AmiumScripter – Modular UI & Signal Framework

Comprehensive Documentation & Architecture  
  
Includes: Concept, Page Pattern, Signal Management, Thread/Task Utilities, and Tutorials.

# 1. AmiumScripter Concept & Architecture

AmiumScripter – Architecture and Concept Description

# 1. Purpose

AmiumScripter is a modular, dynamic UI and signal framework for technical and scientific applications. It focuses on maximum flexibility, runtime dynamism, and a clean separation of logic, visualization, and control.

# 2. Component Overview

* a) Projects & Pages

- A project consists of multiple "Pages" – functional units with their own logic and interface.  
- Each page lives in its own directory (Pages/PageName/) and consists of three files:

• Page.cs: Workflow, business logic, signal definitions (no UI)  
 • controls.cs: Visual controls, pure layout (generated/editable in the editor)  
 • view.cs: UI logic (behavior, events, visual logic)

* b) Central Control

- Project.cs: Manages all Pages and Views, instantiation, initialization, start/stop.  
- UIEditor: Visual editor for controls, tab management, drag & drop.  
- SignalManager/DataStorage: Central management and pooling of all signals.

* c) Dynamic Build System

- Uses Roslyn to compile and load Pages and Controls at runtime.  
- Dynamic assemblies allow new/edited Pages to be available immediately in the running system.

* d) Thread, Task and Class Management

- AThread and ATask provide centralized management and lifecycle control for background jobs. They enable monitoring, registration, and clean termination of all running threads and tasks.  
- The TokenManager enables controlled, global cancellation of background operations during project rebuilds or unloads.  
- ClassManager (BaseClass):  
 - The ClassManager automatically registers all classes that inherit from BaseClass (such as workers, managers, or other runtime objects).  
 - All active BaseClass instances are centrally listed and managed, making it possible to enumerate, monitor, and clean up all objects during destroy/rebuild cycles.  
 - During destroy or rebuild, ClassManager ensures that all BaseClass instances are notified and can release resources, stop operations, or close open handles.

# 3. Core Principles & Advantages

- Separation of logic, layout, and UI behavior: increases maintainability, testability, and reusability.  
- Signal-based data model: All data flows via centrally managed signals.  
- Runtime dynamism: Changes to controls, pages, layouts possible without app restart.  
- Clean resource management: Centralized handling of all tasks, threads, and tokens.  
- Automated refactoring enforcement: Build checker prevents critical coding patterns.

# 4. Technical Interplay

Loading process:  
1. Build/Compile: All pages/views/controls are compiled dynamically.  
2. Project.cs: Instantiates objects, adds them to dictionaries, calls Initialize()/Run().  
3. UIEditor: Binds the views into the UI (tabs).  
4. SignalManager: Provides the data pools.  
5. Background threads are started via AThread/ATask and can be stopped centrally at any time.

# 5. Typical Data and Signal Flow Diagram

The following diagram shows the interaction between Page, View, Controls, and the central SignalPool:

TBA

# 6. Extensibility

- New pages can be added via editor or code.  
- Controls can be added or modified via drag & drop, code, or editor.  
- Signal connectivity can be integrated via custom SignalClients/AClients.  
- Robustness through hot-rebuilds, central unload, and build checker.

# 7. Best Practices

- All logic involving threads/tasks/signals should use the helpers (AThread, ATask, TokenManager).  
- Dumb controls, smart views, intelligent pages.  
- No business logic in the UI, no UI logic in the page.  
- For "while(true)", always use IsRunning or CancellationToken.

# Conclusion

AmiumScripter offers a highly flexible, robust, and team-ready architecture for modular, dynamic applications with clear separation of all responsibilities.

# 2. Dynamic Page Concept

Dynamic Page Concept – UI & Logic Separation

This document describes the architectural concept for dynamic pages in the AmiumScripter framework. The design separates business logic, control structure, and UI behavior for maximum flexibility, reusability, and maintainability.

# Concept Overview

Each Page consists of three files and separates core responsibilities:

1. 1. Page.cs – Page Logic

- Contains business logic, data models, thread or timer logic.  
- Defines and manages all relevant signals (BaseSignal and derivatives).  
- Does NOT contain any direct UI code or WinForms controls.

