitfalls are the same. This file lists only the differences.

Interfaces: `CADIPC.Model`, `CADIPC.Feature`, `CADIPC.Body`. Helpers: `CADIPC.DotnetHelper`.

---

1. Getting a CAD model

```
using CADIPC.DotnetHelper;
using CAMIPC.DotnetHelper;

using var cadNodeCom = geomCom.AddCadGroup("Profile", identityLcs);   // lcs is
required over IPC
using var cadCom = cadNodeCom.GetCadApiModel();                       //
ICadIpcModel
```

CAMAPI	CAMIPC
`geomCom.AddCadGroup(name, lcs?)`	`geomCom.AddCadGroup(name, lcs)` — no default for `lcs`
`nodeCom.AsCadModel()` → `ICadApiModel`	`nodeCom.GetCadApiModel()` → `ICadIpcModel`

2. Naming

CAMAPI	CAMIPC
`ICadApiModel`	`ICadIpcModel`
`ICadApiSketch`	`ICadIpcSketch`
`ICadApiSketchFace` / `ICadApiSketchContour`	`ICadIpcSketchFace` / `ICadIpcSketchContour`
`ICadApiSegment` / `ICadApiLine` / `ICadApiArc` / `ICadApiPoint`	`ICadIpcSegment` / `ICadIpcLine` / `ICadIpcArc` / `ICadIpcPoint`
`ICadApiBody`	`ICadIpcBody`
`ICadApiFeature` and … `Extrude/Revolve/Fillet/Chamfer/DraftFeature`	`ICadIpcFeature` and … `Extrude/Revolve/Fillet/Chamfer/DraftFeatur`

CAMAPI	CAMIPC
`TCadApiBooleanOperation`	`TCadIpcBooleanOperation` — same values `bo_New … bo_Auto`

Every interface adds `GetInstanceId()`. Raw calls take `ref TExecuteContext` instead of `out TResultStatus`, and properties become `GetX(ref ctx)` / `SetX(value, ref ctx)`.

3. Helpers

Helper names match CAMAPI (`AddSketch`, `Save`, `AddRectangle`, `Face`, `AddExtrude`, `SetOperation`, `SetBodyId`, `Body`, `Id`, `EdgeId`, …). They create the `TExecuteContext` and throw on `rsError`.

Differences:

CAMAPI helper	CAMIPC helper
`AddFillet / AddChamfer / AddDraft(IEnumerable<string> ids, …)`	Take `IListString ids`
`bodyCom.GetBoxMin()` / `GetBoxMax()` → `(X, Y, Z)` tuple	→ `TST3DPoint`
`revCom.GetAxisPoint()` / `GetAxisDirection()` → `(X, Y, Z)` tuple	→ `TST3DPoint`

4. Cut example

```
using var baseSketchCom = cadCom.AddSketch(3, 0);
baseSketchCom.AddRectangle(0, 0, 100, 40);
using var baseFaceCom = baseSketchCom.Face(0);
using var baseExtCom = cadCom.AddExtrude(baseFaceCom, 20);
cadCom.Save();

string baseBodyId;
using (var bodyCom = cadCom.Body(0))
    baseBodyId = bodyCom.Id();

using var cutSketchCom = cadCom.AddSketch(3, 20);
cutSketchCom.AddSlot(20, 20, 80, 20, 6);
using var cutFaceCom = cutSketchCom.Face(0);
using var cutExtCom = cadCom.AddExtrude(cutFaceCom, -8);
cutExtCom.SetOperation(TCadIpcBooleanOperation.bo_Sub);
cutExtCom.SetBodyId(baseBodyId);
cadCom.Save();
```

Feature Finder — ENCY CAM IPC

This document describes the IPC (out-of-process) variant of the feature-recognition interfaces. The IPC layer mirrors the CAMAPI feature finder but is designed for standalone `.exe` applications talking to a running ENCY instance over the IPC channel.

Read the CAMAPI feature-finder reference first: ../api/feature-finder.md. This document covers only the differences between the IPC and API layers.

---

1. Overview of IPC differences

Every IPC interface differs from its CAMAPI counterpart in the same two systematic ways (see `geometry.md §1` for the full explanation):

GetInstanceId()

Every feature interface (`ICamIpcFeatureFinder`, `ICamIpcFeatureList`, `ICamIpcFeature`, and all specialised feature interfaces) exposes `GetInstanceId(): string`. It identifies the COM proxy for IPC routing; extension code does not normally call it.

TExecuteContext parameter

Methods that perform a blocking or stateful host operation carry an extra `[in, out] TExecuteContext*` parameter in place of the CAMAPI `out TResultStatus`. Read-only properties do not. Always pass the context obtained from your connection.

Naming convention

CAMAPI interface	CAMIPC interface
`ICamApiFeatureFinder`	`ICamIpcFeatureFinder`
`ICamApiFeatureList`	`ICamIpcFeatureList`
`ICamApiFeature`	`ICamIpcFeature`
`ICamApiHoleFeature`	`ICamIpcHoleFeature`
`ICamApiComplexHoleFeature`	`ICamIpcComplexHoleFeature`
`ICamApiHoleGrooveFeature`	`ICamIpcHoleGrooveFeature`
`ICamApiPocketFeature`	`ICamIpcPocketFeature`

CAMAPI interface	CAMIPC interface
`ICamApiFilletFeature`	`ICamIpcFilletFeature`
`ICamApiChamferFeature`	`ICamIpcChamferFeature`
`ICamApiPlaneFeature`	`ICamIpcPlaneFeature`
`ICamApiEdgeFeature`	`ICamIpcEdgeFeature`
`ICamApiTangentEdgeFeature`	`ICamIpcTangentEdgeFeature`
`ICamApiHandlerFeatureFinderUpdated`	`ICamIpcHandlerFeatureFinderUpdated`

The enums (`TCamApiFeatureType`, `TCamApiFeatureStatus`, `TCamApiPocketType`, `TCamApiEdgeGeomType`, `TCamApiHoleGrooveShapeType`) are shared with the CAMAPI layer — the IPC interfaces import and reuse them unchanged. See `../api/feature-finder.md §10`.

---

2. Obtaining the feature finder

`ICamIpcProject.FeatureFinder` is a read-only property returning `ICamIpcFeatureFinder` (same access path as CAMAPI, just on the IPC project object).

```
using var projectCom = appCom.GetActiveProject();
using var finderCom  = projectCom.FeatureFinder();   //
ComWrapper<ICamIpcFeatureFinder>
```

---

3. ICamIpcFeatureFinder — method signature differences

Properties `IsUpdating`, `RecognitionProgress`, `UpdateStamp` are unchanged (read-only, no context). All retrieval and recognition methods take `ref TExecuteContext`:

CAMAPI	CAMIPC
`GetFeatures(includeDeleted, out status)`	`GetFeatures(includeDeleted, ref ctx)`
`GetAllFeatures(includeDeleted, out status)`	`GetAllFeatures(includeDeleted, ref ctx)`
`GetSelectedFeatures(out status)`	`GetSelectedFeatures(ref ctx)`
`GetFeatureById(id, out status)`	`GetFeatureById(id, ref ctx)`

CAMAPI	CAMIPC
`RunRecognition(waitForCompletion, out status)`	`RunRecognition(waitForCompletion, ref ctx)`
`GetFeaturesForNode(name, useRef, refMatrix, out status)`	`GetFeaturesForNode(name, useRef, refMatrix, ref ctx)`
`GetFeaturesForSelected(out status)`	`GetFeaturesForSelected(ref ctx)`
`SelectFeatureByBaseEntities(names, type, out status)`	`SelectFeatureByBaseEntities(names, type, ref ctx)`
`CancelRecognition(out status)`	`CancelRecognition(ref ctx)`

`CancelRecognition` aborts an in-progress background recognition (no-op if nothing is running), mirroring the CAMAPI addition.

