Copyright 2026 Synthiam Inc.

Create Robot Skill

Create

This manual walks through making a robot skill for ARCx: what a robot skill is, how to create the project, build and try it, write it, test it, and publish it to the Robot Skill Store. It is the walkthrough, not the reference. Two companion manuals sit beside it and hold the detail:

- `RobotSkill-Services-And-Core.md` - everything a worker can call: the robot controllers, video, audio, speech, movement, kinematics, navigation, depth cameras, the Robot Bus, scripting, variables, logging, configuration, and the worker overrides themselves.
- `RobotSkill-UI-Components.md` - everything visual: the three UI base classes, layout, redrawing, every `UC*` component, Custom UI widgets.

`ARCx.xml`, beside `ARCx.dll`, is the authoritative per-member reference for both.

It describes ARCx as of October 2026 (template version 2026.09.27.00). ARCx is an ASP.NET Core Blazor Server application on .NET 10 with a MudBlazor UI. A robot skill is a .NET class library that ARCx loads at run time.

Contents

1. What a robot skill is
2. Before you start
3. Step 1: Create the project
4. The project on disk
5. Step 2: Build and try it
6. Step 3: Write the skill
7. Step 4: Test it
8. Step 5: Publish it
9. Troubleshooting

---

1. What a robot skill is

A robot skill is a plugin that adds one capability to a robot project: a camera, a servo controller, a speech engine, a tracking algorithm, a scripting surface, a connection to an external system. Users add robot skills to a project from the Add Robot Skill menu, arrange their windows on the workspace desktops, configure them, and save the whole thing as an `.arcx` project file.

A robot skill assembly contains up to five kinds of class. ARCx finds them by base type, so their names do not matter, but the template's names are used throughout this manual.

Class	Base type	Required	Role
`RobotSkillWorker`	`RobotSkillWorkerAbstractEntity`	Yes	The logic. Owns state, background work, hardware access, configuration and control commands. Exactly one instance per robot skill in the project.
`RobotSkillUIMain`	`RobotSkillUIMainBaseEntity` (Blazor	Yes	The window on the workspace. One

	component)		instance per connected browser.
`RobotSkillUIConfig`	`RobotSkillUIConfigBaseEntity` (Blazor component)	No	The settings page, shown when the user opens the skill's configuration.
`RobotSkillUIWidget`	`RobotSkillUIWidgetBaseEntity` (Blazor component)	No	A compact piece of the skill the user can place on their own Custom UI screens.
`Configuration`	none	Convention	The settings, serialized as JSON into the project file.

The worker is the source of truth. UI components are transient views over it: browsers connect and disconnect, and several browsers can show the same skill at once, so nothing that must persist or keep running belongs in a UI component.
ARCx drives the worker through a fixed order: constructor (services injected), `Setup()`, `Initialize()`, `OnSetConfiguration()`, `OnProjectLoadCompleted()`, then run time, and finally `DisposeOverride()` when the skill is removed. Section 6 says what goes in each; `RobotSkill-Services-And-Core.md` section 1 has the full detail.

---

2. Before you start

- A Synthiam account, signed in to ARCx (**Options, Account**). Creating a skill registers it in the store under that account.
- The .NET 10 SDK, and Visual Studio or any editor that builds .NET projects.
- Optionally, an AI coding assistant. ARCx writes a prompt and an `AGENTS.md` for it, and an assistant connected to ARCx's MCP server can build, install and test the skill against the running ARCx for you.

---

3. Step 1: Create the project

In ARCx open **Robot Skills, Create** (the green button in the top bar). The Create tab asks for:

- **Name**: 5 or more characters; letters, digits, spaces, parentheses, hyphen and period only; and it must not contain "ARCx" in any letter case. It is title-cased, and becomes the store title, the project folder name and (with spaces replaced by underscores) the namespace and assembly name.
- **Company / Organization**: 5 or more characters.
- **Description**: 30 or more characters. Shown in the store and the Add Robot Skill list.
- **Categories**: 1 to 3, chosen from the store's live category list.
- Agreement to the robot skill terms.

The author name comes from your signed-in account.
When you press Create, ARCx:

1. Registers the robot skill with Synthiam and receives a **store GUID**. The GUID identifies your skill forever: it names the install folder, and it is what projects save to find the skill again. Nobody else sees the skill until you upload a build and make it public.
2. Creates `My Documents\ARCx Data\Robot Skill Projects\<Name>\` (`~/ARCx Data/Robot Skill Projects/<Name>/` on Linux and macOS). It refuses a folder that already holds a `plugin.json`, before anything is registered.
3. Writes `plugin.json` from the details the store returned.
4. Unpacks the project template into the folder, with the template's namespace, GUID,

plugin folder and ARCx application folder replaced by yours.
5. Writes `AGENTS.md`, which explains ARCx, the project files, where the manuals and `ARCx.xml` are on this machine, and the rules a robot skill follows, so any AI coding assistant opened on the folder later has the platform explained to it.
6. Opens the project folder and moves to the **created** page.

The created page lists what was made, the next steps, and an AI prompt built from your description. It is rebuilt from `plugin.json`, so you can come back to it later. If ARCx's MCP server is on, it also says how to let an AI finish the skill through MCP.

---

4. The project on disk

```
<Name>\
  <Name>.csproj
  plugin.json              store manifest; edit Version here before each upload
  AGENTS.md                the platform explained to an AI coding assistant
  _Imports.razor           the @using lines every .razor file gets
  Configuration.cs         the settings class
  RobotSkillWorker.cs      the logic
  RobotSkillUIMain.razor   the window (+ .razor.cs)
  RobotSkillUIConfig.razor the settings page (+ .razor.cs)
  RobotSkillUIWidget.razor optional Custom UI widget; delete it and the GetCustomUIWidgets()
override together
  wwwroot\                 created empty; ARCx does not serve it
```

There is deliberately **no `title.png`**. It is the thumbnail the store shows, and it is yours to draw: a square PNG, 256x256 or larger, in the project folder. Nothing needs it until you upload; until then every build warns "This robot skill has no thumbnail".
The template compiles and runs as it is: a timer, a counter, a log, a configuration page with a text field, a number field and a servo selector, control commands, a Custom UI widget, and an MCP tool and resource. Replace the demo with your skill and keep the file layout and base classes.

plugin.json

```
{
  "Name": "Face Tracking",
  "AuthorName": "Synthiam Inc.",
  "CompanyName": "Synthiam Inc.",
  "GUID": "d8294feb-405a-41b6-aab1-b795db1f3b43",
  "Version": 3,
  "Description": "Track faces from any of the ARC video sources.",
  "Categories": [ "Camera" ],
  "DLLFilename": "Face_Tracking.dll"
}
```

- `GUID` is the store GUID. Never change it.
- `DLLFilename` is the assembly ARCx loads, the csproj's assembly name.
- `Version` must be raised above the store's version before each upload.
- The presence of this file in the install folder is what marks a skill as installed.

The csproj

The generated project file is the template's with four values substituted. What matters:

Property	Meaning
`TargetFramework` = `net10.0`, SDK `Microsoft.NET.Sdk.Razor`	A Razor class library.
`Platforms`	Build `AnyCPU`. A robot skill is platform-agnostic managed code, and ARCx loads the same build on Windows, Linux and macOS.

`EnableDynamicLoading` = `true`	Emits the `.deps.json` ARCx resolves your dependencies and native libraries from.
`ResolveAssemblyWarnOrErrorOnTargetArchitectureMismatch` = `None`	`ARCx.dll` is architecture specific and your skill is AnyCPU by design. Do not "fix" MSB3270.
`ARCxGUID`	Your store GUID. Also the install folder name.
`ARCxAppFolder`	The folder holding `ARCx.dll`, `ARCx.xml` and `ARCx.exe`: the copy of ARCx that created the project. Debugging launches `ARCx.exe` from here.
`ARCxPluginFolder`	Where ARCx loads installed skills from, normally `C:\ProgramData\ARCx Data\Plugins`.
`GenerateDocumentationFile` = `True`	Ships `<Name>.xml` beside the DLL. ARCx reads it so your `///` comments describe the configuration to the AI agent.
`FrameworkReference Microsoft.AspNetCore.App`	Required: the Razor SDK does not add it, and ARCx's public API is Blazor typed.

The **References** item group names each ARCx library the skill compiles against: `ARCx`, `MudBlazor` and `DeepCloner` to start with, all `Private=false` so nothing is copied into your output. A comment beneath them lists the other libraries ARCx ships (OpenCvSharp, ONNX Runtime, NAudio, SkiaSharp, System.IO.Ports and more); copy a line out of the comment when you need one. A library ARCx does not ship is an ordinary NuGet `PackageReference`. `RobotSkill-Services-And-Core.md` section 28 has the rules, including never shipping your own ONNX Runtime.
To ship an image or other asset, embed it rather than putting it in `wwwroot`; `RobotSkill-UI-Components.md` section 17 shows how.

---

5. Step 2: Build and try it

Open the `.csproj` in Visual Studio and press F5, or build from the command line:

```
dotnet build "<Name>.csproj" -c Debug -p:Platform=AnyCPU
```

Name the `.csproj` on the command line: a solution file beside it spells the platform "Any CPU", and `dotnet build` given only the folder fails with MSB4126.
Every successful build **installs** the skill. The post-build step copies the output into the plugin folder:

```
C:\ProgramData\ARCx Data\Plugins\
  <StoreGUID>\
    plugin.json
    title.png              when the project has one
    AnyCPU\
      <DLLFilename>, its dependencies, <Name>.deps.json, <Name>.xml
```

Then in ARCx open **Robot Skills, Add**, find your skill, and add it to a project. F5 launches `ARCx.exe` from `ARCxAppFolder` with the debugger attached, so breakpoints in the worker and the Razor code-behind work.
A rebuild while ARCx has the skill loaded does not take effect. The DLL is in use, so the copy can fail (or appear to succeed) and removing and re-adding the skill keeps running

the old code. Stop ARCx, rebuild, and start it again. If the build itself cannot overwrite the file, the previous instance did not unload: something is still subscribed, see section 9.

---

6. Step 3: Write the skill

Work through the template file by file. Each step below says what goes where and which reference section has the detail.

The worker: where your code goes

Override	Put here	Detail
Constructor	Store the injected services. Nothing else; `Properties` is still null.	Services 1, 2
`Setup()`	Return the default window size and whether multiple instances and full screen are allowed.	Services 1
`Initialize()`	Create timers, executors and subscriptions, register video sources, kinematic chains or depth sensors. The skill is not in the project yet.	Services 1
`OnSetConfiguration()`	Read the saved configuration, or keep defaults when `IncludesConfig` is false.	Services 1, 19
`OnProjectLoadCompleted()`	Look up other robot skills, video sources, speech engines and chains by name. The only safe place to do so.	Services 1, 17
`GetConfigOnProjectSave()`	Write the configuration.	Services 19
`DisposeOverride()`	Unsubscribe from every event, stop and dispose everything `Initialize()` created, deregister what you registered.	Services 1, 28

Services you need are constructor parameters. The list of injectable services, and which are static, is in `RobotSkill-Services-And-Core.md` section 2.

The configuration

`Configuration.cs` is a plain class of public properties, each with a default and a `///` summary written for someone who has never seen the skill. Return `typeof(Configuration)` from `GetConfigurationType()` so the AI agent and other tools understand it, and mark properties that hold another skill's window title with `[RobotSkillReference("kind")]`. Do not rename properties once the skill is published; saved projects depend on them. See `RobotSkill-Services-And-Core.md` section 19.

The main window

`RobotSkillUIMain.razor` binds to the worker's fields and calls its methods. The worker pushes, the window never polls:

- Call `RefreshUI()` on the worker whenever what the user sees changes. It reaches every

browser and every Custom UI widget and is safe from any thread.
- Put what a loop updates several times a second in a `UCRefreshRegion` and redraw it with `RefreshUIRegion(name)`, so the rest of the window is not redrawn with it.
- Skip work that only feeds the display while `HasAnyUIClient` is false (or the narrower `HasUIClient` / `HasCustomUIClient`), and bring the display fields up to date in `UpdateMainUIStateRequested()` when a window is rendered again.

`RobotSkill-UI-Components.md` sections 1 to 3 cover the base class, layout and redrawing; sections 4 to 14 are the components a window is built from.

The configuration page

`RobotSkillUIConfig.razor` edits a deep clone of the configuration and copies it back to the worker in `OnSave()`, so Cancel really discards. Throw from `OnSave()` to reject a save with a message for the user. Lay settings out in `UCTitleComponentTableRowColumn` rows. See `RobotSkill-UI-Components.md` sections 1 and 4 to 7.

Control commands

Control commands are what scripts, other skills and AI agents call on your skill by window title. Declare them in `GetSupportedControlCommands()` with a description, named parameters and a `ReturnType`, and handle them in `ControlCommand()`. The template has working examples. See `RobotSkill-Services-And-Core.md` section 16.

Optional pieces

If the skill...	Do this	Detail
belongs on a phone or touch panel	Keep `RobotSkillUIWidget.razor` and `GetCustomUIWidgets()`; otherwise delete both.	UI 16
should give AI clients a typed tool or an image	Override `GetMcpTools()` / `GetMcpResources()`. Control commands reach them without this.	Services 16
is a camera	Register a source with `VideoRouteService`.	Services 8
speaks	Implement `TextToSpeechEngineInterface` on the worker.	Services 10
recognizes speech	Add a `SpeechToTextEngineAbstractEntity` subclass.	Services 11
drives the robot	Implement `MovementPanelInterface` on the worker.	Services 12
moves an arm or leg to a point	Register a chain with `KinematicsService`.	Services 13
is a depth camera or lidar	Register with `DepthSensorService` or publish to `NMSService`.	Services 14, 26
shares data with outside programs	Publish topics and services on the Robot Bus.	Services 27

offers sounds, camera actions or Auto Position actions to Blockly	Override `GetBlocklySoundItems()`, `GetBlocklyCameraItems()` or `GetBlocklyAutoPositionItems()`.	Services 1, 15
serves its own images or files	Override `OnDynamicResourceRequest()`.	UI 11
draws a map, meter or plot	Use `UCSkillCanvas`.	UI 10
runs a neural network	Use ARCx's shared ONNX Runtime and `OnnxAccelerationService`.	Services 23

Before moving on, read `RobotSkill-Services-And-Core.md` section 28, "Rules that will bite you".

---

7. Step 4: Test it

Run through these before you publish. Each one catches a mistake that does not show in a quick try.