1. 2. controls.cs – Control Declarations

- Declares only the visual controls (e.g., SignalView, buttons, grids) and their layout.  
- Can be modified by the App UI Editor or by code.  
- Contains no logic or event handling – only declarative UI structure.

1. 3. view.cs – UI Behavior

- Contains all logic affecting visual controls (highlighting, event binding, custom behaviors).  
- Does not define data logic or initialize controls, but orchestrates their behavior and appearance.

# Benefits

- Clean separation of concerns: Business logic, UI structure, and UI logic are separated.  
- Enables dynamic (re-)building of UI without touching core logic.  
- Allows for programmatic and visual UI editing.  
- Promotes code reuse, modularity, and easy maintenance.

# Component Responsibilities

# Architectural Diagram

The following diagram shows the relationships and data flow between the components:

+-----------+  
 | Page.cs |  
 +-----------+  
 | ^  
Signals, Data | | Uses signals, logic  
------------------+ |  
 v |  
 +----------------+  
 | view.cs |  
 +----------------+  
 | ^  
Events/ | | Control instances  
Behavior | |  
 v |  
 +--------------+  
 | controls.cs |  
 +--------------+

# Summary

This concept provides a robust, scalable foundation for dynamic, modular WinForms UI design, where logic, layout, and behavior are clearly separated for maintainability and extensibility.

# 3. Working with Signals

Working with Signals: Integration Patterns

In this system, there are three main ways to interact with signals. Understanding these patterns helps you write reliable and maintainable code when handling data streams and signal management.

# 1. Direct Creation

If you create a new signal directly in your code using the Signal constructor, the signal is automatically registered in the SignalPool and becomes globally available throughout the system.

Example:

public Signal MySignal = new Signal(  
 name: "mySignal",  
 text: "MySignal",  
 unit: "°C",  
 format: "0.00",  
 value: 23.5  
);

# 2. Registration via AClient

When using an AClient, your code delegates the registration to the SignalManager, which then adds the signal to the SignalPool. No explicit Signal instance exists in your code—all registration and management is handled centrally.  
  
The SignalManager will also normalize the signal name to PascalCase. For example: myClient.Sig1 will become MyClient\_Sig1.

Example:

MyAClient.Push(new Signal(  
 name: "myClient.Sig1",  
 unit: "°C",  
 format: "0.00",  
 value: 23.5  
));

# 3. Access by Name

You can retrieve a reference to any already-registered signal by its name from the SignalPool. This does not create a new signal instance; it simply provides a reference (pointer) to the existing signal object.

Examples:

// Static access  
public Signal MySignal = SignalPool.MyClient\_Sig1;  
  
// Dynamic access  
public Signal MySignal = SignalPool["MyClient\_Sig1"];

Summary:  
- Direct creation provides you with a code-visible signal instance.  
- AClient registration manages signals centrally, without direct instances in your code.  
- Access by name is used to get a reference to any existing signal in the pool.

(Optional: Add a diagram or table to visualize these relationships if needed.)

# 4. Thread, Task and Token Management

## 4.1 AThread – Managed Thread Lifecycle

AThread – Managed Thread Lifecycle Utility

AThread provides a safe, registered, and controllable wrapper for background thread execution in .NET/C#. It integrates automatic registration with a central ThreadsManager, cooperative cancellation, and logging. This ensures that all threads can be monitored, stopped, and cleaned up—especially useful in modular, reloadable apps.

# Key Features

- Named thread instances for traceability  
- Registration and deregistration with ThreadsManager  
- Cooperative cancellation via CancellationTokenSource  
- Clean start/stop methods and status queries  
- Exception handling and logging for thread lifecycle events

# AThread Code Example

Here is the class implementation:

public class AThread  
{  
 public string InstanceName { get; init; }  
 private Thread \_thread;  
 private CancellationTokenSource \_cts = new();  
 public bool IsRunning => \_thread.IsAlive && !\_cts.IsCancellationRequested;  
  
 public AThread(string instanceName, Action work, bool isBackground = true)  
 {  
 InstanceName = instanceName;  
  