`ICamIpcFeatureList` (`Count`, `Feature[i]`) and `ICamIpcFeature` (all base properties, `SubFeature[i]`, `BaseEntityName[i]`) carry the same members as their CAMAPI counterparts with no context parameter — they are pure property reads — plus `GetInstanceId()`. `ICamIpcFeature` also adds the `Highlighted` (RW) property — a transient viewport highlight that does not change the actual selection (see ../api/feature-finder.md).

The specialised feature interfaces (`ICamIpcHoleFeature`, `ICamIpcComplexHoleFeature`, `ICamIpcHoleGrooveFeature`, `ICamIpcPocketFeature`, `ICamIpcFilletFeature`, `ICamIpcChamferFeature`, `ICamIpcPlaneFeature`, `ICamIpcEdgeFeature`, `ICamIpcTangentEdgeFeature`) expose the identical measurement properties documented in [../api/feature-finder.md §7](../api/feature-finder.md), each with an added `GetInstanceId()` and no context parameter.

Typical flow

```
using var finderCom = projectCom.FeatureFinder();
finderCom.RunRecognition(waitForCompletion: true, ref ctx);

using var listCom = finderCom.GetFeatures(includeDeleted: false, ref ctx);
int count = listCom.Count();
for (int i = 0; i < count; i++)
{
    using var featureCom = listCom.Feature(i);
    // read Caption / FeatureType / measurements exactly as in the CAMAPI doc
}
```

4. The FeatureFinderUpdated event

`ICamIpcHandlerFeatureFinderUpdated.FeatureFinderUpdated(handlerIdent)` mirrors the CAMAPI handler. Register it through the IPC event mechanism (`RegisterHandler` / `ICamIpcEventHandler`) rather than the in-process `ICamApiApplication.RegisterHandler`; see `connection.md` and `project.md` for the IPC event-listener pattern. Listener callbacks arrive on a background thread — marshal to your UI thread if needed.

Tools & Machine — CAMIPC Reference

This document describes the IPC layer variants of the Tools & Machine interfaces. The IPC layer (`CAMIPC.*`) is used for out-of-process communication with ENCY CAM. All interfaces mirror their `CAMAPI.*` counterparts but operate over IPC transport.

For full property and method descriptions see the CAMAPI reference: docs/api/tools-machine.md.

General IPC Differences

Every IPC interface carries an additional `GetInstanceId()` method that returns the opaque string handle used internally by the IPC message bus. You do not need to call this directly in typical extension code.

Procedures that take an `out TResultStatus` parameter in CAMAPI are replaced in CAMIPC by a `[in, out] struct TExecuteContext*` parameter. The `TExecuteContext` carries both the error status and additional execution context for the IPC transport.

ICamIpcMachiningTool

Library: `CAMIPC.Tools` (uuid `a8203348-327f-4354-b98d-4fb7ea005329`) Interface uuid: `4819106e-f393-4ad5-9af1-15e2a17b91d8`

Identical surface to `ICamApiMachiningTool`: single read-only property `ToolName`.

No .NET helper extension class exists at this layer; use `ComWrapper<ICamIpcMachiningTool>` directly.

ICamIpcMachiningToolInfo

Library: `CAMIPC.ToolsList` (uuid `59644b7b-a5c5-4356-b8c7-a5c03db7c55a`) Interface uuid: `2a27b612-a9a8-4e83-a124-438b42ce8d3e`

Exposes the same properties as `ICamApiMachiningToolInfo`:

Property	Type	Access	Description
`ToolGUID`	`string`	R	Global unique tool identifier
`ToolID`	`string`	R	Project-local tool identifier
`ToolCaption`	`string`	R	Display name
`ConnectorID`	`string`	RW	Machine connector identifier

Property	Type	Access	Description
`MagazineNumber`	`Integer`	R	Magazine slot
`ToolNumber`	`Integer`	R	CNC T-code number
`FirstCorrectorNumber`	`Integer`	R	First CNC corrector table entry
`ToolType`	`string`	R	CAM tool type string
`ToolProperties`	`ICamIpcXmlPropPointer*`	R	XML property tree

Differences from CAMAPI:

- `ToolProperties` returns `ICamIpcXmlPropPointer*` instead of `IST_XMLPropPointer*`. The IPC XML prop interface is the CAMIPC-specific counterpart and carries the same data over the IPC channel.
- `ToolEntity` (`ICamApiMachiningTool*`) is not present in the IPC interface. Tool geometry access is not exposed over IPC.
- `GetInstanceId()` is available for transport-level identification.

ICamIpcMachiningToolsList

Library: `CAMIPC.ToolsList` Interface uuid: `0e64e645-1d06-4155-bdfe-7343ae13fc1c`

Member	Type	Description
`Count`	`Integer` (R)	Number of tools
`ToolInfo[Index]`	`ICamIpcMachiningToolInfo*` (IR)	Tool info at index
`GetInstanceId()`	`string`	IPC handle

Differences from CAMAPI:

- `IndexOfToolID(toolId)` is not present in the IPC interface.
- `GetOperationsUsingTheTool(toolId)` is not present in the IPC interface.

If you need to search by ID or iterate operations via IPC, retrieve the full list and filter client-side.

ICamIpcMachiningToolsManager

Library: `CAMIPC.Tools` Interface uuid: `3bbf4605-5649-4a2b-8266-2781f054fec6`

Method	Signature difference from CAMAPI
`OpenExistingLibrary`	Last parameter is `[in, out] TExecuteContext*` instead of `out TResultStatus`
`AddToolToProject`	Last parameter is `[in, out] TExecuteContext*` instead of `out TResultStatus`
`GetInstanceId()`	Additional IPC-only method

Error handling: inspect `TExecuteContext.ResultStatus` after the call.

ICamIpcMachineInfo

Library: `CAMIPC.Machine` (uuid `4f4116ff-061a-4cff-8b5f-eda2bd7ea0a4`) Interface uuid: `3882595f-caca-4eeb-95df-5efd0bca9cf6`

Surface is identical to `ICamApiMachineInfo`:

Property	Description
`SchemaFilePath`	Path to machine schema file
`XMLNodeName`	Root XML node name
`GUID`	Machine schema GUID
`MachineCaption`	Display name
`MachineTypeName`	XML type identifier

Additional: `GetInstanceId()`.

ICamIpcMachine

Library: `CAMIPC.Machine` Interface uuid: `13b47db3-8362-4292-8310-09ccfe1bd991`

Member	Difference from **`ICamApiMachine`**
`XMLProp`	Returns `ICamIpcXmlPropPointer*`
`WorkpieceConnector[Index]`	Returns `ICamIpcWorkpieceConnector*`

Member	Difference from **`ICamApiMachine`**
`LoadFromOperationXml`	Takes `ICamIpcXmlPropPointer*` and `[in, out] TExecuteContext*`
`ToolPieceConnectorsCount` / `ToolPieceConnector`	Not present in IPC interface
`CreateEvaluator()`	Returns `ICamIpcMachineEvaluator*`

Tool connector enumeration is not available over IPC. If connector information is needed, use the CAMAPI layer (in-process) or obtain it via `ICamIpcMachineInfo` properties.

ICamIpcMachineEvaluator

Library: `CAMIPC.Machine` Interface uuid: `18cc27d0-9d02-4ad6-9c97-2a9a8862991a`

The IPC evaluator exposes only `GetInstanceId()`. The full kinematic API (`CalcNextPos*`, `SetNextPos`, `GetAbsoluteMatrix`, etc.) is not available over IPC.