1. **Add it to a new project.** It should work with the defaults, before its configuration has ever been opened.
2. **Save the project, close ARCx, reopen the project.** Every setting should come back.
3. **Open the configuration while the skill is busy.** Its work should pause, and Cancel should leave the configuration as it was.
4. **Open ARCx in two browsers at once.** Both windows should update.
5. **Add a second instance**, if `AllowMultipleInstances` is true. The two must not share state, logs or variable names.
6. **Remove the skill and add it again.** Nothing should keep running, and the log should show no errors.
7. **Put it on a Custom UI screen**, through its widget or the built-in Robot skill window widget, without opening the workspace. It should still update.
8. **Call each control command from a script**, with wrong parameters as well as right ones.
9. **Run it on Linux** if you can. ARCx supports Ubuntu 24.04 LTS on x64 and ARM64; a skill that calls Windows-only APIs or ships only Windows native libraries fails there.

---

8. Step 5: Publish it

1. **Add** **`title.png`** to the project folder: square, 256x256 or larger.
2. **Raise** **`Version`** in the project's `plugin.json` above the version in the store.
3. **Build Release**: `dotnet build "<Name>.csproj" -c Release -p:Platform=AnyCPU`, with ARCx stopped so the DLL can be replaced. The upload packages the whole install folder, so delete any `.pdb` a Debug build left in `Plugins\<guid>\AnyCPU\`.
4. In ARCx open **Robot Skills, Manage**. It lists the skills your account owns, matched against what is installed. Press **Upload** on yours.
5. Write the **release notes** (required). They are posted on the skill's store page as what changed in this version.
6. Tick **Send control command syntax** to publish your control commands as the Control Commands section of the skill's manual on synthiam.com. It replaces that section, so leave it unticked if you wrote it by hand. Reading the commands loads the skill the way a workspace does, so a skill that cannot load is stopped here instead of reaching the store.
7. On synthiam.com, use the skill's **Edit** link from **Robot Skills, Manage** to write its manual, add screenshots, and make it public when it is ready.

ARCx checks the package before it sends anything and says what is missing: a version not newer than the store's, no build output, a `DLLFilename` that is not in the build, no `title.png`, or a `plugin.json` without a name, description or 1 to 3 categories.
Users see the update in Add Robot Skill when the store version is newer than the one they have installed.

---

9. Troubleshooting

The name is refused with a message about "ARCx". A skill name cannot contain "ARCx" in any letter case. Choose another.
The skill does not appear in Add Robot Skill. Check `C:\ProgramData\ARCx Data\Plugins\<guid>\plugin.json` exists and that the `AnyCPU` folder holds the DLL named in `DLLFilename`.
"Unable to find RobotSkillWorkerAbstractEntity in assembly". The assembly has no public class deriving from the worker base, or a dependency failed to load. If the message says the type derives from "a different copy of it", the skill is shipping its own `ARCx.dll`: set the ARCx reference to `<Private>false</Private>` and rebuild. The ARCx log has the inner exception.
"Robot Skill is not supported by this platform". No folder matches the running platform and there is no `AnyCPU` folder.
The build succeeded but ARCx still runs the old code. ARCx had the DLL loaded. Stop ARCx, rebuild, start it again.
The DLL is locked and the build cannot overwrite it. A previous instance did not unload: something is still subscribed or a timer is still alive. Fix `DisposeOverride()`. Restarting ARCx releases it.
It works with one instance and fails with two. Each instance loads its own copy of every bundled DLL, and some libraries find themselves by name across the process. See `RobotSkill-Services-And-Core.md` section 1, "Loading and removal".
NullReferenceException in the constructor or Setup. `Properties` is not assigned yet. Move the code to `Initialize()`.
Another skill is not found in Initialize. It may not be loaded yet. Move the lookup to `OnProjectLoadCompleted()`.
Only one browser updates. You called Blazor's `StateHasChanged` instead of the base class one, or you are holding one component instead of calling `RefreshUI()` on the worker.
A Custom UI widget does not update, or nothing updates until the workspace is opened. You are calling `CurrentRobotSkillUIMainComponent?.StateHasChanged()`, which is null until a browser renders the skill's own window. Call `RefreshUI()`.
The window shows stale values when it is opened. The worker skips display work while `HasUIClient` is false. Set the display fields in `UpdateMainUIStateRequested()`.
The config page save is ignored. `OnSave()` must copy the edited clone back to the worker's live configuration.
The AI agent guesses at my settings. It sees no schema. Return the type from `GetConfigurationType()`, make sure `<Name>.xml` sits beside the DLL in the install folder, and give every property a summary.
The camera image never appears. The source name in the configuration does not match a registered source. Names are the camera skill's window title; default to the first source in `OnProjectLoadCompleted()`.
Upload says the robot skill could not be loaded. Reading the control commands loads the skill exactly as a workspace does. Usually a NuGet package or DLL the skill uses is not shipped next to it, or an ARCx reference has Copy Local set to true. The ARCx log has the detail.
MSB3270 architecture mismatch warning. Expected. The template disables it for a reason; do not change the platform to x64 to silence it.

Let AI Write Your Robot Skill

A step-by-step guide to making a robot skill for ARCx. No programming experience is needed: you describe your idea, ARCx sets up the project, and an AI coding assistant writes the code. Your job is to explain what you want, answer a few questions, and try the result on your robot.

Plan on about 30 minutes for the one-time setup, and a few minutes to create each robot skill after that.

Contents

1. What is a robot skill?
2. The big picture
3. What you need
4. Step 1: Install Visual Studio Code
5. Step 2: Install the .NET 10 SDK
6. Step 3: Add an AI assistant to Visual Studio Code
7. Step 4: Create the robot skill in ARCx
8. Step 5: Get the AI prompt ARCx wrote for you
9. Step 6: Open the project in Visual Studio Code
10. Step 7: Give the prompt to the AI
11. Step 8: Review the plan, then let it build
12. Step 9: Try your robot skill in ARCx
13. Step 10: Improve it
14. Step 11: Share it in the Robot Skill Store
15. Writing a good description
16. Troubleshooting
17. Quick checklist

---

1. What is a robot skill?

A robot skill is an add-on for ARCx that gives a robot something new to do.

When you open ARCx and press **Robot Skills → Add**, every item in that list is a robot skill: a camera viewer, a joystick, a speech recognizer, a servo driver. Each one appears as a small window on your workspace with its own controls, and each one has its own settings. Robot skills work together - a camera skill can feed a face tracking skill, which can move servos that another skill set up.

Every robot skill has the same four parts:

Part	What it is
The window	What you see on the workspace: buttons, readouts, a camera picture, a log
The settings	The configuration page you open with the skill's gear icon. Saved with your project
The behavior	The code that does the actual work: reading a sensor, moving a servo, talking to a service
Commands	Actions that scripts, other skills and the AI agent can trigger, such as `Start` or `Speak`

A few ideas of what you could build:

- **Connect hardware** - read a new sensor, or control a device ARCx does not support yet.
- **Add intelligence** - connect a vision or AI service so the robot can recognize or respond to things.

- **Make control easier** - turn a behavior you use often into a panel of buttons.

Start small. One useful idea done well is a great first robot skill.

---

2. The big picture

Here is the whole journey in one glance:

```
You describe your idea in ARCx
              |
              v
  ARCx creates a ready-to-build project  +  writes an AI prompt for it
              |
              v
  You open the project in Visual Studio Code and paste the prompt into the AI
              |
              v
  The AI proposes a plan, you approve it, the AI writes and builds the code
              |
              v
  The skill installs itself into ARCx - add it to a project and try it
              |
              v
  Happy with it? Upload it to the Robot Skill Store to share it
```

Why this works so well: an AI assistant on its own has never heard of ARCx, so it would guess at code that does not exist. ARCx solves that for you. The prompt it writes tells the AI exactly what ARCx is, where your project is, what every file is for, where the developer manuals and the full ARCx reference live on your computer, and the rules a robot skill must follow. The AI reads all of that before it writes a single line.

---

3. What you need

- **ARCx**, installed and running.
- **A Synthiam account**, signed in to ARCx. Creating a robot skill registers it under your account.
- **Visual Studio Code** - a free code editor. Step 1.
- **The .NET 10 SDK** - a free download that lets your computer build robot skills. Step 2.
- **One AI coding assistant** - we suggest either:

  Claude Code by Anthropic (needs a Claude account with a paid plan, or an Anthropic API account), or
  Codex by OpenAI (needs a ChatGPT account with a paid plan, or an OpenAI API key).

You only need one of the two. Both work very well for robot skills. If you already pay for Claude or ChatGPT, pick the one that matches.

Step 1: Install Visual Studio Code

Visual Studio Code (often called "VS Code") is where you will open your robot skill project and talk to the AI.

1. Go to **https://code.visualstudio.com/**.
2. Press the big **Download** button. The site picks the right version for your computer.
3. Run the installer.

   On Windows, accept the defaults. On the "Select Additional Tasks" page, it is handy to tick **Add "Open with Code" action to Windows Explorer directory context menu** - then you can right-click any folder and open it in VS Code.

4. Start Visual Studio Code once to make sure it opens.

Optional: In VS Code, open the Extensions panel (the four-squares icon on the left, or `Ctrl+Shift+X`), search for **C# Dev Kit** and install it. It colors the code and explains errors nicely. The AI does not need it, but it makes the code easier for you to read.

Step 2: Install the .NET 10 SDK

Robot skills are written in C#. The .NET SDK is the free Microsoft tool that turns that code into a robot skill ARCx can load. The AI uses it to build your skill and check its own work.

1. Go to **https://dotnet.microsoft.com/download/dotnet/10.0**.
2. Under **SDK**, download the installer for your computer (for most Windows PCs, **Windows x64**).
3. Run the installer and accept the defaults.
4. Check it worked: in VS Code, open a terminal with **Terminal → New Terminal**, type the line below and press Enter:

```
dotnet --version
```

You should see a number that starts with **10**. If the command is not found, close VS Code completely, open it again, and retry - it needs a restart to notice the new install.

Step 3: Add an AI assistant to Visual Studio Code

Pick **one** of the two options below.

Option A: Claude Code (Anthropic)

1. In VS Code, open the **Extensions** panel (`Ctrl+Shift+X`).
2. Search for **Claude Code**. Pick the one published by **Anthropic** and press **Install**.
3. When it finishes, a Claude icon appears in VS Code (in the sidebar, and at the top right of the editor when a file is open). Click it to open the Claude panel.
4. Sign in when asked. Choose your **Claude account** (Pro, Max, Team or Enterprise plan) or an **Anthropic Console** account if you pay for the API. A browser window opens - approve the sign-in and come back to VS Code.
5. You are ready. You will see a chat box where you can type to Claude.

Good to know:

- Claude asks permission before it edits files or runs commands. You can approve each one, or choose to allow edits for the rest of the session so it can work without stopping.
- If the installer mentions **Git for Windows**, install it from **https://git-scm.com/** and restart VS Code. Claude Code uses it on Windows.

Option B: Codex (OpenAI)

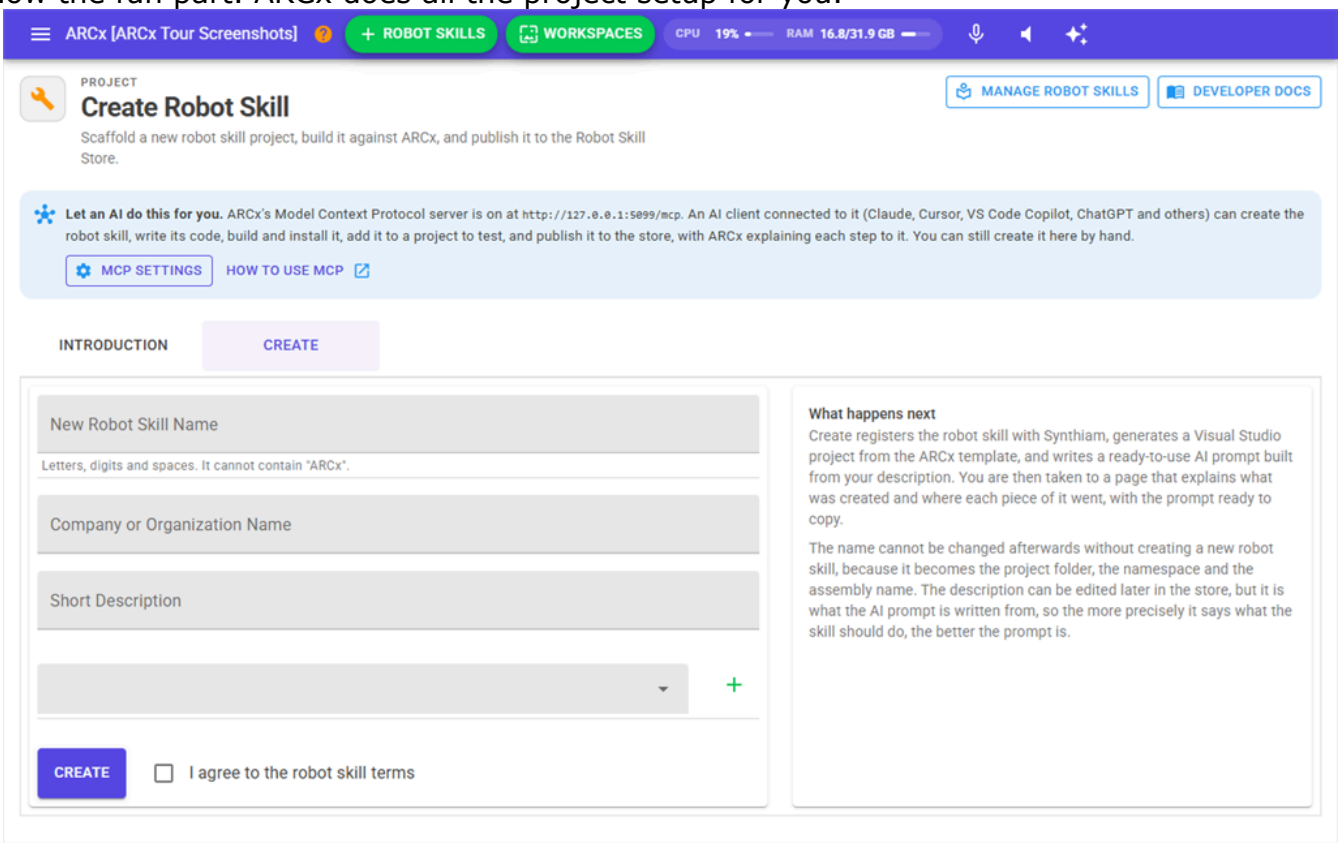
1. In VS Code, open the **Extensions** panel (`Ctrl+Shift+X`).
2. Search for **Codex**. Pick **Codex - OpenAI's coding agent**, published by **OpenAI**, and press **Install**.
3. When it finishes, a Codex icon appears in the VS Code sidebar. Click it to open the Codex panel. (If you do not see it, it may be tucked into the sidebar's overflow menu - drag it wherever you like.)
4. Press **Sign in with ChatGPT** and use your ChatGPT account (Plus, Pro, Business, Edu or Enterprise plan). A browser window opens - approve the sign-in and come back to VS Code. You can also use an OpenAI API key instead.
5. Under the chat box there is a mode setting. Choose **Agent**, so Codex can read and edit the project files and run the build. ("Chat" mode can only talk, not edit.)

