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

## 1.1 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.

## 1.2 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, Token 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.

## 1.3 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.

## 1.4 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.

## 1.5 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.

## 1.6 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.

## 1.7 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.

## 2.1 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.

## Architectural Diagram

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

## 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.

## 3.1 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  
);

## 3.2 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  
));

## 3.3 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, Token and Class 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.

### 4.1.1. 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

#### 4.1.2. 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.

#### 4.1.3. 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.

#### 4.1.4. 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.

### 4.2.1. 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

### 4.2.2. Variants

- ATask

- ATask<TResult>

#### 4.2.3. ATask

Usage Example (Fire-and-Forget):

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

#### 4.2.4 ATask<TResult> – Awaitable with Return Value

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

### 4.2.3 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.

### 4.2.4 Summary

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

## 4.3. AToken() – Managed Token Handling

Using AToken() 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.

### 4.3.1 Creating and Using Cancellation Tokens

Use 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, AToken());  
}

### 4.3.2 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.

### 4.3.3. 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.

### 4.3.4 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.

## 4.4. ClassManager – Centralized Destory

TBA