To perform kinematic calculations, use `ICamApiMachineEvaluator` via the in-process CAMAPI layer.

ICamIpcMachineConfiguration

Library: `CAMIPC.MachineConfiguration` (uuid `c34915c6-39e9-441d-a5bc-96c842c3ca51`) Interface uuid: `687f5a11-67c1-4f1e-b259-26436fa703f5`

Surface matches `ICamApiMachineConfiguration` with the following differences:

- `AxisAvailable` property is not present in the IPC interface. Availability must be inferred by checking whether an axis ID is meaningful for the current machine.
- `SetAxesValues(evaluator, rotaryOnly, TExecuteContext*)` — the last parameter is `[in, out] TExecuteContext*` and the evaluator parameter type is `ICamIpcMachineEvaluator*`.
- `GetInstanceId()` is available.

All flip and axis read/write properties are present with the same semantics as CAMAPI.

Per-operation axis limits

Mirrors the CAMAPI properties (../api/tools-machine.md). All three are indexed by axis, read/write, and take no `TExecuteContext`:

Property	Type	Description
`AxisLimitsEnabled[index]`	`boolean`	`true` — the limits below apply; `false` — inherited from the parent
`AxisLimitMin[index]`	`double`	Lower limit, degrees for rotary axes and mm for linear
`AxisLimitMax[index]`	`double`	Upper limit, same units

> The bounds are only honoured while `AxisLimitsEnabled[index]` is `true`.

ICamIpcMachinesLibrary

Library: `CAMIPC.MachinesLibrary` (uuid `10bb7587-a67a-4374-8b5f-c28dbe5a045f`) Interface uuid: `90bf22a4-5253-46d1-af7f-32e92ef5319b`

Member	Notes
`CurrentMachine`	Returns `ICamIpcMachineInfo*`
`FindMachine(Guid, FilePath, TypeName)`	Parameter names are capitalized vs. CAMAPI but semantics are identical
`GetInstanceId()`	IPC-only

ICamIpcWorkpieceCoordinateSystem

Library: `CAMIPC.Workpiece` (uuid `72a43f75-bda8-42c4-aac5-ff106e77c2e7`) Interface uuid: `5cb4785f-ffdd-4388-b51d-58991080d88b`

Member	Difference from **`ICamApiWorkpieceCoordinateSystem`**
`CoordinateSystemName`	Read-only property (`propertyR`) — write is not available as a property
`SetCoordinateSystemName(value, TExecuteContext*)`	Separate mutating procedure with IPC execute context

Member	Difference from `ICamApiWorkpieceCoordinateSystem`
`GetInstanceId()`	IPC-only

All other members (`ID`, `Mode`, `Offset`) are read/write properties, identical to CAMAPI.

ICamIpcWorkpieceSetup

Library: `CAMIPC.Workpiece` Interface uuid: `29dc5fee-a89a-4650-853a-b7ddb03a5347`

Surface is identical to `ICamApiWorkpieceSetup` with the addition of `GetInstanceId()`. All three properties (`MachineSideConnectorIndex`, `WorkpieceSideCoordinateSystemName`, `Offset`) are read/write.

ICamIpcWorkpieceConnector

Library: `CAMIPC.Workpiece` Interface uuid: `dd26492b-28ab-4468-b10f-e487e7151253`

Single `Name` property (read-only), plus `GetInstanceId()`. Semantics identical to `ICamApiWorkpieceConnector`.

Note: In CAMIPC, `ICamIpcWorkpieceConnector` is defined in `CAMIPC.Workpiece`, whereas in CAMAPI it is defined in `CAMAPI.Machine`. There is no IPC-layer tool connector interface — `ICamApiToolConnector` has no CAMIPC equivalent.

Summary of IPC Limitations

Feature	Available in CAMAPI	Available in CAMIPC
Full tool info (all properties)	Yes	Yes (except `ToolEntity`)
`IndexOfToolID`	Yes	No — filter client-side
Operations-using-tool iterator	Yes	No
Tool connector enumeration	Yes	No
Machine kinematic evaluator (full)	Yes	Stub only (`GetInstanceId`)
`AxisAvailable` on configuration	Yes	No

Feature	Available in CAMAPI	Available in CAMIPC
XML prop write on machine	Yes	Yes (via `ICamIpcXmlPropPointer`)
`CoordinateSystemName` as writable property	Yes	No — use `SetCoordinateSystemName` procedure

NC Generation & Simulation — ENCY CAM IPC

This document covers the IPC (inter-process communication) equivalents of the NC generation and simulation interfaces. IPC interfaces are used when calling ENCY CAM from an external process. Each IPC interface mirrors the corresponding CAMAPI interface but routes calls through the IPC transport layer via a `TExecuteContext` argument.

For full conceptual documentation of what each interface does, see ../api/nc-simulation.md. This document focuses on the IPC-specific differences.

General IPC conventions

- Every mutating method and every method that returns a new object takes a `[in, out] TExecuteContext*` parameter instead of the `out TResultStatus` parameter used in the direct API.
- Every IPC object exposes `GetInstanceId()` which returns the string identifier used in the IPC messaging layer.
- IPC interfaces are defined in the `CAMIPC_*` libraries and mirror their `CAMAPI_*` counterparts.

ICamIpcNCMaker — NC program generation (IPC)

Mirrors `ICamApiNCMaker`. Obtained via the IPC project object.

Postprocessor settings types

The `TCamApiNCMakerSettingsType` enumeration is shared with the direct API:

```
ncsDotnet = 0   // .NET postprocessor
ncsSppx   = 1   // SPPX postprocessor
```

ICamIpcMakeCncSettings

Base settings interface. Provides `GetInstanceId()` for IPC routing.

ICamIpcMakeCncDotnetSettings

Method	Description
`GetSettingsFilePath(ctx)`	Returns the path to the XML settings file
`SetSettingsFilePath(path, ctx)`	Sets the path to the XML settings file

ICamIpcMakeCncSppxSettings

Method	Description
`GetOutputFolder(ctx)`	Returns the output folder
`SetOutputFolder(folder, ctx)`	Sets the output folder
`GetNcFileName(ctx)`	Returns the NC file name
`SetNcFileName(name, ctx)`	Sets the NC file name

ICamIpcNCMaker methods

Method	Description
`GetInstanceId()`	Returns the IPC instance identifier
`CreateSettings(type, ctx)`	Creates settings of the given type; returns `ICamIpcMakeCncSettings*`
`Generate(clDataFile, ppFile, settings, ctx)`	Runs the postprocessor; returns `IListString*` of generated file names

IPC workflow example

```
// Create SPPX settings
var settingsBase = ipcNCMaker.CreateSettings(
    TCamApiNCMakerSettingsType.ncsSppx, ref ctx);
var settings = settingsBase as ICamIpcMakeCncSppxSettings
    ?? throw new Exception("Unexpected settings type");

settings.SetOutputFolder(outputDir, ref ctx);
settings.SetNcFileName("part.nc", ref ctx);

// Generate NC program
var fileList = ipcNCMaker.Generate(clDataFile, ppPath, settingsBase, ref ctx);
```

ICamIpcSimulator — machining simulation (IPC)

Mirrors `ICamApiSimulator`. Obtained via the IPC project object.