Good to know:

- In Agent mode, Codex asks before doing anything outside the project folder. Approve the build command when it asks - that is how it checks its work.

Step 4: Create the robot skill in ARCx

Now the fun part. ARCx does all the project setup for you.



1. In ARCx, press the green **Robot Skills** button at the top, then choose **Create**.
2. The **Introduction** tab explains robot skills. Press **Create your robot skill** (or open the **Create** tab).
3. Fill in the form:

Field	What to type
New Robot Skill Name	A short, clear name, such as `Distance Alarm`. At least 5 characters, using letters, numbers and spaces. **Choose carefully - it cannot be changed later** without making a new skill
Company or Organization Name	Your name, club, school or company. At least 5 characters
Short Description	What the skill should do, in plain words. At least 30 characters, but more is better - **the AI builds from this**. See Writing a good description
Categories	Pick a category from the list and press **+**. Choose 1 to 3

4. Tick **I agree to the robot skill terms**.
5. Press **Create**.

If anything is missing, the box on the right lists what to fix. Correct it and press **Create** again.

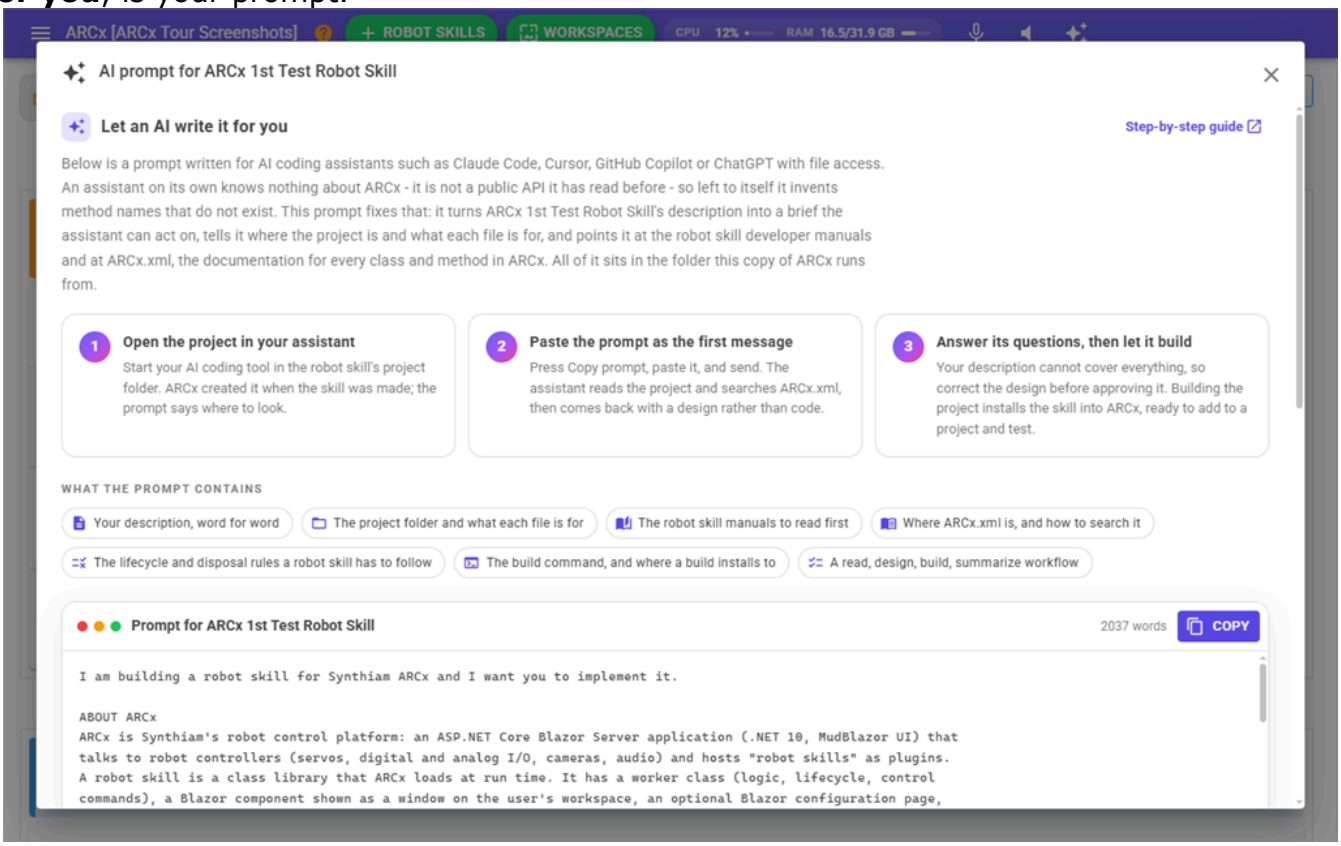
That one button press does six jobs for you:

- **Registers the skill with Synthiam** under your account and gives it a permanent ID. Nobody else can see it until you choose to share it.
- **Writes the skill's store details** (`plugin.json`) from what you typed.
- **Creates a complete, working project** from the ARCx template. It already builds and runs as a small demo (a timer and a counter) - the AI replaces that with your idea.
- **Connects the project to your copy of ARCx**, so it builds against the right version.
- **Sets up install-on-build**: every successful build installs the skill into ARCx automatically.
- **Writes you an AI prompt**, plus a guide file for the AI inside the project (more on that

below).
On Windows, ARCx also opens the new project folder in File Explorer. You will find your projects in **Documents\ARCx Data\Robot Skill Projects**, one folder per skill.

Step 5: Get the AI prompt ARCx wrote for you

After you press **Create**, ARCx takes you to a page that says **"[your skill] is ready to build"**. It shows what was created and where, and at the bottom, under **Let an AI write it for you**, is your prompt.



What the prompt is

It is a ready-made message for your AI assistant - a complete briefing, usually a couple of thousand words, that you never have to write yourself. It contains:

- **Your description**, word for word - what the skill should do.
- **The project** - what each file is for and which parts to change.
- **The robot skill manuals** - where they are on your computer, and a request to read them first.
- **ARCx.xml** - the full reference of every ARCx feature, and how to search it.
- **The rules** a robot skill must follow so it runs smoothly and shuts down cleanly.
- **The build command**, and where a build installs to.
- **A way of working**: read everything, propose a design, ask questions, build, then summarize.

You can scroll through the prompt on the page to see exactly what the AI will be told.

The buttons on that page

Button	What it does
Copy prompt	Copies the whole prompt, ready to paste. This is the one you need
Open project folder	Opens your skill's project folder in File Explorer (Windows)

Open manuals folder	Opens the folder holding the developer manuals, including this one
Write AGENTS.md / **Rewrite AGENTS.md**	Writes the AI guide file into the project folder (see below)
Go to Manage	Takes you to the page listing all your robot skills

Finding the prompt again later

You do not need to save the prompt anywhere. ARCx can show it again at any time:

1. Press **Robot Skills → Manage**.
2. Find your skill in the list.
3. Press the **sparkle** button (✨) in its row - the tooltip says *"Show the AI prompt for building this robot skill"*.

The same prompt, with the same buttons, opens in a window. It is rebuilt each time from the description in your project's `plugin.json`, so it always matches what the project says.

The AGENTS.md file - your AI's memory of ARCx

When ARCx created your project, it also put a file called **AGENTS.md** in the project folder. It holds the same background as the prompt: what ARCx is, what each file does, where the manuals and reference are, and the rules to follow.

AI assistants such as Claude Code and Codex read AGENTS.md **by themselves** whenever you open the project. That means you only paste the prompt **once**, for the first conversation. Any time you come back later - tomorrow or next year - just open the project and ask for what you want; the AI already knows about ARCx.

If you have an older project without AGENTS.md, open its prompt from **Manage** and press **Write AGENTS.md**.

Step 6: Open the project in Visual Studio Code

1. In VS Code, choose **File → Open Folder...**
2. Go to **Documents\ARCx Data\Robot Skill Projects** and select your skill's folder (it has the same name as your skill). Press **Select Folder**.

   Shortcut: press **Open project folder** on the prompt page, then right-click the folder and choose **Open with Code**.

3. If VS Code asks **"Do you trust the authors of the files in this folder?"**, choose **Yes, I trust the authors**. It is your own project, and the AI needs this to work.

You will see the project files on the left. You do not need to open or understand any of them - that is the AI's job - but here is what they are, just so they are not a mystery:

File	What it holds
`RobotSkillWorker.cs`	The behavior - the "brain" of the skill
`RobotSkillUIMain.razor`	The window on the workspace
`RobotSkillUIConfig.razor`	The settings page
`RobotSkillUIWidget.razor`	An optional small piece of the skill for your Custom UI screens
`Configuration.cs`	The list of settings that get saved with your project

`plugin.json`	The store details: name, description, categories, version
`AGENTS.md`	The AI's guide to ARCx

Step 7: Give the prompt to the AI

1. In ARCx, press **Copy prompt** (on the created page, or from the ✨ button on the Manage page).
2. In VS Code, open your AI panel:

   Claude Code: click the Claude icon.
   Codex: click the Codex icon, and check the mode is **Agent**.

3. Click in the chat box, paste the prompt (`Ctrl+V`), and press Enter.

That is it. The AI now reads your project, the manuals and the ARCx reference. This takes a minute or two. You will see it opening files and searching as it goes.

Step 8: Review the plan, then let it build

The prompt asks the AI to **show you a plan before writing any code**. It will come back with something like:

- what settings the skill will have,
- what the window will show,
- which commands it will offer to scripts,
- which parts of ARCx it will use,
- and questions about anything your description did not cover.

This is your moment to steer. Read the plan and reply in plain words, just like messaging a person:

Looks good. Make the alarm distance a setting, default 30 cm. Use the digital port the user picks, not a fixed one. Add a big red indicator when the alarm is on.

You do not need technical language. "Add a Stop button", "the number should be bigger", "use servo port 3 by default" - all perfectly fine.

When the plan is right, tell it to go ahead:

That's the design. Please build it.

The AI then writes the code and builds it. If the build shows errors, the AI reads them and fixes them on its own - you just approve its actions when it asks. When it is done it gives you a summary of what it built, what each setting and command does, and what to test on your real robot.

About the thumbnail warning: every build says *"This robot skill has no thumbnail"*. That is expected. The thumbnail is the picture shown in the store, and it is yours to add later (see Step 11). The prompt already tells the AI to ignore it.

Step 9: Try your robot skill in ARCx

Every successful build installs the skill into ARCx automatically - nothing to copy, nothing to upload.

1. In ARCx, open a project (or start a new one) and connect your robot if the skill uses hardware.
2. Press **Robot Skills → Add**.
3. Find your skill in the category you chose, and add it to the workspace.
4. Try it: press its buttons, open its settings with the gear icon, and check it does what you asked.

If you rebuild while ARCx is running: once a robot skill has been used, ARCx keeps its

files open, so a new build may not be able to replace them. If the AI reports that the build could not copy or overwrite files, close ARCx, ask the AI to build again, then reopen ARCx.

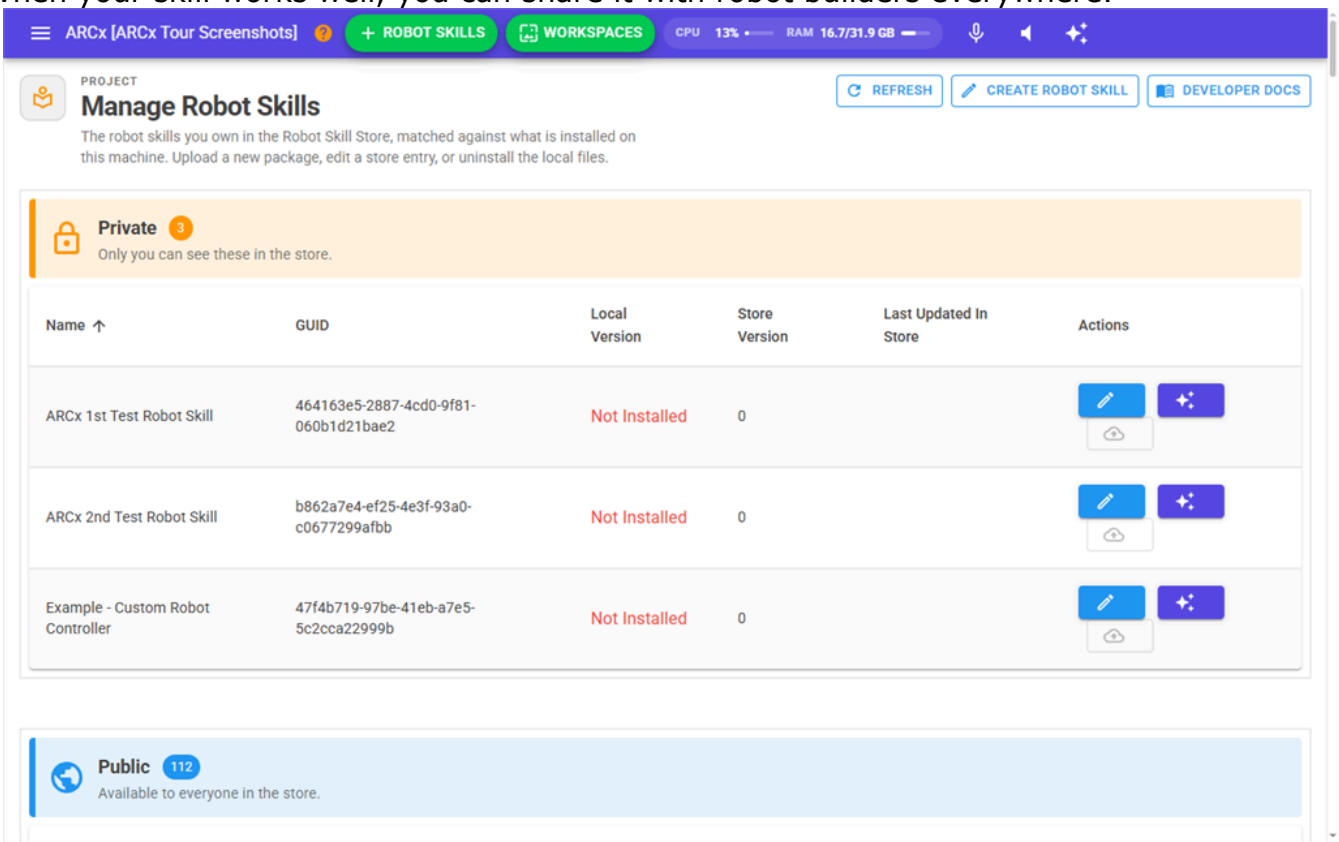
Step 10: Improve it

Found something to change? Just tell the AI in the same VS Code chat:
When I press Start nothing shows in the log. Can you log each reading?
Add a setting for how often it checks, in milliseconds.
The window is too cramped, put the buttons on one row.
The AI changes the code and rebuilds. Close and reopen ARCx if needed (see the note above), and try again. Repeat until you love it.
Coming back another day? Open the project folder in VS Code and start a new chat. **No need to paste the prompt again** - the AI reads AGENTS.md on its own. Just say what you want next.
A few tips:

- **Describe what you see**, not what you think the code should be. "The number stays at zero even when I move my hand" is perfect.
- **Copy error messages** from ARCx's log straight into the chat. The AI loves details.
- **Change one thing at a time**, and test in between. Small steps are easier to check.
- **Test with the real hardware.** The AI cannot see or touch your robot, so it relies on you to say what really happened.