 \_thread = new Thread(() =>  
 {  
 try { work(); }  
 catch (OperationCanceledException)  
 {  
 Logger.Log($"[AThread] {InstanceName} cancelled.");  
 }  
 catch (Exception ex)  
 {  
 Logger.Log($"[AThread] {InstanceName} error: {ex.Message}");  
 }  
 });  
 \_thread.IsBackground = isBackground;  
  
 ThreadsManager.Register(this);  
 Logger.Log($"[AThread] Registered: {InstanceName}");  
 }  
  
 public void Start()  
 {  
 if (\_thread.ThreadState == ThreadState.Unstarted)  
 {  
 \_thread.Start();  
 Logger.Log($"[AThread] Started: {InstanceName}");  
 }  
 }  
  
 public void Stop()  
 {  
 if (\_thread == null || !\_thread.IsAlive)  
 return;  
  
 Logger.Log($"[AThread] Stop requested: {InstanceName}");  
  
 \_cts.Cancel();  
 if (!\_thread.Join(2000))  
 {  
 Logger.Log($"[AThread] Still running after Cancel: {InstanceName} — trying Interrupt...");  
 \_thread.Interrupt();  
  
 if (!\_thread.Join(1000))  
 {  
 Logger.Log($"❌ [AThread] Cannot stop thread {InstanceName} cleanly.");  
 throw new InvalidOperationException($"Thread {InstanceName} refused to stop.");  
 }  
 }  
  
 Logger.Log($"✅ [AThread] Cleanly stopped: {InstanceName}");  
 ThreadsManager.Deregister(this);  
 }  
}

# How to Use

1. Create a new AThread with a name and a work method:  
 var myThread = new AThread("Worker1", () => { while(IsRunning) { ... } });  
2. Start the thread with myThread.Start();  
3. Stop it any time with myThread.Stop();  
All instances are automatically registered for management and can be safely terminated on app shutdown or project rebuild.

# Best Practices

- Always check IsRunning in long-running work loops to allow cooperative cancellation.  
- Use ThreadsManager to monitor and stop all threads if required.  
- Prefer AThread over direct Thread usage for resource safety and app lifecycle management.

# Diagram: Thread Lifecycle

TBA

# Summary

AThread is designed to make multithreaded code safer and more maintainable in dynamic, modular apps.

## 4.2 ATask – Managed Task Utility

ATask – Managed Task Lifecycle Utility

ATask provides a managed, registered, and cancellable abstraction for background task execution in .NET/C#. It supports both fire-and-forget and result-returning asynchronous operations, with centralized registration for lifecycle management and safe cleanup.

# 1. Key Features

- Named task instances for traceability  
- Registration/deregistration with TaskManager  
- Cooperative cancellation via CancellationToken  
- Both fire-and-forget and awaitable generic variants  
- Exception handling and logging

# 2. Variants

## ATask – Fire-and-Forget

public class ATask  
{  
 public string InstanceName { get; }  
 private CancellationTokenSource \_cts = new();  
 private Task \_task;  
  
 public bool IsRunning => !\_task.IsCompleted && !\_cts.IsCancellationRequested;  
  
 public ATask(string instanceName, Action<CancellationToken> work)  
 {  
 InstanceName = instanceName;  
 \_task = new Task(() => work(\_cts.Token), \_cts.Token);  
 TaskManager.Register(this);  
 }  
  
 public void Start()  
 {  
 if (\_task.Status == TaskStatus.Created)  
 \_task.Start();  
 }  
  
 public void Stop()  
 {  
 \_cts.Cancel();  
 TaskManager.Deregister(this);  
 }  
}

Usage Example (Fire-and-Forget):

var myTask = new ATask("Worker1", token => {  
 while (!token.IsCancellationRequested)  
 {  
 DoSomeWork();  
 }  
});  
myTask.Start();  
// ... later:  
myTask.Stop();

## ATask<TResult> – Awaitable with Return Value

public class ATask<TResult>  
{  
 public string InstanceName { get; }  
 private CancellationTokenSource \_cts = new();  
 private Task<TResult> \_task;  
  
 public bool IsRunning => !\_task.IsCompleted && !\_cts.IsCancellationRequested;  
  
 public ATask(string instanceName, Func<CancellationToken, TResult> work)  
 {  
 InstanceName = instanceName;  
 \_task = new Task<TResult>(() => work(\_cts.Token), \_cts.Token);  
 TaskManager.Register(this);  
 }  
  
 public void Start()  
 {  
 if (\_task.Status == TaskStatus.Created)  
 \_task.Start();  
 }  
  
 public async Task<TResult> WaitAsync()  
 {  
 return await \_task;  
 }  
  
 public void Stop()  
 {  
 \_cts.Cancel();  
 TaskManager.Deregister(this);  
 }  
}