Properties (direct, no context required)

All boolean flags and `SimulationSpeedPercent` are read/write properties with no context parameter, identical to the direct API:

Property	Type	Description
`CheckGouges`	bool	Detect gouges
`CheckHolderCollisions`	bool	Detect holder collisions
`CheckMachineCollisions`	bool	Detect machine collisions
`BreakOnStopCommand`	bool	Pause on STOP command
`BreakOnEndOfOperation`	bool	Pause after each operation
`BreakOnErrors`	bool	Pause on simulation errors
`SimulationSpeedPercent`	int	Smooth simulation speed 0–100

Methods (no context)

Fast simulation methods and smooth simulation control methods have no context parameter:

Method	Description
`FastSimulateCurrentOperation()`	Simulate current operation only
`FastSimulateUpToCurrentOperation()`	Simulate from start up to current
`FastSimulateAllOperations()`	Simulate all operations
`ResetSimulationResults()`	Clear all results
`SmoothSimulationStart()`	Begin continuous playback
`SmoothSimulationStop()`	Stop playback
`SmoothSimulationStepForward()`	Step one command forward
`SmoothSimulationStepBackward()`	Step one command backward

SaveMachiningResultToSTL

Takes a `TExecuteContext` in the IPC version:

```
procedure SaveMachiningResultToSTL(
    in(PartStage, ICamIpcPartStage*),
    in(FileName, string),
    [in, out] struct TExecuteContext* ExecuteContext
);
```

```
ipcSimulator.CheckGouges           = true;
ipcSimulator.CheckHolderCollisions = true;
ipcSimulator.ResetSimulationResults();
ipcSimulator.FastSimulateAllOperations();
ipcSimulator.SaveMachiningResultToSTL(null, stlPath, ref ctx);
```

ICamIpcCLDReceiver — toolpath command stream (IPC)

Mirrors `ICamApiCLDReceiver` exactly. The method signatures are identical — no `TExecuteContext` is required on individual toolpath commands. The full command set is documented in ../api/nc-simulation.md#icamapicldreceiver--toolpath-command-stream.

Key methods:

Method	Description
`CutTo(p)`	3-axis linear move
`CutTo5d(p, n)`	5-axis linear move
`ArcTo2d(pe, pc, plane, rc, canBeFull)`	2D arc move
`OutStandardFeed(feed)`	Set feed type by `TFeedTypeFlag` value
`AddSpindleSpeedOnRPM(rpm, range, direction)`	Set spindle RPM
`AddSpindleOff()`	Stop spindle
`AddComment(comment)`	Embed comment
`AddCoolant(onOff, pipeNumber)`	Coolant control

ICamIpcModelFormer family — geometry model assignment (IPC)

Mirrors the `ICamApiModelFormer` family. All `Add…Selected()` methods return IPC list wrappers and operate on the current geometry selection. All mutating methods take a `TExecuteContext`.

ICamIpcModelFormer

Method / Property	Description
`GetInstanceId()`	IPC instance identifier
`SupportedItems`	`ICamIpcModelFormerSupportedItems*`
`MakeSupportedItems(callback, ctx)`	Register supported-items callback
`FillItemsBySupportedItems()`	Apply supported items list
`Count` / `Item[i]`	Enumerate model items
`AddItem(id, type, typeName)`	Add a model item
`FindItem(id)`	Look up an item
`SearchInsertItem(type, id)`	Find or create an item
`DeleteItem(item)`	Remove an item
`DeleteItemById(id, ctx)`	Remove by ID
`AddSupportedItemsSelected(type, id)`	Add selected geometry for a supported type

Specialised model former interfaces

All mirror their `ICamApi*` counterparts. The `GetInstanceId()` method is added to each for IPC routing:

Interface	Key method	IPC difference
`ICamIpcModelFormerWithFaces`	`AddFacesSelected()`	`GetInstanceId(` added
`ICamIpcModelFormerWithLevels`	`SupportsLevel(type)`, `AddLevelSelected(type)`	`GetInstanceId(` added
`ICamIpcModelFormerWithZones`	`SupportsJobZones()`, `AddJobZoneSelected()`, `SupportsRestrictedZones()`, `AddRestrictedZoneSelected()`	`GetInstanceId(` added

Interface	Key method	IPC difference
`ICamIpcModelFormerWithCurve2D`	`AddCurves2DSelected()`	`GetInstanceId(` added
`ICamIpcModelFormerWithCurve5D`	`AddCurves5DSelected()`	`GetInstanceId(` added
`ICamIpcModelFormerWithHoles`	`AddHolesSelected()`, `CreateNewHole(lcs, diameter)`	`GetInstanceId(` added
`ICamIpcModelFormerWithAreas`	`AddAreasSelected(areaMode)`	`GetInstanceId(` added
`ICamIpcModelFormerWithDriveFaces`	`AddDriveFacesSelected()`	`GetInstanceId(` added
`ICamIpcModelFormerWithProjectCurves`	`AddProjectCurvesSelected()`	`GetInstanceId(` added
`ICamIpcModelFormerWithPocket`	`AddPocketSelected()`	`GetInstanceId(` added
`ICamIpcModelFormerWithTurnGeometry`	`AddTurnGeometrySelected()`	`GetInstanceId(` added
`ICamIpcModelFormerWithTurnMachineModel`	`SetItemMode(...)`	`GetInstanceId(` added
`ICamIpcModelFormerWithGeom25D`	`AddGeom25DSelected()`	`GetInstanceId(` added
`ICamIpcModelFormerWithSharpEdge` / `WithChamferFaces`	`AddSharpEdgesSelected()` / `AddChamferFacesSelected()`	`GetInstanceId(` added
`ICamIpcModelFormerWithBoxPrimitives` / `WithCylinderPrimitives` / `WithCastingPrimitive`	`AddBoxPrimitive(...)` / `AddCylinderPrimitive(...)` / `AddCastingPrimitive(...)`	`GetInstanceId(` added
`ICamIpcModelFormerWithReferenceToPrevious`	`SetReferenceToPrevious(flag)`	`GetInstanceId(` added
`ICamIpcModelFormerWithProbingItems`	Full probing-cycle factory surface (see below)	`GetInstanceId(` added

Interface	Key method	IPC difference
`ICamIpcModelFormerWithJobMode`	`JobMode` (RW)	New — swarf "Drive mode", see below

Identifying items — Role and LevelType

`ICamIpcModelItem` adds read-only `Role` (`TModelItemRole`), the flat classification saying which **`Add…Selected`** produced the item — the field to switch on when walking an existing job assignment. `ICamIpcLevelModelItem` adds read-only `LevelType` (`TModelFormerLevelType`) next to its `Stock`, which is how you tell a Top level from a Bottom one.

Both mirror CAMAPI exactly — the value table is in ../api/nc-simulation.md, along with the note that `Caption` is a full hierarchical geometry path that round-trips through `FindByFullName`.

ICamIpcModelFormerWithJobMode — swarf drive mode

Mixin of the swarf operation: `JobMode` (RW, `TModelFormerJobMode`) toggles between `mfjmByBottomEdge` (single guide curve) and `mfjmByTwoCurves`, plus `mfjmNone` when not applicable. The enum is imported from `CAMAPI.ModelFormerTypes`.

> Setting **`JobMode`** clears the current job assignment — set the mode *before* adding curves, or the geometry is discarded. See ../api/nc-simulation.md.

Fixtures — nested nodes

`ICamIpcFixtureNodeItem` gains `NodeCount`, `GetNode(index)` and `AddNode()`. The fixture tree is a chain, not a flat list: a vise is component → Body → Jaw, so the jaw is a child *of the body*, not a second node of the component.

`ICamIpcModelFormerWithFixtures` also mirrors the fixtures library, so a ready-made clamp can be taken whole instead of being assembled node by node:

Member	Description
`LibraryComponentCount`	Number of packages in the fixtures library
`LibraryComponentFile[index]`	Full path of a library package (`.mcp`). Match packages by file name — the caption inside a package may be localized
`ImportComponentFromFile(fileName)`	Import the package (full path, or a bare name resolved against the library folder). Returns

Member	Description
	`ICamIpcFixtureConnectorItem*`, or nil when not found
`FindConnector(caption)`	Find a connector item by caption; nil if there is none
`AddVise()` / `AddClamp()` / `AddNode()`	Build a fixture by hand instead

One `ImportComponentFromFile` call builds the whole sub-tree — connector, component, body and jaw axes with their captions, directions and travel limits — and loads the geometry into a hidden "Fixture model" folder of the geometry tree. See `../api/nc-simulation.md` for the in-process walkthrough.