Step 11: Share it in the Robot Skill Store

When your skill works well, you can share it with robot builders everywhere.



1. **Add a thumbnail.** Make a square picture, **256 × 256 pixels or larger**, save it as a PNG named `title.png`, and put it in the project folder. This is what people see in the store.
2. **Raise the version.** Every upload needs a version number higher than the one already in the store. Ask the AI: "raise the version in plugin.json for a new release and build". (Or change `Version` in `plugin.json` yourself and build once more.)
3. In ARCx, press **Robot Skills → Manage**, press **Refresh** so it sees your latest build,

and find your skill.
4. Press the green **upload** button in its row. Read the summary of what will be sent and press **Upload**. (If the upload button is greyed out, hover over it - it tells you exactly what is missing, such as the thumbnail or a higher version number.)
5. Press the **edit** button (the pencil) in the same row to open your skill's store page on synthiam.com. Add setup instructions, and set it to **public** when you are ready for others to find it.

Writing a good description

The description is the single most important thing you type. The AI prompt is built from it, so a clear description means a better first version and fewer questions.
A good description says:

- **What it does** - the main job, in one sentence.
- **What the user sees** - buttons, numbers, pictures in the window.
- **What the user can set** - the options on the settings page, with sensible defaults.
- **What hardware or services it uses** - which sensor, port, camera or website.
- **What other skills or scripts can do with it** - commands such as Start, Stop, Get Value.

Too short:
Reads a distance sensor.
Great:
Watches an ultrasonic distance sensor on a port the user picks and sounds an alarm when something comes closer than a set distance (default 30 cm). The window shows the current distance in large numbers, a red/green status light, and Start and Stop buttons. Settings: the sensor port, the alarm distance, how often to check, and a script to run when the alarm triggers. Scripts can start and stop it and read the latest distance.
Do not worry about getting it perfect. The AI will ask about anything unclear, and you can always add more detail in the chat.

Troubleshooting

Problem	What to do
The **Create** button shows errors	Read the list on the right. Common ones: name under 5 characters, description under 30 characters, no category, terms box not ticked
"The destination folder has a plugin.json"	A project with that name already exists. Pick a different name, or move the old folder out of **Documents\ARCx Data\Robot Skill Projects**
`dotnet` is not recognized	Install the .NET 10 SDK (Step 2), then close and reopen VS Code
The AI says it cannot edit files	Codex: switch the mode to **Agent**. Claude Code: approve the edit when it asks. Both: make sure you trusted the folder when VS Code asked
The AI invents features that do not exist	Remind it: "Check the manuals and ARCx.xml before using anything, and ask me if something is not there." The prompt and AGENTS.md already say this, but a nudge helps

The build cannot overwrite files	ARCx is holding the old version. Close ARCx, build again, reopen ARCx
The skill is not in **Robot Skills → Add**	Make sure the last build succeeded (the AI will say so). Look under the category you chose. Restart ARCx if it was running during the build
"This robot skill has no thumbnail"	Expected until you add `title.png`. Only matters when publishing
The upload button is greyed out	Hover over it to see why. Usually the thumbnail is missing, the version in `plugin.json` is not higher than the store's, or the skill has not been built yet. Fix it, build, then press **Refresh** on the Manage page
I lost the prompt	**Robot Skills → Manage**, then the ✨ button on your skill's row
The project was moved and the prompt looks wrong	Open the prompt from **Manage** and press **Rewrite AGENTS.md** to refresh the guide file

---

Quick checklist

One-time setup

- [ ] Visual Studio Code installed
- [ ] .NET 10 SDK installed (`dotnet --version` shows 10)
- [ ] Claude Code **or** Codex installed in VS Code and signed in
- [ ] Signed in to your Synthiam account in ARCx

For each new robot skill

- [ ] **Robot Skills → Create**: name, company, a detailed description, categories, agree, **Create**
- [ ] **Copy prompt** on the page that follows
- [ ] Open the project folder in VS Code and trust it
- [ ] Paste the prompt into Claude Code or Codex (Agent mode)
- [ ] Read the plan, answer its questions, say "build it"
- [ ] **Robot Skills → Add** in ARCx and try it
- [ ] Ask the AI for changes until it is just right
- [ ] Add `title.png`, then **Robot Skills → Manage → Upload** to share it

That is all there is to it. Describe your idea, let ARCx and the AI do the heavy lifting, and give your robot a new skill.

Want to go deeper? The other manuals in this folder - `RobotSkill-UI-Components.md` and `RobotSkill-Services-And-Core.md` - are the technical reference the AI reads. They are written for developers, but they are a great way to see everything a robot skill can do.

Robot Skill UI Components

Every user interface component ARCx provides to a robot skill, what it is for, and the parameters it takes.

ARCx is an ASP.NET Core **Blazor Server** application using **MudBlazor**. A robot skill's UI is one or more Razor components compiled into the skill's own assembly and rendered by ARCx inside
the workspace. Every MudBlazor component is available to a robot skill, plus the ARCx `UC*`

components listed here, which exist because they do something MudBlazor cannot: pick an EZB
servo port, show a live camera frame, edit a script, draw a lidar map.
This manual is a companion to `RobotSkill-Services-And-Core.md`, which covers the non-visual
API (EZB hardware, video, audio, kinematics, scripting). Both sit beside `ARCx.exe`, `ARCx.dll` and `ARCx.xml`. `ARCx.xml` remains the authoritative per-member reference; this manual is the map that tells you what to look up.

Contents

1. The three component base classes
2. Layout rules
3. Redrawing the UI
4. Settings row components
5. Hardware port pickers
6. Servo configuration
7. Script editing
8. Logging and status output
9. Video
10. Drawing: UCSkillCanvas
11. Serving your own images: UCDynamicImage
12. Movement and input
13. Kinematics
14. Page furniture and misc
15. Popups: snackbars, dialogs, navigation
16. Custom UI widgets
17. MudBlazor conventions ARCx expects

---

1. The three component base classes

Namespace `ARCx.Components.InterfaceEntities`.
A robot skill has up to three Razor components, each deriving from one of these.

Base class	File in the template	Where it is shown
`RobotSkillUIMainBaseEntity`	`RobotSkillUIMain.razor`	The skill's window on the workspace
`RobotSkillUIConfigBaseEntity`	`RobotSkillUIConfig.razor`	The skill's configuration page
`RobotSkillUIWidgetBaseEntity`	`RobotSkillUIWidget.razor`	A widget on a user's Custom UI screen (optional)

Declare the base with `@inherits` at the top of the `.razor` file:

```
@inherits ARCx.Components.InterfaceEntities.RobotSkillUIMainBaseEntity
```

RobotSkillUIMainBaseEntity

The window on the workspace desktop.

Member	Notes
`CurrentRobotSkillWorker`	`[Parameter]`. The worker this UI belongs to. **Null on the prototype instance** ARCx creates at skill load and never renders, so null-guard anything that may run before a

	browser rendered the skill.
`AnyActiveClients`	True when not disposed and at least one browser is connected. Check it before doing expensive display work.
`IsDisposed`	True once `Dispose()` has run. Refreshes are ignored afterwards.
`StateHasChanged()`	**Hides** the Blazor one. Throttled, thread safe, fans out to every connected browser, does nothing when nobody is watching. Use this one.
`StateHasChangedForced()`	Same but bypasses the throttle timer and the rate limit. Not recommended.
`ShouldRender()`	Enforces the `RobotSkillUIQueuedUpdateMS` rate limit on every render, including Blazor's own after a UI event. If you override it, combine your condition with `base.ShouldRender()`.
`OnDispose()`	`virtual`. Raised when the browser navigates away or the skill is removed. Unsubscribe here. Exceptions are only logged.
`OnResizeEnd()`	`virtual`. Raised after the user finishes dragging the window. New size is in `CurrentRobotSkillWorker.Properties.Width` / `.Height`.

One instance exists **per connected browser**. State that every browser must agree on belongs on
the worker, not in a field on this component.
Cast the worker once, as the template does:

```
RobotSkillWorker _currentRobotSkillWorker => base.CurrentRobotSkillWorker as
RobotSkillWorker;
```

RobotSkillUIConfigBaseEntity

The configuration page. ARCx supplies the SAVE and CANCEL buttons.

Member	Notes
`CurrentRobotSkillWorker`	Cascading value, `private set`. Null on the prototype instance.
`OnSave()`	`virtual`. Raised on SAVE, before the worker's `OnConfigWindowClosed(ConfigChanged = true)`. **Throw to reject the save** - the message is shown to the user, the page stays open, the worker is not notified. This is where validation goes.
`OnCancel()`	`virtual`. Raised on CANCEL. Revert anything you changed on the worker.

	Throwing also keeps the page open.
`StateHasChanged()`	Thread safe, marshalled through `InvokeAsync`. Does nothing once disposed or when no browser is connected.
`OnDispose()`	`virtual`. Raised when the user leaves the page or the skill is removed.

The convention the template follows is to edit a **deep clone** of the configuration and copy it back to the worker in `OnSave()`, so CANCEL genuinely discards.
While any configuration window is open, `RobotSkillWorkerAbstractEntity.AllWorkersPaused` is true for **every** skill in the project. Worker timers should check it and skip their work.

RobotSkillUIWidgetBaseEntity

An optional compact piece of the skill the user can drop on their own Custom UI screen. If this is not provided, the RobotSkillUIMainBaseEntity will be used on Custom UI screens. See section 16.

Member	Notes
`CurrentRobotSkillWorker`	`[Parameter]`. The same worker instance the skill window uses.
`WidgetKey`	`[Parameter]`. The `Key` from the worker's `GetCustomUIWidgets()` entry, for a component that serves several definitions.
`Settings`	`[Parameter]`. `Dictionary<string, string>` the user filled in, seeded from `DefaultSettings`.
`IsEditMode`	`[Parameter]`. True while shown in the designer. **Do not move the robot, run scripts or take any action** while true.
`Width` / `Height`	`[Parameter]`. The box in design pixels, default 200 x 120.
`GetSetting(key, defaultValue = "")`	A setting, or the default when missing or blank.
`GetSettingNumber(key, double defaultValue)`	As a number, or the default.
`GetSettingBool(key, bool defaultValue)`	Understands `true`, `1`, `on`, `yes`.
`AnyActiveClients`	True when not disposed and a browser is connected.
`StateHasChanged()`	Throttled by the `RobotSkillUIQueuedUpdateMS` preference like the skill window, thread safe.
`OnDispose()`	`virtual`. Raised when the widget leaves a screen or the browser leaves the page.

2. Layout rules

The skill window body is a **flex column**. A direct child given `flex: 1 1 0` takes whatever height the other children leave over, so the window can be dragged to any size with no gap at
the bottom and nothing spilling out.
ARCx ships three helper classes in `arcx.css` so every robot skill lays out the same way:

Class	Use it for
`.arcx-skill-fill`	Fill what is left: a table, a list, a control area. `flex: 1 1 0; min-height: 0; min-width: 0;`
`.arcx-skill-log`	A `UCLogBuffered` wrapper. Grows with the window, never under 60px.
`.arcx-skill-media`	A camera image or canvas. Fills what is left, clips, never grows the window. `min-height: 80px; overflow: hidden;`

Rules:

- **Never give the log, a camera view or a canvas a fixed pixel height.** They already fill their
container; a fixed height is exactly what stops them scaling.
- A `<style>` block in a `.razor` file is **global to the page**, and every robot skill on a workspace shares that page. Prefix every class name with something unique to your skill (its
initials) so your rules cannot collide with another skill's.
- `.arcx-skill-body` sets `overflow: hidden` on the window, so anything that overflows is clipped
rather than escaping.

The template's main window is the canonical shape:

```
<style>
  .tpl-main {
    flex: 1 1 0;
    min-height: 0;
    min-width: 0;
    overflow: auto;
    display: flex;
    flex-direction: column;
    gap: 8px;
    padding: 4px;
  }
  .tpl-controls { flex: 0 0 auto; display: flex; flex-direction: column; gap: 8px; }
  .tpl-row { display: flex; flex-wrap: wrap; align-items: center; gap: 6px; }
</style>

<div class="tpl-main">

  <div class="tpl-controls">
    @* buttons, readouts, sliders - only as tall as their content *@
  </div>

  <div class="arcx-skill-log">
    <UCLogBuffered InstanceGUID="@(_currentRobotSkillWorker.Properties.InstanceGuid)"
Lines="3" />
  </div>
</div>
```

The default window size comes from the worker's `Setup()` override (`RobotSkillWorkerDefaultParams.Width` / `.Height`, and `AllowFullScreen`).

3. Redrawing the UI

There are four ways to redraw, and they are not interchangeable.

Call	From	Reaches
`worker.RefreshUI()`	The worker	Every browser showing the skill window, **and** every Custom UI widget the skill provides
`worker.RefreshCustomUIWidgets()`	The worker	Only the Custom UI widgets
`worker.RefreshUIRegion(name)`	The worker	Only the `UCRefreshRegion`s with that name, on every browser - see below
`this.StateHasChanged()`	Inside a UI component	That component, on every connected browser

`RefreshUI()` is the one a worker calls after it changes something the user can see. It is safe from any thread, safe before anything is rendered (headless / auto-start), and does nothing when
no browser is connected.
Renders are **throttled** by a timer whose interval is the `RobotSkillUIQueuedUpdateMS` preference (Advanced tab, default 250ms), so many calls in quick succession produce one render.
That is why you can call it freely from a timer.
The same interval is a hard rate limit on the component itself. The base classes' `ShouldRender()`
holds any render that comes sooner than the interval after the last one - including the render
Blazor does on its own after every UI event (dragging a slider, ticking a checkbox) and when the
parent passes new parameters - and draws it once the interval has passed.
`StateHasChangedForced()`
is the only way past it. If you override `ShouldRender()`, combine your condition with `base.ShouldRender()` or your skill loses the limit.
Two components bypass Blazor rendering entirely for efficiency, and have their own refresh path:

- `UCLogBuffered` sets the text with JavaScript. Call `UCLogBuffered.AddEntry(...)`.
- `UCDynamicImage` sets the `img src` with JavaScript. Call `RefreshImage()`.