Usage Example (Result/awaitable):

var calcTask = new ATask<int>("Adder", token => {  
 int sum = 0;  
 for (int i = 0; i < 100 && !token.IsCancellationRequested; i++)  
 sum += i;  
 return sum;  
});  
calcTask.Start();  
int result = await calcTask.WaitAsync(); // returns 4950

# 3. Best Practices

- Always use the provided CancellationToken to exit loops and support cooperative cancellation.  
- Register all tasks for lifecycle management and shutdown.  
- Prefer ATask and ATask<TResult> over direct Task usage for better safety and app management.

# 4. Task Lifecycle Diagram

TBA

# 5. Summary

ATask makes asynchronous and background code safer and more maintainable in dynamic, modular applications.

## 4.3 TokenManager – Centralized Cancellation

Mini-Tutorial: Using TokenManager for Robust Task Cancellation

To ensure that all running tasks, threads, or asynchronous operations can be properly cancelled and cleaned up during rebuilds or unloads, use a central TokenManager. This approach helps you avoid resource leaks and ensures system stability, especially in dynamic environments.

# 1. Creating and Using Cancellation Tokens

Use TokenManager.AToken() whenever you start a long-running task, async operation, or thread. This will register a new CancellationTokenSource internally and provide a CancellationToken for use in your code.

Example:

public static async Task ASleep(int delay)  
{  
 await Task.Delay(delay, TokenManager.AToken());  
}

# 2. Cancelling All Tasks During Rebuild/Unload

When you perform a rebuild or unload your project, call TokenManager.CancelAll(). This will cancel all registered tokens and ensure that all dependent tasks or threads are stopped gracefully.

Example:

TokenManager.CancelAll();

# 3. Best Practices

- Always acquire your CancellationToken from TokenManager for any task that should be cancellable.  
- Ensure your async methods, threads, or loops regularly check for token.IsCancellationRequested.  
- Document this requirement for all developers working on the project.

# 4. Example: Complete TokenManager Implementation

public static class TokenManager  
{  
 private static readonly List<CancellationTokenSource> \_sources = new();  
  
 public static CancellationToken AToken()  
 {  
 var cts = new CancellationTokenSource();  
 \_sources.Add(cts);  
 return cts.Token;  
 }  
  
 public static void CancelAll()  
 {  
 foreach (var cts in \_sources)  
 cts.Cancel();  
 \_sources.Clear();  
 }  
}

# Summary

Using a central TokenManager ensures all background work can be cancelled safely and reliably, especially during reloads or shutdowns. This prevents resource leaks and keeps your application robust.

# 5. Mini-Tutorials & ConversionExtensions

# Mini-Tutorial: ConversionExtensions – Useful Conversion & Manipulation Methods

This mini-tutorial summarizes all public methods available in the static class ConversionExtensions. It provides concise descriptions, parameter details, and return values to help you integrate these helpers into your own codebase quickly.

## ToInt(this string text)

Converts a string to an integer by parsing it as a double and then casting to int. If conversion fails, returns 0.

* Parameters:

- text: The input string to convert.

* Returns:

- Integer value, or 0 if conversion fails.

Example:  
var i = "3.5".ToInt(); // i == 3

## ToDouble(this string text)

Converts a string to a double value. Supports numbers in decimal, hexadecimal (e.g. "FFH", "0xFF"), binary (e.g. "1010B"), and certain keywords ("LONG", "BYTE"). Invalid or unrecognized inputs return double.NaN.

* Parameters:

- text: The input string to convert.

* Returns:

- Parsed double value, or double.NaN if conversion is not possible.

## Tolerant(this string text)

Returns a tolerant (normalized) version of the string: all uppercase, spaces/tabs removed, and German umlauts replaced (e.g., "Ä" -> "AE"). Returns null if input is null.

* Parameters:

- text: The input string (nullable).