ICamIpcModelFormerWithProbingItems — probing cycles (IPC)

Mirrors `ICamApiModelFormerWithProbingItems`. Exposes the same factory methods but each returns the IPC-typed cycle interface and takes `ctx` on mutations.

Method (IPC)	Returns
`AddSurfaceCycle(ctx)`	`ICamIpcSurfaceProbingCycle*`
`AddBossCycle(ctx)` / `AddHoleCycle(ctx)` / `AddHoleProtectedCycle(ctx)`	corresponding `ICamIpc…ProbingCycle*`
`AddWebCycle(ctx)` / `AddGrooveCycle(ctx)` / `AddGrooveProtectedCycle(ctx)`	corresponding cycle
`AddThreePointsWebCycle(ctx)`	`ICamIpcThreePointsWebProbingCycle*`
`AddExternalRectangleCycle(ctx)` / `AddInternalRectangleCycle(ctx)` / `AddInternalRectangleProtectedCycle(ctx)`	`ICamIpc…RectangleProbingCycle*`
`AddDoubleWall*CornerCycle(ctx)` / `AddTripleWall*CornerCycle(ctx)`	`ICamIpc…CornerCycle*`
`AddNcActionItem(ctx)` / `AddFrameOutputCycle(ctx)`	`ICamIpcNcActionProbingCycle*` / `ICamIpcFrameOutputProbingCycle*`
`AddMovement(ctx)` / `AddGroup(ctx)` / `AddCycleByTemplate(fileName, ctx)`	`ICamIpcProbingModelItem*`
`GetItems(ctx)`	`ICamIpcProbingModelItemIterator*` (depth-first tree)

Method (IPC)	Returns
`GetSelectedItem(ctx)` / `DeleteItem(item, ctx)` / `Clear(ctx)`	item / void
`GetTemplateLibraryCount()` / `GetTemplateLibrary(index, ctx)`	count / `ICamIpcProbingTemplateLibrary*`

Cycle actions (`ICamIpcProbingAction` with `ICamIpcSetToolOffsetProbingAction`, `CheckBrokenTool`, `SetWcs`, `CalibrateToolProbe`, `CalibratePartProbe`, `WriteToReport`, `CustomPropGroup` specialisations) are accessed via `GetCycleAction(i, ctx)` and added with `AddSetToolOffsetAction(ctx)`, `AddCheckBrokenToolAction(ctx)`, ⋯ on each cycle.

ICamIpcFeedPointList

Mirrors `ICamApiFeedPointList`. Feed-zone mutation methods take a `TExecuteContext`:

```
AddFeedPoint(position, length, ctx) → index
RemoveFeedPoint(index, ctx)
Clear(ctx)
```

Read-only indexed properties (`TStart`, `TEnd`, `Position`, `FeedType`, `FeedRatePercentage`, `FeedRateChangeType`) require no context.

IDL reference

Interface	Library	GUID
`ICamIpcNCMaker`	`CAMIPC_NCMaker`	`8f7d03a1-bee1-4f44-8798-deea797ba41f`
`ICamIpcMakeCncDotnetSettings`	`CAMIPC_NCMaker`	`82cf324d-0b79-4657-8b51-1d939951deb3`
`ICamIpcMakeCncSppxSettings`	`CAMIPC_NCMaker`	`aceb72e3-f0a3-4ddb-a9c6-1bec6b8fbe73`
`ICamIpcSimulator`	`CAMIPC_Simulator`	`abb53f60-55a5-4f65-a133-fd32e37bc775`
`ICamIpcCLDReceiver`	`CAMIPC_MCDFormerTypes`	`ea1f4c02-0292-4a97-9c6f-719ffc088f23`

Interface	Library	GUID
ICamIpcModelFormer	CAMIPC_ModelFormerTypes	fb0ad4d1-87fc-4a44-9b34-cdb8f8199d9b
ICamIpcModelFormerWithFaces	CAMIPC_ModelFormerTypes	450bc7d7-f75f-4eea-bbd1-816717bbe6c8
ICamIpcModelFormerWithLevels	CAMIPC_ModelFormerTypes	fe8a4a8a-8d0b-4de6-afcc-a6e1043ec4e6
ICamIpcModelFormerWithZones	CAMIPC_ModelFormerTypes	6cf8367e-feb3-4066-ba73-df66f60add3e
ICamIpcModelFormerWithCurve2D	CAMIPC_ModelFormerTypes	7dd5100d-cf82-4acf-af31-dc2b89e39b49
ICamIpcModelFormerWithCurve5D	CAMIPC_ModelFormerTypes	40d47f5a-45e8-43a2-bcde-86c1fda8f842
ICamIpcModelFormerWithHoles	CAMIPC_ModelFormerTypes	c2e303e5-d44f-434b-bbd4-bea00ce12c2e
ICamIpcModelFormerWithAreas	CAMIPC_ModelFormerTypes	e1f2a3b4-c5d6-4e7f-8a9b-0c1d2e3f4a5b
ICamIpcFeedPointList	CAMIPC_ModelFormerTypes	958a4b7e-e9f7-4e09-9e22-b4ae4ecde9f7

ENCY CAM IPC API — UI Domain

This document covers the out-of-process UI interfaces for controlling the ENCY main window, 3D viewport, view cube, and the Prime multi-instance shell.

All IPC UI methods that communicate with a remote ENCY process take a `TExecuteContext` parameter. Read-only properties that are cached locally may not require one.

Table of Contents

1. ICamIpcApplicationMainForm — remote main window
2. ICamIpcViewPort — remote 3D viewport
3. ICamIpcViewCube — remote view cube
4. ICamIpcPrimeView — Prime shell UI actions
5. ICamIpcPrimeViewModel — Prime shell lifecycle and instance management
6. ICamIpcListPrimeViewModel — Prime instance list

ICamIpcApplicationMainForm

The IPC proxy for the ENCY main application window. Obtained via `ICamIpcApplication.MainForm` or `ICamIpcApplication.GetMainFormTimeOut(timeout)`.

Event registration

```
RegisterHandler(handlerIdent, handler, listener, ctx)
UnregisterHandler(handlerIdent, ctx)
```

Handler interfaces for the main form (IPC versions mirror the in-process set):

Interface	Method
`ICamIpcHandlerApplicationMainFormIsVisibleChanged`	`ApplicationMainFormIsVisibl`
`ICamIpcHandlerApplicationMainFormMinimizeChanged`	`ApplicationMainFormMinimize minimized)`

Interface	Method
`ICamIpcHandlerApplicationMainFormUiInfoChanged`	`ApplicationMainFormUiInfoCh`
`ICamIpcHandlerApplicationMainFormSaveInCloudClicked`	`ApplicationMainFormSaveInCl`
`ICamIpcHandlerApplicationMainFormShareClicked`	`ApplicationMainFormShareCli`
`ICamIpcHandlerApplicationMainFormCloudChatClicked`	`ApplicationMainFormCloudCha`
`ICamIpcHandlerApplicationMainFormReloadCloudProjectClicked`	`ApplicationMainFormReloadCl`

State methods

Method	Description
`GetVisible(ctx) → boolean`	Whether the main window is visible
`GetUiInfo(ctx) → ICamIpcMainFormUiInfo*`	Current UI state snapshot