What a redraw actually costs

A redraw re-renders **the whole component**, not the part that changed. Blazor then diffs the
result against what the browser is showing. That is cheap for a window of a dozen controls and
expensive for a window with a long list in it, because the cost is per control per redraw.
Two MudBlazor controls cost far more than they look:

- **`MudTooltip` is a popover.** It registers itself with `MudPopoverProvider`, which is a single component in the ARCx page layout shared by every robot skill on the workspace. Each tooltip is
  an entry in a keyed list that the whole page re-renders.
- **`MudFileUpload` carries its own `<InputFile>`**, which captures an element reference back
  into the render tree. So does anything using `@ref` on an element.

Put either of those inside a row of a list and multiply by the row count. A hundred-row table with three tooltips and an upload button per row is several hundred popovers and a hundred file
inputs, all torn down and rebuilt on every redraw. Drive that from a timer and Blazor's render
tree can be corrupted outright - the symptom is an `InvalidCastException` from deep inside
`Microsoft.AspNetCore.Components` naming `ElementReference` or `ComponentState`, followed by the
circuit dropping and the browser losing its connection.

Redraw only the part that changes: UCRefreshRegion

Most windows that redraw often have one small thing that changes - a play position, a counter, a
sensor reading, a status line - sitting in a window that otherwise stays put. Calling `RefreshUI()`
to move that one number redraws every row and every button around it, several times a second.
Wrap the part that changes in a `UCRefreshRegion` and have the worker redraw just that:

```
<UCRefreshRegion Worker="Worker" Name="@RobotSkillWorker.PositionRegion">
  <MudText Typo="Typo.caption">@(Worker.IsPlaying ? $"Playing - {Worker.PlaySeconds:0.0}s" :
"Stopped")</MudText>
</UCRefreshRegion>

@* ...the long table that does not change while a sound plays... *@
```

```
// On the worker
internal const string PositionRegion = "position";

void playbackLoop() {

  while (playing) {

    PlaySeconds = ...;

    if (HasUIClient)
      RefreshUIRegion(PositionRegion);   // the line above, not the window

    Thread.Sleep(250);
  }

  RefreshUI();   // playback ended - buttons, labels and the rest do change now
}
```

Parameter	Notes
`Worker`	Required. The worker whose `RefreshUIRegion()` redraws it.
`Name`	Required. Several regions can share a name - the status line and a dialog's play marker, say - and are redrawn together. Keep the name in a `const` on the worker so the two sides cannot drift.
`ChildContent`	The markup to redraw.

- **This is the default for new robot skills.** Anything a loop or timer updates while the rest of
the window stays the same goes in a region. Keep `RefreshUI()` for the moments the rest of the

window really does change: something started or stopped, an item was added or removed.
• The region is throttled by `RobotSkillUIQueuedUpdateMS` like the window, reaches every connected
browser, and does nothing when no browser is connected. It is also redrawn whenever the window
around it is, so it is never out of date after a `RefreshUI()`.
• **Read what changes from members, not from a local.** The wrapped markup reads fields and
properties when the region draws, so `Worker.PlaySeconds` is always current. A local from the
window's `@{ var tracks = Worker.Tracks; }` block is the value it had when the *window* last
drew, and a region redraw will not update it.
• A region draws no element of its own, so it can sit anywhere: inside a flex row, inside a table
cell, or inside an `<svg>` around a single `<line>` (the Sound Board's play marker does this, so
moving the marker does not redraw the waveform).
• Keep regions small. A region wrapping half the window saves half the cost; a region wrapping one
label saves nearly all of it.

Rules for a window that redraws often

• **Put the part that changes in a `UCRefreshRegion`** and redraw it with `RefreshUIRegion()`
instead of `RefreshUI()`. See above.

• **Redraw no faster than the user can read.** A play position, a counter or a sensor reading does
not need more than four frames a second. The `RobotSkillUIQueuedUpdateMS` throttle sets a floor,
not a target - do not ask for a redraw every 50ms and rely on it.

• **Guard the work with `HasAnyUIClient`** (or the narrower `HasUIClient` / `HasCustomUIClient`) in
any loop or timer that only exists to feed the display, so the skill costs nothing when nobody
has the window open. See below.

• **No `MudTooltip` inside a repeated row.** Use the plain HTML `title` attribute on the control
instead - it is a native browser tooltip, costs nothing and reads the same to the user. Keep `MudTooltip` for the handful of controls in a toolbar or a settings row.

• **One `MudFileUpload` per window, not per row.** Put a single picker outside the list, remember
which row was pressed, and open it with `OpenFilePickerAsync()`:

```
@* One picker for the whole table. It draws nothing itself - the row buttons
are the visible part. *@
<MudFileUpload T="IBrowserFile" @ref="_picker" FilesChanged="filePicked"
Accept=".wav,.mp3">
  <CustomContent Context="p"><span></span></CustomContent>
</MudFileUpload>
```

```
@foreach (var row in rows) {
  <MudIconButton Icon="@Icons.Material.Filled.FileOpen"
                 title="Load a file into this slot"
                 OnClick="@(() => pickFor(row))" />
}
MudFileUpload<IBrowserFile> _picker;
MyRow                       _target;

async Task pickFor(MyRow row) {

  _target = row;

  await _picker.OpenFilePickerAsync();
}

async Task filePicked(IBrowserFile file) {

  var target = _target;

  _target = null;

  if (target == null || file == null)
    return;

  // ...
}
```

• Anything genuinely high frequency - a waveform, a map, a live plot - belongs on `UCSkillCanvas` or `UCDynamicImage`, which do not re-render at all. See sections 10 and 11.

Skip work nobody will see

`RefreshUI()` already does nothing when nothing is rendered, but the work a worker does *before* calling it - formatting a table, building a preview image, averaging a buffer for a meter - still runs. On a single board computer that is CPU spent for no one. The worker can ask whether anything of the skill is actually on a screen and skip that work:

Worker member	True when
`HasUIClient`	At least one browser is showing this skill's window, on any workspace. Covers the `UCRefreshRegion`s inside it
`HasCustomUIClient`	At least one browser is showing a Custom UI screen that holds one of this skill's widgets
`HasAnyUIClient`	Either of the above. The check that matches `RefreshUI()`, which redraws both
`AnyActiveClients`	Any browser is connected at all - even one on a different workspace. Broader than the three above

```
void sensorTimer_Elapsed(object sender, EventArgs e) {

  // Reading the sensor and acting on it is the skill's job - always do it
```

```
  var reading = readSensor();

  actOnReading(reading);

  // Building the display text is only for the user - skip it when nobody is looking
  if (!HasAnyUIClient)
    return;

  DisplayText = formatReading(reading);

  RefreshUI();
}
```

• **Only skip display work.** Anything the robot depends on - reading a sensor, driving a servo,
setting a script variable, publishing to another skill - runs whether or not a browser is looking.
• **Catch up when the window comes back.** When a browser renders the skill's window, it calls the
worker's `UpdateMainUIStateRequested()`; set the display fields to their latest values there, so
the window does not show stale data until the next tick. Widgets have no such call and pick up the
worker's next refresh.
• The flags change as browsers switch workspaces, open and close Custom UI screens, and disconnect.
They are cheap (a lock and a count) - check them every tick rather than caching them.

Never render straight from a collection the worker mutates

A worker's lists are changed from whatever thread a ControlCommand, a script or another robot
skill arrives on. The browser renders on its own thread. Indexing a live `List<T>` while drawing
means a render can be handed an item a script removed a microsecond earlier, and **an exception
thrown part way through a render leaves the render tree half built, which Blazor cannot recover
from**. It is not the same as an exception in an event handler.
Expose a snapshot from the worker and take it once per render:

```
// On the worker
readonly object _itemsLock = new();

internal MyItem[] Items {
  get {
    lock (_itemsLock)
      return _items.ToArray();
  }
}
```

```
@* One snapshot for the whole of this render *@
@{
  var items = Worker.Items;
}

@for (int i = 0; i < items.Length; i++) {
  ...
}
```

Every method on the worker that adds, removes or replaces an item takes the same lock. Call
`RefreshUI()` **after** releasing it.

@key on a list row

Give a repeated row a `@key` so Blazor reuses the row instead of rebuilding it, and make sure the
key is:

- **Never null or blank.** Blazor silently ignores a null key, which leaves keyed and unkeyed rows
  mixed in one list and produces a bad diff. Repair missing keys in `OnSetConfiguration()` when
  loading a project saved by an older build of the skill.
- **Unique among its siblings.** Two rows with the same key is the same problem.
- **Stable for the life of the row.** Key on the slot or the record, not on its contents. When the
  user replaces what is in a slot, keep the slot's key so the row survives the change.

A render exception no longer takes the page down

ARCx wraps each robot skill in its own error boundary, so a skill that throws while rendering shows the error in its own window with a **Try Again** button, and the rest of the workspace keeps
running. That is a safety net, not a licence: the window stops drawing until the user presses the
button, and the entry lands in the debug log with the skill's name on it. Handle your own errors.

---

4. Settings row components

The standard way to lay out a configuration page: a title with a help icon on the left, the editor on the right.

UCTitleComponentTableRowColumn

Renders a `<tr>`, so it must sit inside a `<table>` / `<tbody>`. **This is the row robot skill configuration pages use.**

Parameter	Type	Default	Notes
`ChildContent`	`RenderFragment`	-	The editor. A cascading `"Disabled"` value is provided to it, so ARCx input components inside follow `Disabled`.
`Title`	`string`	`"Value"`	The label in the title cell
`HelpText`	`string`	`"Enter a value for this parameter"`	Tooltip on the help icon
`Href`	`string`	empty	When set, the help icon becomes a link opening in a new tab
`Disabled`	`bool`	`false`	Blurs the row and cascades `Disabled`
`ContentWidth`	`string`	empty	CSS width for the value cell, i.e. `"300px"`

`TitleWidth`	`string`	empty	CSS width for the title cell

```
<table>

  <UCTitleComponentTableRowColumn Title="Poll interval (ms):"
                                  HelpText="How often the sensor is read. Lower uses more
CPU.">
    <MudNumericField T="int" Min="50" Max="10000" @bind-Value="@(_configuration.PollMS)" />
  </UCTitleComponentTableRowColumn>

  <UCTitleComponentTableRowColumn Title="Enabled at start:"
                                  HelpText="Start polling as soon as the project loads.">
    <MudCheckBox T="bool" @bind-Value="@(_configuration.AutoStart)" />
  </UCTitleComponentTableRowColumn>

</table>
```

UCTitleComponentRow

The same idea without a table. Use it in a main window or anywhere a `<table>` would be awkward.

Parameter	Type	Default	Notes
`ChildContent`	`RenderFragment`	-	The editor
`Title`	`string`	`"Value"`	
`HelpText`	`string`	`"Enter a value for this parameter"`	
`Vertical`	`bool`	`false`	Title above the editor instead of beside it
`Href`	`string`	empty	Help icon becomes a link
`Disabled`	`bool`	`false`	Blurs the row, cascades `Disabled`

---

5. Hardware port pickers

Namespace `ARCx.Components.UserComponents`. Descriptor types are in `ARCx.Config.Helpers`.

Each picker edits a **port descriptor** stored in the skill's configuration. A descriptor holds the EZB board index and the port. The dialog also lets the user name the port; the name goes
into the project's port description dictionaries and is then shown by every other picker in the project, and on the Robot Configuration picture.

All of these edit `Value` in place and never replace the instance. `ValueChanged` exists only so `@bind-Value` compiles; it is not raised. Pass the instance that lives in your configuration.

Component	Descriptor	Companion `*AndLabel`
`UCPortEditDigitalButton`	`DigitalPortDescriptor`	`UCPortEditDigitalButtonAndLabel`
`UCPortEditADCButton`	`ADCPortDescriptor`	`UCPortEditADCButtonAndLabel`
`UCPortEditPWMButton`	`PWMPortDescriptor`	`UCPortEditPWMButtonAndLa`

		`bel`

The bare button shows the selected port and its project description, and opens the picker dialog.
The `*AndLabel` variant puts a help icon and a title label in front of it as a single row.

Parameter	Type	Notes
`Value`	the descriptor	**Must not be null.** Edited in place
`Disabled`	`bool`	Also disabled while the dialog is open
`Title`	`string`	`*AndLabel` only. Defaults: `"Digital Port"`, `"ADC Port"`, and **`"Digital Port"` for the PWM one** - set it to `"PWM Port"` explicitly
`HelpText`	`string`	`*AndLabel` only

```
<UCPortEditDigitalButtonAndLabel Title="Trigger port"
                                 HelpText="The digital port wired to the sensor's trigger
pin."
                                 Value="@(_configuration.TriggerPort)" />
```

Reading the port at run time:

```
var ezb = EZBManager.TryGetEZB(_Configuration.TriggerPort.BoardIndex);

if (ezb == null || !ezb.IsConnected)
  return;

var state = ezb.Digital.GetDigitalPort(_Configuration.TriggerPort.Port);
```

`ToString()` on any descriptor gives `"board.PORT"`, i.e. `"0.D3"` or `"0.ADC0"`.

---

6. Servo configuration

UCServoSelector

The standard panel for the servo or servos a skill controls: port, min/max position, velocity, acceleration and invert for the first servo, plus a "multiple servos" button opening a dialog to
add more servos that follow the first.
Dragging the min or max slider **physically moves the servo** to that position so the user can
see the limit. The single-servo controls are disabled when the collection holds more than one
servo; those are edited in the multiple-servo dialog.

Parameter	Type	Default	Notes
`Value`	`ServoCollection`	-	**Must not be null and must contain at least one `ServoDescriptor`.** `Servos[0]` is the one the panel edits.

			Modified in place
`Title`	`string`	`"Servo Details"`	Heading
`HelpText`	`string`	`"Select the servo parameters. Click for more information."`	
`Disabled`	`bool`	`false`	Blurs and disables every control
`ValueChanged`	`EventCallback<ServoCollection>`	-	For `@bind-Value` compatibility. Not raised

```
<UCServoSelector @bind-Value="@(_configuration.PanServo)"
                 Title="Pan servo"
                 HelpText="The servo that turns the head left and right." />
```

`ServoCollection` defaults to a single unassigned (`NA`) servo, so a new configuration already has a row to edit. Moving it:

```
// Applies each descriptor's own min, max, invert, multiplier, velocity and acceleration
EZBManager.SetServoPosition(_Configuration.PanServo, 90);

// Map an arbitrary input range onto each servo's own min..max
EZBManager.SetServoPositionScalar(_Configuration.PanServo, 0, 100, 50);
```

Supporting components in `ARCx.Components.UserComponents.UCServoSelectorComponents` (`UCMultipleServoSelector`, `UCPortEditServo`) are used by `UCServoSelector` itself; a skill normally uses the selector rather than these directly. The template's `_Imports.razor` already
has a `@using` for that namespace.