* Returns:

- Normalized string, or null if input is null.

## RemoveInvalidCharsFromFilename(this string filename)

Replaces all invalid filename characters in the input string with a dot ('.').

* Parameters:

- filename: The file name to sanitize (not null).

* Returns:

- A safe filename with all invalid characters replaced.

## RemoveRedundantBrackets(string orig)

Removes redundant outer square brackets from a string if present. Returns the original string if no redundant brackets are found.

* Parameters:

- orig: Input string (can be null or empty).

* Returns:

- String without redundant outer brackets.

## ToImage(this string base64String)

Decodes a Base64-encoded string and returns an Image object. Returns null if decoding fails or input is null/empty.

* Parameters:

- base64String: The Base64 string representing image data.

* Returns:

- Image object or null if conversion fails.

## ChangeBrightness(this Color color, double percent)

Adjusts the brightness of a color. Positive percent lightens, negative percent darkens. Returns black if operation fails.

* Parameters:

- color: The original color.

- percent: Brightness adjustment (positive = lighter, negative = darker).

* Returns:

- Color with adjusted brightness, or black if failed.

## ChangeTransparency(this Color color, double percent)

Changes the alpha channel of a color based on the percent parameter (0 = fully transparent, 100 = opaque). Returns black if input is out of range or operation fails.

* Parameters:

- color: The original color.

- percent: Transparency as a percentage (0-100).

* Returns:

- Color with new alpha value, or black if failed.

## ToColor(this object color)

Converts an object to a System.Drawing.Color. Accepts color names (e.g. "Red"), hex codes (e.g. "#FF0000"), or Color objects. Returns black for unsupported types or if conversion fails.

* Parameters:

- color: The input object (Color or string).

* Returns:

- Corresponding Color object, or black if conversion fails.

## ToBase64String(this Image image, ImageFormat format)

Converts an Image to a Base64 string using the specified image format (e.g. PNG, JPEG). Returns null if conversion fails.

* Parameters:

- image: The image to convert.

- format: The image format.

* Returns:

- Base64 string, or null if conversion fails.

## IsBase64ImageStringIsValid(string base64)

Checks if a string is a syntactically valid Base64 string. Returns true only if the string matches Base64 format.

* Parameters:

- base64: String to validate.

* Returns:

- true if valid Base64, otherwise false.

## ToFormatedBase64String(this Image image, int width = 0, int height = 0, RotateFlipType flipType = RotateFlipType.RotateNoneFlipNone, ImageFormat format = null)

Converts an Image to a Base64 string after optional scaling and rotation. Width and height specify the target size; both zero uses the original size. Format defaults to JPEG. Returns null if conversion fails.

* Parameters:

- image: The image to convert.

- width: Target width in pixels (optional).

- height: Target height in pixels (optional).

- flipType: Image rotation/flip (optional, default: none).

- format: Image format (optional, default: JPEG).

* Returns:

- Base64 string of the processed image, or null if failed.

## Summary

ConversionExtensions provides a set of handy extension methods for robust string, color, and image conversion. These utilities simplify parsing, normalization, and format conversion in various .NET projects.

- Place external libraries in dlls for seamless project integration.

- Use Pages\<PageName>\Classes for encapsulated logic only needed by one page.

- Use Shared\Classes for code you want globally accessible to all pages.

Best Practice:

- This is ideal for page-specific logic, data structures, or helpers you do not want to expose project-wide.

- All C# files placed in this folder are compiled with the page, but are only available within that page (due to namespace scoping).

- Every page directory (e.g., Pages\TestPage\Classes) may contain its own Classes subfolder.

Page-Local Classes (Pages\<PageName>\Classes)

During the build process, all DLLs found here will be automatically referenced and made available to your dynamic pages and project code.

You can integrate third-party or custom .NET libraries by simply copying their DLL files into this folder.

- dlls:

External Libraries Directory (dlls)

Central location for resources (e.g., images, fonts, localization files) shared throughout the project.

- Shared\Ressources:

All C# classes placed here are available to every page in the project. This is the recommended place for business logic, models, helpers, or utilities that should be reused across pages.

- Shared\Classes:

Every project contains a Shared directory that is accessible to all pages. It has typically the following subfolders:

Project Structure: Shared Resources, Classes and DLLs