Project management

Method	Description
`OpenProject(fileName, ctx)`	Opens a project file
`OpenProjectSnap(fileName, snapName, ctx)`	Opens a project snapshot
`SaveCurrentProject(ctx)`	Saves in-place

Method	Description
`SaveCurrentProjectAs(ctx)`	Shows the Save As dialog
`SaveProjectAs(fileName)`	Saves to a specified path without a dialog
`SaveProject()`	Saves without a dialog
`ExportCurrentProject(ctx)`	Exports with all snapshots
`ExportProjectWithHistory(fileName)`	Saves with full snapshot history
`ExportDrillPoints(fileName)`	Exports selected drill operation points
`CreateNewProject(ctx)`	Creates a new empty project
`SaveAsMachineSetup(fileName)`	Saves as a machine setup file
`ImportMachineSetup(fileName)`	Imports a machine setup
`ExportSimulationResults(fileName)`	Exports simulation results

Application actions

Method	Description
`BeginFreeze(freezeType)`	Prevents user interaction (`TFreezeInterfaceType`)
`EndFreeze()`	Restores user interaction
`RunAppSetup(out settingsChanged)`	Opens the application settings dialog
`RunUtilitiesSetup(anchorRect)`	Opens the utilities configuration popup. `anchorRect` (`TCamApiRect`) is the screen rectangle of the button that triggered it, so the popup opens directly underneath; pass an empty rect (all zeroes) to center instead
`OpenAiAssistant()`	Opens the AI assistant
`ShowHelpContents()`	Shows the help documentation
`SupportRequest()`	Opens the support request dialog
`ShowTutorialWnd()`	Shows the tutorial window

Method	Description
`ShowSnapshotManager(projectPath)`	Shows the snapshot manager
`CrashReport()`	Opens the crash report dialog
`OpenDotnetInterpreterInVSCode(fileName)`	Opens a .NET script in VS Code
`SetCurrentInPrime(value)`	Notifies the form that it is the active Prime instance

PLM and cloud

Method	Description
`LoadProjectFromPLM(connectionId)`	Imports from PLM
`ExportProjectToPLM(connectionId)`	Exports to PLM
`ConnectPLMConnection(connectionId)`	Connects a PLM connection
`DisconnectPLMConnection(connectionId)`	Disconnects a PLM connection
`CloudsCollabShare()`	Shares in cloud collaboration
`CloudsCollabDownload()`	Downloads the latest cloud version
`CloudsCollabOpenChat()`	Opens cloud collaboration chat

Viewport access

```
GetMainViewPort(ctx) → ICamIpcViewPort*
```

Main window handle

Returns the OS window handle (`HWND`, 64-bit value) of the ENCY main window. Use it as the parent window for modal dialogs opened by out-of-process tooling.

```
GetMainWindowHandle(ctx) → int64
```

Plugin hotkeys

```
GetHotkeyManager(ctx) → ICamIpcHotkeyManager*
```

IPC mirror of the CAMAPI hotkey manager (../api/application.md). Over IPC the manager keeps the binding table only — a hotkey is added by shortcut and caption in one call; the command-fire callback is not part of this manager (fire notifications go through the application event-listener mechanism, not a per-hotkey `OnExecute`).

Status-bar progress — ICamIpcProgressIndicator

```
CreateProcessIndicator(caption, ctx) → ICamIpcProgressIndicator*
```

Drives ENCY's own status-bar progress indicator, so an out-of-process tool can report progress in the host UI instead of its own window. IPC mirror of `ICamApiProgressIndicator`.

Method	Description
`Show()`	Occupies the status bar and shows the indicator. Fails if another process already holds it
`Hide()`	Hides the indicator and releases the status bar
`SetPercent(value)`	Completion 0..100. Fails unless this indicator currently holds the status bar
`SetCaption(value)`	Text shown next to the indicator. Same precondition
`GetPercent()` / `GetCaption()`	Current values
`RegisterHandler(ident, handler, listener, ctx)`	Subscribe an `ICamIpcHandlerProgressIndicator` to Show/Hide/Break/Progress
`UnregisterHandler(ident)`	Remove the subscription

The handler receives `ProgressIndicatorEvent(handlerIdent, eventType)` where `eventType` is `TProgressIndicatorEventType`: `pietShow` (0), `pietHide` (1), `pietBreak` (2), `pietProgress` (3). **`pietBreak`** is the user pressing Break — that is how a cancel request reaches your loop.

> The status bar is a single shared resource. Always `Hide()` when done (or on failure), otherwise it stays occupied for every other client.

Verify compare — ICamIpcVerifyCompareManager

```
GetVerifyCompareManager(ctx) → ICamIpcVerifyCompareManager*
```

Controls the "Verify compare" mode — machining result against the part, coloured by deviation. IPC mirror of `ICamApiVerifyCompareManager`.

Method	Description
`GetEnabled()` / `SetEnabled(value)`	Whether compare mode is active
`GetTolerance()` / `SetTolerance(value)`	Half-width of the green "in tolerance" band, mm
`GetScale()` / `SetScale(scale)`	Deviation scale as `TCamApiVerifyCompareCompareScale` (`Stock`, `PosInner`, `PosOuter`, `NegInner`, `NegOuter`)
`GetMeasureEnabled()` / `SetMeasureEnabled(value)`	Whether clicking a point reports its local deviation

> `GetScale` returns only the edge levels per side and `SetScale` takes `abs()` and sorts what you pass — see the CAMAPI note before doing a read-modify-write.

Script editor — ICamIpcScriptEditor

```
OpenScriptEditor(ctx) → ICamIpcScriptEditor*
```

Opens ENCY's script editor (Script IDE) and returns its handle. The only method is `Load(scriptPath, ctx)`, which loads a `.spr` script project/file and shows it for editing.

- `ICamIpcHotkeyManager` — `AddShortcut(shortcut, caption, ctx)` (fails if taken), `RemoveShortcut(shortcut, ctx)` (fails for reserved), `FindByShortcut(shortcut, ctx)`, `GetCount(ctx)`, `GetHotkey(index, ctx)`.
- `ICamIpcHotkey` — `GetShortcut(ctx)`, `GetCaption(ctx)`/`SetCaption(caption, ctx)`, `GetEnabled(ctx)`/`SetEnabled(enabled, ctx)`, `GetIsReserved(ctx)`. The `Shortcut` is fixed at creation.

Native application shortcuts are pre-registered as reserved entries, so `FindByShortcut` reports conflicts against them.

ICamIpcMainFormUiInfo — UI state snapshot

Returned by `GetUiInfo(ctx)`. Equivalent to the in-process `ICamApiMainFormUiInfo` with one addition:

Method	Returns	Description
`GetAppProcessId()`	`integer`	Process ID of the ENCY instance this info belongs to
`GetCaption()`	`string`	Current window title
`GetProjectFile()`	`string`	Path to the open project file
`GetDisplayPath()`	`string`	Path as shown in the UI
`GetStorageType()`	`string`	Storage type identifier
`GetProjectIsModified()`	`bool`	Whether the project has unsaved changes
`GetProjectIsNew()`	`bool`	Whether the project has never been saved
`GetProcessExists()`	`bool`	Whether a background process is running
`GetProcessStage()`	`int`	Background process completion percentage
`GetCloudsCollabIsEnabled()`	`bool`	Whether cloud collaboration is available
`GetCloudsIsShared()`	`bool`	Whether the project is shared in the cloud
`GetCloudsIsOutdated()`	`bool`	Whether a newer cloud version exists
`GetCloudsHasNewMessages()`	`bool`	Whether new cloud chat messages exist
`GetCloudsLastUpdateTime()`	`string`	Timestamp of the last cloud update
`GetUiPLMInfo()`	`ICamIpcMainFormUiPLMInfo*`	PLM connection states

Usage example

```
// Get the main form proxy (wait up to 5 seconds):
var mainForm = app.GetMainFormTimeOut(5_000);

// Open a project:
mainForm.OpenProject(@"C:\projects\part.encam", ref ctx);

// Get UI state:
var uiInfo = mainForm.GetUiInfo(ref ctx);
bool modified = uiInfo.GetProjectIsModified();
string caption = uiInfo.GetCaption();

// Freeze during a batch operation:
mainForm.BeginFreeze(TFreezeInterfaceType.afiiGeneral);
try { /* ... */ }
finally { mainForm.EndFreeze(); }

// Access viewport:
var viewPort = mainForm.GetMainViewPort(ref ctx);
viewPort.ZoomAll(fluently: true, ref ctx);
```

ICamIpcViewPort

IPC proxy for the 3D viewport.