---

7. Script editing

UCScriptEditButton

A button that opens the full screen script editor dialog - Blockly, JavaScript and Python tabs -
for a `ScriptDefinition` (`ARCx.Config.Helpers`). Use one for every script the skill exposes. The
button label shows the script's current type, or "Empty". The dialog provides Save, Cancel (reverts to the script as it was when the dialog opened) and Panic.

Parameter	Type	Default	Notes
`Value`	`ScriptDefinition`	-	**Must not be null.** Edited in place, so pass the instance from your configuration
`HelpText`	`string`	`"Enter the script for this event. Click for more information"`	Shown in the dialog's help tooltip. Say what the script is for
`Disabled`	`bool`	`false`	Also disabled while

			the dialog is open
`Class`	`string`	-	Extra CSS classes on the button

UCScriptEditButtonAndLabel

The same button with a help icon and title label in front, as one row.

Parameter	Type	Default
`Title`	`string`	`"Script"`
`Value`	`ScriptDefinition`	-
`HelpText`	`string`	`"Enter the script for this event. Click for more information"`
`Disabled`	`bool`	`false`

```
<UCScriptEditButtonAndLabel Title="On object detected"
                            HelpText="Runs each time the camera reports a new detection."
                            Value="@(_configuration.OnDetectedScript)" />
```

Run the script from the worker with an `Executor` obtained from `ScriptManagerService` - see the
services manual, section "Scripting".

---

8. Logging and status output

UCLogBuffered

A read-only console box with a refresh buffer. **This is the standard place a robot skill tells
the user what it is doing**, and the first place a user looks.
It writes the text with JavaScript rather than re-rendering, which makes it far cheaper than `StateHasChanged`. Entries are kept in a process-wide cache keyed by `InstanceGUID`, so the last
entries survive navigating away and back, and every browser showing the same `InstanceGUID` sees
them. The cache keeps the most recent **100** entries.

Parameter	Type	Default	Notes
`InstanceGUID`	`string`	empty	**Set this.** Normally the worker's instance guid. For a second log in the same skill, append a suffix: `$"{guid}-console2"`. Never a static literal, or two skills share one log
`Lines`	`int`	`0`	The textarea `rows` attribute. `0` lets CSS decide the height - preferred

`FontSize`	`int`	`0`	px. `0` uses the `ARCxConsoleLogFontSize` preference
`RefreshBufferMS`	`int`	`500`	Buffer before pushing text to the browser. Read once at init
`LineHistoryCount`	`int`	`50`	Currently unused; the cache always keeps 100
`Class` / `Style`	`string`	empty	Extra CSS on the textarea

Static entry points, callable from any thread:

```
UCLogBuffered.AddEntry(this, "text");              // from the worker
UCLogBuffered.AddEntry(this, "text");              // from a UIMain component
UCLogBuffered.AddEntry(instanceGuid, "text");      // by guid
UCLogBuffered.ClearLogs(instanceGuid);             // drop the cache for one instance
```

Usage:

```
<div class="arcx-skill-log">
  <UCLogBuffered InstanceGUID="@(_currentRobotSkillWorker.Properties.InstanceGuid)" Lines="3"
/>
</div>
```

`UCLogBuffered` is for the user. `ARCLogService` is for errors and diagnostics (see the services manual). A caught exception goes to `ARCLogService`; "connected to COM3 at 115200" goes
to `UCLogBuffered`.

UCEmptyState

The blank-slate panel: what would appear here, and what the user would do to make it appear.

Parameter	Type	Default	Notes
`Title`	`string`	empty	The situation, plainly: "No COM ports are open"
`Description`	`string`	-	What would appear here and how to get it
`Icon`	`string`	-	A Material icon
`IconColor`	`Color`	`Color.Default`	Muted by default. An empty state is not a warning
`Actions`	`RenderFragment`	-	Buttons offering the way out
`ChildContent`	`RenderFragment`	-	
`Class`	`string`	empty	

9. Video

Both components display frames from a source registered with `VideoRouteService` - a camera robot
skill, for example. The service pushes a JPEG to the component when the source produces a frame
and the component is due for one; the browser fetches it from `/_videoSource/[component guid]`.
Frames are dropped, never queued, so a slow browser only ever sees the latest image. A camera
stub image is shown until the first frame arrives.

UCVideoSourceImage

Parameter	Type	Default	Notes
`VideoRouteSourceName`	`string`	empty	Case-insensitive source name. Empty uses the first registered source (the `*Changed` callback is not raised for that default)
`DisplayProcessedVideo`	`bool`	`true`	`true` shows processed frames (with vision overlays), `false` the originals
`DelayMSBetweenFrames`	`int`	`250`	Minimum ms between frames sent to this browser. Skipped **before** JPEG encoding, so it is a real CPU saving on small boards
`Width` / `Height`	`int?`	`null`	`null` stretches to 100% of the container
`Class` / `Style`	`string`	-	

The `*Changed` callbacks exist for `@bind` support and are not raised by the component.

UCVideoSourceImageAdvanced

The same image with a "Video Source" drop-down listing every registered source, and a "Display
Processed" checkbox above it, so the user picks what to watch without the skill providing its own
controls. Same parameters plus `Alt`.

```
<div class="arcx-skill-media">
  <UCVideoSourceImage
VideoRouteSourceName="@(_currentRobotSkillWorker._Configuration.CameraSource)"
                     DelayMSBetweenFrames="200" />
</div>
```

Store the **source name**, never the guid - guids change every session. In the configuration,

mark the property with `[RobotSkillReference("camera source")]` if it holds another skill's window
title so the AI agent knows what it must match.

---

10. Drawing: UCSkillCanvas

Namespace `ARCx.Components.UserComponents`; the draw list and its types are in `ARCx.Components.UserComponents.UCSkillCanvasComponents`.
A drawing surface for skills that generate graphics: lidar maps, audio UV meters, occupancy grids, path plots, spectrum displays, and maps drawn over satellite photographs or floor plans.
It sends **drawing commands**, not a rendered picture, and lets the browser rasterise on the GPU.
A 360 point lidar sweep is under 3KB on the wire against roughly 40KB for the same thing as a
PNG, with no image encode on the server at all. The server never knows the pixel size of the panel and never needs to, so resizing costs nothing.

Component parameters

Parameter	Type	Default	Notes
`RobotSkillWorker`	`RobotSkillWorkerAbstractEntity`	-	How the skill finds its canvas without holding a component reference
`CanvasName`	`string`	empty	Only needed when one skill shows more than one canvas
`RefreshMS`	`int`	`100`	Shortest gap between frames. `TryBeginDraw` returns false until it has passed, so it is a real CPU limit, not just a display throttle
`OnPointer`	`EventCallback<SkillCanvasPointerArgs>`	-	Leave unset and **no DOM listeners are attached at all**
`PointerMove`	`SkillCanvasPointerMoveEnum`	`WhileDown`	`Never`, `WhileDown`, `Always`. Movement is coalesced to one report per animation frame regardless
`EnableWheel`	`bool`	`false`	Also stops the wheel scrolling the page while over the canvas. Only turn on if the canvas zooms
`Width` / `Height`	`int?`	`null`	`null` stretches to 100%; the browser redraws the last

			frame itself on resize
`Class` / `Style`	`string`	-	

Drawing from the worker

```
if (!UCSkillCanvas.TryBeginDraw(this, out var dl))
  return;                                  // nobody is watching, or it is not time yet

using (dl) {

  dl.View(SkillCanvasViewModeEnum.World, 0, 0, 10);
  dl.Clear();
  dl.Points(_scanXY, _scanPointCount, SkillCanvasColors.Green, 2);
  dl.Circle(0, 0, 0.25f, SkillCanvasColors.Red, 2);
}
```

The `TryBeginDraw` guard is the single most important line for CPU. It is false when no browser is connected, when no panel is showing this canvas, and when the last frame was too
recent, so a skill never spends anything building a frame that would be thrown away.
Static entry points:

Method	Notes
`UCSkillCanvas.TryBeginDraw(worker, out SkillCanvasDrawList dl, string canvasName = null)`	Also overloads taking a `RobotSkillUIMainBaseEntity` or an instance guid string
`UCSkillCanvas.IsDue(worker, string canvasName = null)`	Only needed when gathering the data is expensive and happens before the draw list is wanted
`UCSkillCanvas.ClearFrames(worker)`	Drop the cached last frame

SkillCanvasDrawList

Disposing the draw list sends the frame (`Send()` does the same explicitly). `MAX_VERTICES` is 65535.

Method	Notes
`View(SkillCanvasViewModeEnum mode, float centerX = 0, float centerY = 0, float spanX = 1)`	Sets the coordinate system for everything that follows
`Clear(uint color = SkillCanvasColors.Transparent)`	
`Line(x1, y1, x2, y2, uint color, float thickness = 1)`	
`Rect(x, y, width, height, uint color, float thickness = 1, bool filled = true)`	
`Circle(centerX, centerY, radius, uint color, float thickness = 1, bool filled = false)`	
`Polyline(float[] interleavedXY, int pointCount, uint color, float thickness = 1, bool closed = false)`	

`Points(float[] interleavedXY, int pointCount, uint color, float size = 2, bool square = true)`	
`Text(x, y, string text, uint color, float sizePixels = 12, SkillCanvasTextAlignEnum align = Left)`	
`Image(string url, x, y, width, height, float opacity = 1)`	A picture stretched over a rectangle (map tiles, a floor plan). Only the address is sent; the browser fetches and caches it and repaints the canvas when it arrives. Draw pictures before the geometry over them

`SkillCanvasViewModeEnum`:

Value	**Coordinates**
`Normalized`	X and Y run 0 to 1 across width and height. X and Y scale **independently**. Right for meters, bars, gauges - anything that should stretch to fit
`World`	Real world units with **Y pointing up**, centred on a point you choose. Uniform scaling, so circles stay circular. Right for a lidar map, occupancy grid or path plot
`Pixels`	CSS pixels from the top left, Y down. Only when the drawing genuinely depends on pixel size

`SkillCanvasColors` packs colours as `0xAARRGGBB` in a `uint` - four bytes on the wire, zero allocations. Constants: `Transparent`, `Black`, `White`, `Red`, `Green`, `Blue`, `Yellow`, `Orange`, `Cyan`, `Magenta`, `Grey`, `DarkGrey`. Helpers: `Rgb(r, g, b)`, `Rgba(r, g, b, a)`, `FromHex(string)`.

Pictures and map tiles

`Image(url, x, y, width, height, opacity)` draws a picture stretched over a rectangle in the current view's units. Only the address travels in the frame, so a map made of thirty tiles costs a few hundred bytes a frame however large the pictures are.

- **The browser fetches the picture**, once, and keeps it in a cache shared by every canvas on the page (the newest 600 pictures). Later frames draw it from there. The address therefore has to be reachable from the browser, not from the robot: an `https://` tile server, a path ARCx serves (`/_content/...`, or a `UCDynamicImage`-style route of your own), or a `data:` URI for something small.
- **A picture that has not arrived is simply missing from that frame.** When it lands, the canvas repaints its last frame by itself, so a map fills in tile by tile without the skill sending anything more.
- **Draw pictures first** and the vector drawing over them. Later operations paint over earlier ones.
- In a `World` view (Y up) the opposite corner is `(x + width, y + height)`, so `y` is the picture's **bottom** edge. Corners are transformed like `Rect`, so the picture stays the right way up.
- Pictures entirely off the canvas cost nothing to draw, so sending a margin of tiles beyond the edges (to cover the next pan) is fine.
- A page served over `https` cannot load `http` pictures (the browser blocks mixed content).

Pictures from another site are fine: the canvas is never read back.
• Respect the tile server's terms. OpenStreetMap's tile servers are for light use and need a credit on the map; most commercial servers need a key and a credit too. Draw the credit with `Text` in a `Normalized` view.

A satellite map background, from the standard web map tile scheme (Web Mercator, zoom 0 is the whole world in one 256 pixel tile):

```
dl.View(SkillCanvasViewModeEnum.World, (float)viewX, (float)viewY, (float)spanMetres);
dl.Clear(SkillCanvasColors.Black);

// The zoom whose pixels best match the screen's
int z = (int)Math.Round(Math.Log2(156543.03 * Math.Cos(centreLat * Math.PI / 180) /
metresPerScreenPixel));

for (int ty = rowNorth; ty <= rowSouth; ty++)
  for (int tx = colWest; tx <= colEast; tx++) {

    // Each tile's corners as latitude/longitude, then into the map's metres
    var (latN, lonW) = tileCorner(tx, ty, z);
    var (latS, lonE) = tileCorner(tx + 1, ty + 1, z);
    var sw = toLocal(latS, lonW);
    var ne = toLocal(latN, lonE);

    dl.Image($"https://server.arcgisonline.com/ArcGIS/rest/services/World_Imagery/MapServer/
tile/{z}/{ty}/{tx}", (float)sw.X, (float)sw.Y, (float)(ne.X - sw.X), (float)(ne.Y - sw.Y),
0.85f);
  }

// ...then the zones, the route and the robot over the photograph...

dl.View(SkillCanvasViewModeEnum.Normalized);
dl.Text(0.99f, 0.97f, "Imagery © Esri, Maxar, Earthstar Geographics",
SkillCanvasColors.White, 9, SkillCanvasTextAlignEnum.Right);
```

The server never knows the canvas's pixel width. A skill that needs it to choose a zoom can work it out from a pointer event: `PixelX` against the world `X` it unprojected to, as the Outdoor Navigator robot skill does.

Pointer input

`SkillCanvasPointerArgs` arrives with `X` and `Y` **already converted into the units of the first**
view the last frame set up - a click on a lidar map is handed to the skill in metres, with no
unprojecting to do.

Member	Notes
`EventType`	`Down`, `Move`, `Up`, `Leave` (treat as `Up`; no `Up` is coming), `Wheel`
`X`, `Y`	In view units
`PixelX`, `PixelY`	Raw CSS pixels
`Buttons`	Bitmask
`IsButtonDown`	Convenience
`WheelDelta`	Direction and amount
`ShiftKey`, `CtrlKey`, `AltKey`	

---

11. Serving your own images: UCDynamicImage

For an image the skill generates or holds in memory but that is **a genuine bitmap** - a

saved
photo, a decoded frame, a map rendered with SkiaSharp. For generated line graphics prefer `UCSkillCanvas`, which costs far less.
The browser requests `/_dynamic/[instance guid]/[FilePath]` and `DynamicContentMiddleware` routes
that to the worker's override:

```
public override DynamicContentMiddlewareResponse
OnDynamicResourceRequest(DynamicContentMiddlewareRequest request)
```

Parameter	Type	Notes
`RobotSkillWorker`	`RobotSkillWorkerAbstractEntity`	Builds the URL from the instance guid and routes the request to that worker
`FilePath`	`string`	The path your override checks for, i.e. `"/myImage.jpg"`
`Width` / `Height`	`int?`	`null` leaves it to the image/CSS
`Style`	`string`	Appended after the size rules
`Alt`	`string`	Defaults to the request URL

`RefreshImage()` appends a cache-busting counter and sets `img src` with client-side JavaScript -
no Blazor re-render. **Call it whenever the image content changes.** It does nothing when no
clients are connected. Fire and forget (`async void`); exceptions are logged.
Have the bytes cached and ready in the override; it runs on an HTTP request thread.