Methods

Method	Description
`GetScaleFactor(ctx) → double`	Current zoom factor
`GetViewMode(ctx) → TViewPortViewMode`	Current rendering mode
`ZoomAll(fluently, ctx)`	Fits all geometry into view
`GetMatrix(ctx) → TST3DMatrix`	Reads the view transformation matrix
`SetMatrix(value, ctx)`	Applies a view transformation matrix immediately
`SetMatrixFluently(matrix, ctx)`	Applies a matrix with animation
`GetViewBox(ctx) → TST2DBox`	Reads the view bounding box
`SetViewBox(value, ctx)`	Applies a view bounding box immediately

Method	Description
`SetViewBoxFluently(box, ctx)`	Applies a view bounding box with animation
`GetCube(ctx) → ICamIpcViewCube*`	Returns the view cube proxy

TViewPortViewMode values — see api/ui.md.

Usage example

```
var vp = mainForm.GetMainViewPort(ref ctx);

// Zoom fit:
vp.ZoomAll(fluently: true, ref ctx);

// Read the current matrix:
TST3DMatrix m = vp.GetMatrix(ref ctx);

// Apply a new matrix:
vp.SetMatrix(m, ref ctx);

// Snap the view cube to top:
var cube = vp.GetCube(ref ctx);
cube.Rotate(TViewCubeRotateMode.vcrmFaceTop, ref ctx);
```

ICamIpcViewCube

IPC proxy for the orientation cube widget.

```
Rotate(mode: TViewCubeRotateMode, ctx)
```

`TViewCubeRotateMode` has the same 26 values as in the in-process API — all Faces (6), Edges (12), and Corners (8). See api/ui.md for the complete list.

```
var cube = vp.GetCube(ref ctx);

cube.Rotate(TViewCubeRotateMode.vcrmFaceFront, ref ctx);
cube.Rotate(TViewCubeRotateMode.vcrmEdgeTopFront, ref ctx);
cube.Rotate(TViewCubeRotateMode.vcrmCornerTopFrontRight, ref ctx);
```

ICamIpcPrimeView

The UI-side interface of the Prime shell. Represents the view layer (the actual WPF/Win32 window of the Prime process). Exposes a single action:

```
PressCreateNewApplication()
```

This programmatically triggers the "create new ENCY instance" button in the Prime window — the same as the user clicking the + button.

```
// Obtain ICamIpcPrimeView from the Prime process proxy:
primeView.PressCreateNewApplication();
```

ICamIpcPrimeViewModel

The view-model of a Prime shell instance. Manages the lifecycle of all ENCY application instances embedded in that Prime window.

Properties

Property	Type	Description
`ProcessId`	`integer`	OS process ID of the Prime process

Instance management

Method	Description
`CreateApplication(args, ctx) → integer`	Launches a new ENCY instance inside this Prime window; returns the new process ID
`GetApplicationsList(ctx) → IListInteger*`	Lists the process IDs of all ENCY instances managed by this Prime
`CloseApplication(processId, ctx)`	Gracefully closes a managed ENCY instance
`KillApplication(processId, ctx)`	Forcibly terminates a managed ENCY instance
`ApplicationBelongs(processId, ctx) → boolean`	Returns true if the given process is managed by this Prime
`Close(ctx)`	Closes the entire Prime shell

Project and start-page integration

Method	Description
`ParkStartPageMainWindow(startPageAppId, mainWindowId, ctx)`	Embeds a start-page window into the Prime shell
`StartPageOpenProject(projectFile, ctx)`	Instructs the start page to open a project
`GetRecentProjects() → string`	Returns a JSON array of recently opened project paths

Modal and blocking state

Method	Description
`EnterModalState(appId)`	Called when an embedded ENCY instance enters a modal dialog
`LeaveModalState(appId)`	Called when the modal dialog closes
`EnterBlockPrjChange(appId)`	Called when a project change must be blocked
`LeaveBlockPrjChange(appId)`	Called when the block on project change is lifted

Usage example

```
// Create a new ENCY instance in the Prime window:
int newPid = primeViewModel.CreateApplication("--headless", ref ctx);
Console.WriteLine($"New ENCY instance PID: {newPid}");

// List all managed instances:
var pids = primeViewModel.GetApplicationsList(ref ctx);
for (int i = 0; i < pids.Count(); i++)
    Console.WriteLine($"Instance: {pids.Get(i)}");

// Open a project in a specific instance:
var app = ipcList.GetByProcessId(newPid, ref ctx);
app.WaitForStarted(15_000, out var status);
app.OpenProject(@"C:\projects\part.encam", false, ref ctx);

// Close one instance:
primeViewModel.CloseApplication(newPid, ref ctx);
```

```
// Close the entire Prime shell:
primeViewModel.Close(ref ctx);
```

ICamIpcListPrimeViewModel

A collection of `ICamIpcPrimeViewModel` instances. Used to enumerate and manage all running Prime shells.

```
Count: integer
Get(index, ctx) → ICamIpcPrimeViewModel*
GetByProcessId(processId, ctx) → ICamIpcPrimeViewModel*
Add(item)
RemoveByProcessId(processId)
RemoveAt(index)
Clear()
```

Relationship between Prime and application instances

```
ICamIpcListPrimeViewModel          — all Prime shells
  └── ICamIpcPrimeViewModel        — one Prime shell (one process)
        └── ICamIpcListApplication — all ENCY instances in that Prime
              └── ICamIpcApplication — one ENCY instance
                    └── ICamIpcApplicationMainForm
                          └── ICamIpcViewPort
                                └── ICamIpcViewCube
```

In non-Prime deployments (ENCY launched standalone), each `ICamIpcApplication` is obtained directly from the IPC singletons without going through a Prime view model.

Additional IPC Interfaces

This page covers three specialised IPC interface groups that go beyond the core `ICamIpcApplication` API: cloud project integration, functional test automation, and PLM connection management.

All three follow the same general pattern as the core IPC layer: obtain the interface through `ICamIpcApplication` (or a cast thereof), pass `ref TExecuteContext` on every call, and check `ResultStatus` afterwards.

CAMIPC.CloudsApp — Cloud Project Integration

Defined in: `CAMIPC.CloudsApp.idl`

These interfaces are used by the ENCY cloud service to push cloud-state changes (sharing status, chat notifications, version staleness) into a running ENCY instance over IPC. They are typically consumed by the cloud client process, not by general plugin code.

ICloudsIpcApplication

Obtained by casting `ICamIpcApplication` to `ICloudsIpcApplication` (if the running instance supports cloud features).

```
// Cast the connected application to ICloudsIpcApplication (perform the cast
inside Invoke
// — never against the raw .Instance, which is not safe across the MTA boundary).
using var cloudAppCom = applicationCom.InvokeAndWrap(app => app
as ICloudsIpcApplication);
if (cloudAppCom.IsNull)
    throw new Exception("This ENCY instance does not support cloud integration.");

var ctx = new TExecuteContext();
using var cloudProjectCom = cloudAppCom.InvokeAndWrap(app =>
    app.GetCloudsProject(ref ctx));
if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception("GetCloudsProject: " + ctx.ResultStatus.Description);
```

`GetCloudsProject` always returns an `ICloudsIpcProject` instance, even if the active project is not currently shared on the cloud.

ICloudsIpcProject

`ICloudsIpcProject` provides write-only setters that push state from the cloud service into ENCY's UI:

Method	Description
`SetPid(pid, ctx)`	Registers the PID of the cloud client process with ENCY
`SetIsShared(isShared, ctx)`	Marks the project as shared (or not) in ENCY's title bar and toolbar
`SetIsOutdated(isOutdated, ctx)`	Signals that a newer version is available on the cloud server
`SetHasNewMessages(hasNew, ctx)`	Triggers the "new chat messages" indicator in ENCY's UI
`SetLastUpdateTime(fileTime, ctx)`	Updates the "last synced" timestamp displayed in ENCY

```
var ctx = new TExecuteContext();

// Tell ENCY the project is now shared
cloudProjectCom.Invoke(p => p.SetIsShared(true, ref ctx));
if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception(ctx.ResultStatus.Description);

// Tell ENCY there is a newer version on the server
cloudProjectCom.Invoke(p => p.SetIsOutdated(true, ref ctx));
```

> Note: The IDL comment marks `ICloudsIpcApplication` as a temporary interface pending the cloud subsystem's own dedicated IPC mechanism. Treat it as internal API subject to removal in a future version.

CAMIPC.FunctionalTest — Automated Test Runner

Defined in: `CAMIPC.FunctionalTest.idl`

`IIpcFunctionalTest` exposes the operations needed to drive ENCY's built-in functional test infrastructure from an external .NET test harness. It allows loading projects, running scenarios defined in `.stjob` files, running toolpath simulation, comparing output against reference ("etalon") files, and generating new reference files.

Obtaining IIpcFunctionalTest

```
// Cast the application to IIpcCamAppTests first
using var testAppCom = applicationCom.InvokeAndWrap(app => app as IIpcCamAppTests);
if (testAppCom.IsNull)
```

```
    throw new Exception("This ENCY instance does not expose the test interface.");

var ctx = new TExecuteContext();
using var testCom = testAppCom.InvokeAndWrap(a => a.CreateFunctionalTest(ref ctx));
if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception("CreateFunctionalTest: " + ctx.ResultStatus.Description);
```

IIpcFunctionalTest API

```
// Prepare a clean project state
testCom.Invoke(t => t.NewProject(ref ctx));

// Import a geometry or project file
testCom.Invoke(t => t.Import(@"C:\TestData\Part1.igs", ref ctx));

// Run a .stjob scenario file (contains scripted operations, toolpath
generation, etc.)
testCom.Invoke(t => t.Run(@"C:\TestData\Scenarios\FaceMilling.stjob", ref ctx));
if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception("Scenario failed: " + ctx.ResultStatus.Description);

// Simulate the entire toolpath
testCom.Invoke(t => t.SimulateAll(ref ctx));

// Generate a new etalon (reference output) at the given path
testCom.Invoke(t => t.MakeEtalon(@"C:\TestData\Etalons\FaceMilling", ref ctx));

// Compare current output against the etalon; IgnoreItems filters paths to skip
IListString? ignoreItems = null; // pass null to compare everything
testCom.Invoke(t => t.CompareEtalon(@"C:\TestData\Etalons\FaceMilling", ignoreItems,
ref ctx));
if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception("Etalon mismatch: " + ctx.ResultStatus.Description);
```

Additional methods:

Method	Purpose
`SetMultiprocessorMode(bool, ctx)`	Enable/disable parallel toolpath calculation
`SetSimulationByGCode(bool, ctx)`	Toggle G-code-based vs. toolpath-based simulation

Method	Purpose
`ResetSimulation(ctx)`	Clear simulation state
`SimulateAllSmooth(ctx)`	Run smooth (interpolated) simulation
`RunScript(scriptPath, ctx)`	Execute a standalone script file inside ENCY
`SaveStcx(ctx)`	Save the current project as `.stcx`
`CompareEtalonSources(etalonPath, ignoreItems, ctx)`	Compare source files (XML/JSON operation data) only
`CheckActiveProjectHasNoIncorrectChars(ctx)`	Validate project data for encoding issues

Typical Test Harness Pattern

```
public static void RunTest(ComWrapper<ICamIpcApplication> applicationCom, string
stjobPath, string etalonPath)
{
    using var testAppCom = applicationCom.InvokeAndWrap(app => app
as IIpcCamAppTests);
    if (testAppCom.IsNull)
        throw new Exception("Test interface not available.");

    var ctx = new TExecuteContext();
    using var testCom = testAppCom.InvokeAndWrap(a => a.CreateFunctionalTest(ref
ctx));
    Check(ctx, "CreateFunctionalTest");

    testCom.Invoke(t => t.NewProject(ref ctx));
    Check(ctx, "NewProject");

    testCom.Invoke(t => t.Run(stjobPath, ref ctx));
    Check(ctx, $"Run({Path.GetFileName(stjobPath)})");

    testCom.Invoke(t => t.SimulateAll(ref ctx));
    Check(ctx, "SimulateAll");

    testCom.Invoke(t => t.CompareEtalon(etalonPath, null, ref ctx));
    Check(ctx, "CompareEtalon");
}

private static void Check(TExecuteContext ctx, string step)
```

```
{
    if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
        throw new Exception($"{step} failed: {ctx.ResultStatus.Description}");
}
```

CAMIPC.Extension.PLM — PLM Connection Management

Defined in: `CAMIPC.Extension.PLM.idl`

`IIpcPLMManager` allows an external process to discover PLM integration connections registered in a running ENCY instance. PLM connections link ENCY projects to external Product Lifecycle Management systems (e.g., Teamcenter, Windchill).

Obtaining IIpcPLMManager

```
using var plmManagerCom = applicationCom.InvokeAndWrap(app => app
as IIpcPLMManager);
if (plmManagerCom.IsNull)
    throw new Exception("PLM manager interface not available in this
ENCY instance.");
```

IIpcPLMManager API

```
// Get the IPC instance identifier of this PLM manager
string instanceId = plmManagerCom.Invoke(m => m.GetInstanceId());

var ctx = new TExecuteContext();

// Find the first registered connection for a given PLM extension type
// Returns connection ID string, or empty string if none found
string connectionId = plmManagerCom.Invoke(m =>
    m.FindFirstConnectionByExtensionType("com.example.plm.teamcenter", ref ctx));
if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception("FindFirstConnectionByExtensionType: " +
ctx.ResultStatus.Description);

if (string.IsNullOrEmpty(connectionId))
    throw new Exception("No Teamcenter connection configured in this
ENCY instance.");

// Get all connections for a given extension type as a JSON string
string connectionsJson = plmManagerCom.Invoke(m =>
    m.GetConnectionsByExtensionType("com.example.plm.teamcenter", ref ctx));
```

```
if (ctx.ResultStatus.Code == TResultStatusCode.rsError)
    throw new Exception("GetConnectionsByExtensionType: " +
ctx.ResultStatus.Description);

// Parse connectionsJson with System.Text.Json as needed
```

The `ExtensionTypeId` parameter is the unique identifier string registered by the PLM extension plugin in ENCY's extension registry. It is defined by the PLM extension author and documented in that extension's specification.

`GetConnectionsByExtensionType` returns a JSON-serialised array of connection descriptors. The exact schema depends on the PLM extension type; deserialise with `System.Text.Json.JsonSerializer` or `Newtonsoft.Json` as appropriate for your integration.