---

12. Movement and input

UCMovementButtons

A forward / left / stop / right / reverse pad plus a small text box that, when focused, drives the
robot with the arrow keys (key up stops). The buttons highlight the direction `MovementManagerService` reports, whoever set it.

Parameter	Type	Notes
`OnMovementChange`	`EventCallback<MovementManagerService.MovementDirectionEnum>`	**The handler is responsible for moving the robot**, normally by calling `MovementManagerService`. The keyboard path calls the service directly

UCMovementJoyPad

An on-screen joystick wired to `MovementManagerService`. Forward/back drives, with left and right
speeds blended by the sideways offset; sideways within the dead zone turns; centre stops. **The
component drives the robot itself** - the skill handles no events.

Parameter	Type	Default	Notes
`Sensitivity`	`double`	`0.2`	Dead zone as a fraction of range (0 to 1)
`MaxSpeed`	`byte`	`255`	Speed at the edge of the range
`Width` / `Height`	`int`	`320` / `240`	Canvas size in pixels
`ButtonRadius`	`int`	`70`	Thumb radius
`ReturnToCenterWhenReleased`	`bool`	`true`	Springs back and stops the robot
`CanvasImage` / `ButtonImage`	`string`	-	Optional replacement artwork
`MaxDriveSpeed`	`byte`	`MovementManagerService.MAX_SPEED`	Currently unused
`Refresh()`	method	-	No-op, kept so existing skills still compile. The canvas sizes itself and a `ResizeObserver` redraws it

UCJoypad

The raw joystick with no movement wiring - use it for anything that is not locomotion (a camera
gimbal, a pan/tilt head).

Parameter	Type	Default	Notes
`OnJoyPadMove`	`EventCallback<JoyPadArgs>`	-	`JoyPadArgs.X` and `.Y` run **-1 to +1**; zero is centre, negative Y is up
`Width` / `Height`	`int`	`320` / `240`	A nominal **aspect ratio**, not a pixel size - the canvas fills its container
`ButtonRadius`	`int`	`70`	Scaled with the canvas
`ReportIntervalMs`	`int`	`50`	Minimum ms between C# callbacks (~20Hz)
`ReturnToCenterWhenReleased`	`bool`	`true`	
`CanvasImage` / `ButtonImage`	`string`	-	

UCNumberSlider

A compact draggable number field.

Parameter	Type	Default
`Value` / `ValueChanged`	`int` / `EventCallback<int>`	supports `@bind-Value`
`MinValue` / `MaxValue`	`int`	`0` / `100`
`Disabled`	`bool`	`false`

13. Kinematics

UCKinematicChainSelector

A drop-down of the kinematic chains published in the project, bound to the chain **name**. The
list is live: it refreshes when a provider adds or removes a chain.

Parameter	Type	Default	Notes
`Value` / `ValueChanged`	`string` / `EventCallback<string>`	empty	The chain name. Supports `@bind-Value`
`OnChainSelected`	`EventCallback<KinematicChain>`	-	Raised alongside `ValueChanged` with the descriptor. Null when "(none)" is chosen
`Label`	`string`	`"Kinematic configuration"`	
`HelpText`	`string`	empty	When empty, the selected chain's provider and joint count are shown instead
`AllowNone`	`bool`	`false`	Offers a "(none)" entry. Also turns off `AutoSelectFirst`
`AutoSelectFirst`	`bool`	`true`	Selects the first chain when nothing is selected, so a freshly added skill works without a visit to its configuration. Also applies when a chain is published after the page rendered
`ShowPose`	`bool`	`false`	Shows the live end effector pose (re-rendered at most every 100ms)
`Disabled`	`bool`	`false`	Also honoured when a parent settings row cascades `Disabled`

```
<UCKinematicChainSelector @bind-Value="@(_configuration.ChainName)" ShowPose="true" />
```

Save the **name**. At run time pass it to `KinematicsService.MoveTo(name, x, y, z)` or `GetPose(name)`.

---

14. Page furniture and misc

Component	Purpose	Key parameters
`UCPageHeader`	The page's single `h1`, with an eyebrow, one-sentence description, icon badge and right-aligned actions. For full-page UIs, not skill windows	`Title`, `Eyebrow`, `Description`, `Icon`, `IconColor` (default `Color.Primary`), `Actions`, `Class`
`UCYouTube`	An embedded YouTube player	`YouTubeID`, `Title`, `Class`, `Style`
`UCNotes`	The project notes editor	-
`UCAiChat` (`...UserComponents.AiChat`)	The Athena AI chat panel	-
`UCBlockly` (`...UserComponents.Blockly`)	The Blockly workspace. Normally reached through `UCScriptEditButton` rather than hosted directly	-
`UCRobotImageMap`, `UCRobotPortPanel` (`...UserComponents.RobotConfig`)	The robot picture with port markers, used by the port pickers	-

---

15. Popups: snackbars, dialogs, navigation

`ARCx.Components.Layout.UCPopupController`. These reach **every connected browser**. `UCPopupController.Instance` is the controller of the most recently connected browser, and is
null when no browser is connected - always null-check it. Use `Instance` only for things that
must target exactly one browser (streaming a file download). Informational calls broadcast to all
browsers anyway.

Snackbar

`ShowSnackbarMessage(Core.SnackBar.SnackBarItem e)` shows a short notification in the corner of
every connected browser.

`SnackBarItem` member	Default	Notes
`Message`	empty	Shortened to its first line and sentence, and to about 120 characters
`Severity`	`Severity.Normal`	MudBlazor severity; defines

		icon and colour
`ButtonText`	`"Ok"`	
`OnClick`	`null`	Null means the button only dismisses
`Key`	a new guid	**Reuse the same key to replace an earlier popup** instead of stacking a new one, i.e. when reporting the changing status of one operation

Dialog

`ShowDialogMessageBox(...)` shows a modal on every browser and blocks until the first one responds (or the cancellation token fires); the other dialogs then close. Returns false when no
browser is connected.

`DialogMessageItem` member	**Default**
`Title`	`"Title"`
`Message`	`"Message"`
`OKButtonText`	`"OK"`
`CancelButtonText`	`null` (no cancel button)
`ResponseValue`	`false`. True for OK; false for Cancel, no browser, or cancelled

This path is primarily for the scripting engines, because it uses a cancellation token to close the popup when the script that opened it is stopped. A robot skill UI normally uses a MudBlazor
dialog of its own instead.

---

16. Custom UI widgets

The Custom UI is where a user lays out their **own control panel screens** for the robot - buttons, sliders, joysticks, camera views - and shows them full screen or on a phone.
Two ways a robot skill appears there:

1. **The built-in "Robot skill window" widget**, which embeds `RobotSkillUIMain.razor` exactly as
   it appears on a workspace. This works with no code at all.
2. **Widgets the skill provides itself**: compact, purpose-built pieces of the skill.

Write a widget only when the skill has something a control panel actually wants - a gauge, a reading, one or two big buttons. **A widget is not a second copy of the main window**; it is the
part of it that belongs on a phone in somebody's hand.
Declare them on the worker:

```
public override List<CustomUIWidgetDefinition> GetCustomUIWidgets() {

  return new List<CustomUIWidgetDefinition>() {

    new CustomUIWidgetDefinition() {

      // Saved in the user's project, so keep it stable once the skill is published
```

```
      Key = "counter",

      Title = "Counter",
      Description = "The counter, with start/stop and add buttons.",

      ComponentType = typeof(RobotSkillUIWidget),

      // The size it is dropped at. The user can resize it afterwards.
      DefaultWidth = 200,
      DefaultHeight = 140,

      DefaultSettings = new Dictionary<string, string>() {
        { "Caption", "Counter" },
        { "ShowButtons", "true" },
      },

      SettingsHelp = "Caption: the label above the number. ShowButtons: true or false.",
    },
  };
}
```

`CustomUIWidgetDefinition` member	Default	Notes
`Key`	empty	Saved in the project. Keep it stable once published
`Title` / `Description`	empty	Shown in the designer palette
`ComponentType`	-	A component deriving from `RobotSkillUIWidgetBaseEntity`
`DefaultWidth` / `DefaultHeight`	`240` / `160`	Design pixels
`DefaultSettings`	empty	Seeds the `Settings` dictionary the widget reads
`SettingsHelp`	empty	Explains the settings in the designer

The designer calls `GetCustomUIWidgets()` whenever its palette is shown, so keep it quick.
Writing the widget component:

- Give your root element `width: 100%; height: 100%` and let it fill the box the user sized.
- Read settings with `GetSetting()`, `GetSettingNumber()`, `GetSettingBool()`.
- **Honour `IsEditMode`**: draw a preview, disable the buttons, move nothing.
- Call `StateHasChanged()` (the base class one) to redraw. The worker's `RefreshUI()` already
calls it, so a widget usually needs no timer of its own.
- When data only a widget shows is expensive to produce, the worker can skip it while `HasCustomUIClient` is false. See Skip work nobody will see.
- Several browsers may show the same widget at once, each with its own instance. Shared state
belongs on the worker.
- Scale text with container query units so the widget reads at any size the user drew:

```
.tpl-widget { container-type: size; }                    /* declare on the PARENT */
.tpl-widget-value { font-size: clamp(1.2rem, 30cqh, 5rem); }
```

If the skill offers no widget, delete both `RobotSkillUIWidget.razor` **and** the `GetCustomUIWidgets()` override - they go together.
Scripts address the Custom UI itself as a pseudo robot skill: `ControlCommand("Custom UI", "Show", "My screen")`.

---

17. MudBlazor conventions ARCx expects

- **Give every button an explicit `Variant` and `Color`.** Without them a `MudButton` renders as
plain text and does not read as a button. `Variant.Filled` for the primary action, `Variant.Outlined` for the rest.
- Use `UCTitleComponentTableRowColumn` rows in a configuration page rather than hand-rolled
labels, so every skill's settings look the same.
- Wrap anything non-obvious in a `MudTooltip` - **except inside a repeated row**, where the plain
`title` attribute says the same thing for none of the cost. See
section 3.
- Use `Icons.Material.Filled.*` for icons.
- Theme colours come from MudBlazor CSS variables - `var(--mud-palette-surface)`, `var(--mud-palette-lines-default)`, `var(--mud-palette-text-secondary)`. **Never hard-code a**
colour, or the skill breaks in the other theme.
- Add `@using` directives to `_Imports.razor` rather than repeating them per file. The template
already imports `ARCx.Components.UserComponents`, `ARCx.Components.UserComponents.UCServoSelectorComponents`, `ARCx.Core`, `ARCx.Core.ARCLog`,
`ARCx.Core.Disposable` and `MudBlazor`.
- A robot skill project needs `<FrameworkReference Include="Microsoft.AspNetCore.App" />`; the
Razor SDK does not bring it in on its own and ARCx's public API is Blazor-typed.

Static files in a robot skill

Mark a file as an **embedded resource** in the skill project and reference it as `/_content/[assembly name]/[resource name]`:

```
<MudImage Src="/_content/PluginRobotSkillTemplate/capture.jpg" />
```

Where to look next

- `ARCx.xml` - beside `ARCx.dll`, in the folder this copy of ARCx runs from. The full XML documentation for every public class, method, property and event. Search it rather than reading
it; every entry is a `<member name="...">` where `T:` is a type, `M:` a method, `P:` a property, `E:` an event, `F:` a field.
- `RobotSkill-Services-And-Core.md` - the non-visual API: EZB servos and I/O, video routing,
audio, speech, movement, kinematics, navigation, scripting, logging and configuration.

Robot Skill Services and Core

What a robot skill can call: the robot controllers (servos, digital, ADC, PWM, I2C, UART), video, audio, speech, movement, kinematics, navigation, game controllers and HID devices, browser windows on the robot's own screens, neural networks with ONNX Runtime,

scripting,
variables, logging and configuration.
This is the companion to `RobotSkill-UI-Components.md`, which covers the visual side. Both sit
beside `ARCx.exe`, `ARCx.dll` and `ARCx.xml`. **`ARCx.xml` is the authoritative reference** for every public member; this manual tells you what exists, which of several overloads to reach for,
and the rules that are not obvious from a signature.

Contents

1. The worker: lifecycle and overrides
2. Getting hold of a service
3. EZBManager: the robot controllers
4. Servos
5. Digital, ADC, PWM
6. I2C, UART and ping sensors
7. Serial COM ports
8. Video routing
9. Audio
10. Text to speech
11. Speech to text
12. Movement
13. Kinematics
14. Navigation messaging (NMS)
15. Auto Position
16. Scripting, control commands and variables
17. Talking to other robot skills
18. The project
19. Configuration and the JSON schema
20. Logging
21. Persistence and runtime state
22. Custom UI and robot configuration services
23. Utilities
24. Gamepads, joysticks and HID devices
25. Browser windows: web pages, dashboards, video calls and kiosks
26. Depth cameras
27. Robot Bus: topics, services and frames
28. Rules that will bite you

---

1. The worker: lifecycle and overrides

`ARCx.Services.ProjectTools.RobotSkillWorkerAbstractEntity`, which derives from `ARCx.Core.Disposable.DisposableBase`.
The worker holds the skill's logic and state. One instance per skill instance in the project, shared by every browser.

Lifecycle, in order

Member	Notes
Constructor	**Create nothing here.** No timers, no executors, no subscriptions
`abstract RobotSkillWorkerDefaultParams Setup()`	Window defaults: `Width` (320), `Height` (200), `AllowMultipleInstances` (true),

	`AllowFullScreen` (true). Applied when the skill is added to a project; width and height only when the saved properties have none
`abstract void Initialize()`	Create objects here: timers, executors, service registrations. Errors here are handled cleanly by the skill loader
`virtual void OnSetConfiguration(OnSetConfigurationParams p)`	The saved configuration arrives. `p.IncludesConfig` is false for a skill the user just added by hand - use your defaults. `p.ConfigStream` is **owned by the caller; do not dispose it**
`virtual void OnProjectLoadCompleted()`	Every skill is loaded and configured by now. **The only safe place to look up another robot skill**
`virtual void GetConfigOnProjectSave(Stream s, JsonSerializerOptions o)`	Write your configuration. It is your stream; a zip holding many files is fine
`protected abstract void DisposeOverride()`	Unsubscribe from every event and dispose everything. A leaked handler pins the plugin assembly and the skill will not rebuild

Between `Setup()` and `Initialize()` ARCx assigns `Properties` (the window title made unique),
creates the never-rendered prototype UI instances and the speech engine, and checks the single-instance rules: a second TTS engine, a second movement panel, or a second instance of a
skill whose `Setup()` returned `AllowMultipleInstances = false` is refused there. During `Initialize()` the skill is **not yet in the project list**, so other skills cannot see it.
Throwing from `Initialize()` aborts the add and unloads the assembly.

How each way into a project calls the worker

Entry	**`OnSetConfiguration`**	**`OnProjectLoadCompleted`**
Project load	`IncludesConfig = true`, stream from the project file	Raised on every skill after all are loaded
Merge	`IncludesConfig = true`	Raised on **every** skill in the project, not only the merged ones
User adds from the menu, or the AI agent adds	`IncludesConfig = false`, `ConfigStream` null	Raised on the new skill only
Duplicate button on the workspace	`IncludesConfig = false`	Raised on the new skill only
New project	`IncludesConfig = false` for each default skill	Raised on all
Programmatic configuration change (config page save, AI agent, another skill)	`IncludesConfig = true`, wrapped by `OnConfigWindowOpening` / `OnConfigWindowClosed`	Not raised

Exceptions from `OnSetConfiguration` and `OnProjectLoadCompleted` during a load are caught,
logged and reported as warnings; the project still loads. A programmatic configuration change
does not set `AllWorkersPaused`.

Loading and removal

Each robot skill **instance** gets its own collectible `AssemblyLoadContext`. Dependencies and native libraries resolve through the skill's `.deps.json`, so two skills can ship different versions of one package. `ARCx`, `MudBlazor`, `Microsoft.AspNetCore.Components` and `Microsoft.JSInterop` always come from ARCx's own copy, whatever the skill ships.
Two instances of one skill mean two copies of every DLL it bundles. A library that finds its own
assembly by name across the AppDomain (the OPC UA stack does) can pick up the other instance's
copy and fail with a cast error. Suspect this whenever a bundled library works with one instance
and fails with two.
ARCx takes the first type in the assembly assignable to each base class: `RobotSkillWorkerAbstractEntity` and `RobotSkillUIMainBaseEntity` (required), `RobotSkillUIConfigBaseEntity` and `SpeechToTextEngineAbstractEntity` (optional). Each is
created with `ActivatorUtilities.CreateInstance`, which is why constructors can take services.
On removal (or clearing the project) ARCx, in order: removes the worker from the project list,
clears its logs and canvases, disposes the UI components, raises `ProjectToolsService.OnRobotSkillRemoved`, calls `DisposeOverride()`, unloads the load context and
runs a garbage collection. Anything still subscribed keeps the context alive and the DLL locked.

State to check

Member	Notes
`static bool AllWorkersPaused`	True while **any** skill's configuration window is open. Timers should skip their work
`IsDisposing` / `IsDisposed`	From `DisposableBase`. A queued timer event can fire after removal; bail out
`AnyActiveClients`	True when a browser is connected - any browser, even one looking at a different workspace
`HasUIClient`	True when a browser is showing **this skill's window** (and so any `UCRefreshRegion` in it)
`HasCustomUIClient`	True when a browser is showing a Custom UI screen with one of **this skill's widgets** on it
`HasAnyUIClient`	`HasUIClient` or `HasCustomUIClient`. The check that matches `RefreshUI()`. Skip building display-only data when it is false

`Properties`	`RobotSkillProperties`: `InstanceGuid`, `StoreGUID`, `Title`, `WindowTitle`, `Notes`, `Width`, `Height`, `DesktopNum`
`AllowMultipleInstances`, `AllowFullScreen`	As returned from `Setup()`
`CurrentRobotSkillUIMainComponent`, `CurrentRobotSkillUIConfigComponent`	The rendered components, when there are any

`WindowTitle` is the unique per-instance name the user sees and the address scripts use. `InstanceGuid` survives a rename, so a configuration that points at another skill is better off remembering the guid - but a user-facing setting is normally the window title, marked with `[RobotSkillReference]`.

Redraw

Member	Notes
`RefreshUI()`	Redraws the skill window on every browser **and** every Custom UI widget the skill provides. Safe from any thread, safe before anything is rendered, does nothing when nobody is watching
`RefreshCustomUIWidgets()`	Only the widgets
`RefreshUIRegion(name)`	Only the `UCRefreshRegion` components with that name, on every browser. Use it for what a loop updates often (a play position, a reading) so the rest of the window is not redrawn. See `RobotSkill-UI-Components.md` section 3

Other overrides

Member	Notes
`virtual Type GetConfigurationType()`	Return the configuration class so ARCx can build its JSON Schema. Return null to opt out. See section 19
`virtual ConfigWindowOpenStatus OnConfigWindowOpening()`	Set `CanOpen = false` and a `ReasonMessage` to refuse. Stop threads here if the configuration cannot change under them
`virtual void OnConfigWindowClosed(ConfigWindowClosedStatus s)`	`s.ConfigChanged` says whether SAVE or CANCEL was pressed. Call `base`
`virtual void UpdateMainUIStateRequested()`	The UI component is initializing and wants the latest display values. Set model variables only - do **not** call `RefreshUI()`, and there are no rendered references yet
`virtual List<ControlCommandItem> GetSupportedControlCommands()`	See section 16
`virtual object ControlCommand(ControlCommandArgs`	Fall through to `base.ControlCommand(args)`, which throws

`args)`	the "no such command" exception for the caller
`virtual DynamicContentMiddlewareResponse OnDynamicResourceRequest(DynamicContentMiddlewareRequest r)`	Serve bytes at `/_dynamic/[instance guid]/...`. See the UI manual, `UCDynamicImage`
`virtual List<CustomUIWidgetDefinition> GetCustomUIWidgets()`	Widgets offered to the Custom UI
`virtual List<BlocklySoundItem> GetBlocklySoundItems()`	Sounds for Blockly's Play Audio blocks. `Title` is the choice shown, `Code` the JavaScript that starts the sound, `WaitCode` the JavaScript that plays it and returns when it has finished (leave empty if the skill cannot tell; the wait block then does not wait). `Code` is also what the block saves, so keep it the same for the same sound
`virtual List<BlocklyCameraItem> GetBlocklyCameraItems()`	Actions for Blockly's Camera category. `Code` is a JavaScript call without a semicolon, used both as a statement and as a value. Most skills return `BlocklyCameraItemsFromControlCommands()`, which offers every control command that runs without extra input
`virtual GetBlocklyAutoPositionItemsResponse GetBlocklyAutoPositionItems(...)`	Auto Position actions offered to Blockly

Interfaces a worker can implement

Interface	Effect
`ARCx.Services.Movement.MovementPanelInterface`	Declares the skill the project's **movement panel**: it subscribes to `MovementManagerService` events and drives the motors. Marker interface, no members. Only one per project; adding a second is refused
`ARCx.Services.TextToSpeech.TextToSpeechEngineInterface`	Declares the skill the project's **TTS engine**. One method, `OnTextToSpeech(TextToSpeechRequestCls)`. Only one per project

Speech recognition engines are wired differently - see section 11.

---

2. Getting hold of a service

ARCx services are registered as **singletons** (one robot, not one per browser tab). Inject them
through the **worker's constructor**:

```
public class RobotSkillWorker : RobotSkillWorkerAbstractEntity {

  readonly VideoRouteService _videoRouteService;
```

```
  readonly ProjectToolsService _projectToolsService;

  public RobotSkillWorker(VideoRouteService videoRouteService, ProjectToolsService
projectToolsService) {

    // Constructor only stores references. Objects are created in Initialize()
    _videoRouteService = videoRouteService;
    _projectToolsService = projectToolsService;
  }
}
```

The injectable services:

Type	Namespace	Covered in
`AiAgentService`, `AiAgentToolExecutor`	`ARCx.Services.AiAgent`	-
`AudioService`	`ARCx.Services.AudioServices`	9
`BrowserPresentationService`	`ARCx.Services.BrowserPresentation`	25
`CheckForUpdateService`	`ARCx.Services.CheckForUpdate`	-
`ConfigRegistryManagerService`	`ARCx.Services.ConfigRegistry`	21
`CustomRegistryManagerService`	`ARCx.Services.CustomRegistry`	21
`CustomUIService`	`ARCx.Services.CustomUI`	22
`DepthSensorService`	`ARCx.Services.DepthSensing`	26
`EZBScanService`	`ARCx.Services.EZBScan`	-
`InputDeviceService`	`ARCx.Services.InputDevices`	24
`JsARCxService`	`ARCx.Services.JSHelper`	-
`KinematicsService`	`ARCx.Services.Kinematics`	13
`MovementManagerService`	`ARCx.Services.Movement`	12
`NMSService`	`ARCx.Services.NMS.NMS2Dv1`	14
`PluginManagerService`	`ARCx.Services.PluginManager`	-
`ProjectToolsService`	`ARCx.Services.ProjectTools`	17, 18
`RobotConfigService`	`ARCx.Services.RobotConfig`	22
`RobotBusService`	`ARCx.Services.RobotBus`	27
`RuntimeStateService`	`ARCx.Services.RuntimeState`	21
`ScriptManagerService`	`ARCx.Services.ScriptMana`	16

	`ger`	
`AccountSecurityService`	`ARCx.Services.Security`	-
`SpeechToTextService`	`ARCx.Services.SpeechToText`	11
`StartupScriptService`	`ARCx.Services.StartupScript`	-
`SynthiamFeedService`	`ARCx.Services.SynthiamFeed`	-
`TextToSpeechService`	`ARCx.Services.TextToSpeech`	10
`VideoRouteService`	`ARCx.Services.VideoRouting`	8
`ThemeStateService`	`ARCx.Services.Theme`	Scoped (per browser), not for workers

Not injected - these are static and called directly:
`EZBManager`, `VariableManager`, `ARCLogService`, `ComPortManager`, `FileManagerService`, `ConfigSchemaBuilder`, `UCLogBuffered.AddEntry`, `UCSkillCanvas.TryBeginDraw`, `ProjectToolsService.CurrentProjectProperties`.

---

3. EZBManager: the robot controllers

`ARCx.Core.EZBManager` (static). The EZB classes live in `ARCx.Core.EZBController`.
An "EZB" is a robot controller. A project always has at least one, at index 0. The list is contiguous - there is never a gap between 0 and `EZBSlotCount - 1` - and removing one shifts the
rest down.

Member	**Notes**
`EZB PrimaryEZB`	Index 0. Always exists. Used when no board index is given, and by the movement panels and speech recognition
`EZB[] EZBs`	A snapshot array. No entry is ever null
`int EZBSlotCount`	Number of controllers
`EZB GetEZB(int index)`	**Throws** when out of range
`EZB TryGetEZB(int index)`	**Null** when out of range. Use this when a project may reference a board that is no longer there
`int GetIndexOfEZB(EZB ezb)`	`-1` when not in this project
`int AddEZBToList()`	Returns the new index
`void RemoveEZBByIndex(int id)`	
`void ClearEZB()`	Disposes all and recreates index 0. Used on project close/load
`event OnConnectionChange(EZB ezb, bool isConnected)`	Raised on whichever thread connected or disconnected

`event EventHandler OnPanicReleaseServosAndStopMovementPressed`	The panic button. Stop whatever your skill is doing
`RandomUnique GetRandomUnique`	A random-and-unique value generator
`static readonly DateTime EZBuilderLoadedDateTime`	Effectively when ARCx started

The EZB object

Member	Notes
`bool IsConnected`	**Check this before every hardware call**
`void Connect(string hostname)` / `Connect(string hostname, int baudRate)`	Hostname is a COM port name, or an IP such as `192.168.1.5:23`. Baud is ignored for TCP
`Task ConnectAsync(string hostname, CancellationToken)` / `ConnectAsync(string hostname, int baudRate, CancellationToken)`	Same as `Connect`, without holding the calling thread. Use this from Blazor event handlers so the page stays responsive during the connect timeout and handshake. Only one connect runs per EZB at a time; a second throws
`int Index`, `string GUID`	
`string ConnectedEndPointAddress`	COM port or IP of the current connection
`FirmwareCls Firmware`	Query the connected board's capabilities
`bool PingController()`	Is it still responding
`event OnConnectionChange(EZB sender, bool isConnected)`	
`event OnDataSend(byte[] data, int expectedReturnSize)` / `OnDataReceive(byte[] data)`	
`TimeSpan ConnectedTimeSpan`, `DateTime LastCommandSentDateTime`, `LastCommandReturnDateTime`	
`int BaudRate`, `bool DTREnable`, `bool RTSEnable`, `SerialPort _Serial`	Advanced serial control
`TCPServer TCPServer`, `StartServer(int port)`, `StopServer()`	For an EZB that connects **to** ARCx
`ProtocolVersionEnum GetProtocolVersionEnum()`, `string GetProtocolVersion()`	

Peripheral sub-objects on the EZB: `Servo`, `Digital`, `ADC`, `PWM`, `I2C`, `Uart`, `HC_SR04`, `Audio`, `Power`, `MusicSynth`, `Recorder`, `ConfigurationManager`.

The pattern for every hardware call:

```
var ezb = EZBManager.TryGetEZB(_Configuration.Port.BoardIndex);

if (ezb == null || !ezb.IsConnected)
  return;
```

```
var value = ezb.ADC.GetADCValue(_Configuration.Port.Port);
```

---

4. Servos

Port enums

`Servo.ServoPortEnum` covers `D0`..`D255` (physical digital ports), then `V0`..`V255` (**virtual** ports, which robot skills such as a Dynamixel or Feetech driver hook onto and drive over their own bus), then `NA` for "not assigned".
Constants: `Servo.SERVO_MIN` (1), `Servo.SERVO_OFF` (0), `Servo.SERVO_MAX` (the project's `MaxServoPositions`, default 180), `Servo.SERVO_CENTER`, `Servo.SERVO_PWM_VELOCITY_FASTEST` (0), `Servo.SERVO_PWM_VELOCITY_SLOWEST` (20).

The descriptor helpers (preferred)

`ARCx.Config.Helpers.ServoDescriptor` is what a skill stores in its configuration, normally inside a `ServoCollection` edited by `UCServoSelector`.

`ServoDescriptor` member	**Default**	**Notes**
`Port`	`NA`	
`BoardIndex`	`0`	
`Minimum` / `Maximum`	`SERVO_CENTER`	Positions outside are clamped
`Invert`	`false`	Increasing input moves toward `Minimum`
`Multiplier`	`1` (decimal)	So a servo can follow another at a different ratio
`Velocity` / `Acceleration`	`-1`	`-1` leaves the servo's own value untouched
`Description`	empty	Shown in the servo selector

`ServoCollection.Servos` is a `List<ServoDescriptor>`, defaulting to one unassigned servo.
`Servos[0]` is the master; every following servo derives its position from the first using its own `Multiplier`.
The `EZBManager` helpers apply all of that for you:

Method	**Notes**
`SetServoPosition(ServoCollection servos, int position)`	Exact position, clamped per servo
`SetServoPosition(ServoDescriptor servo, int position)`	
`SetServoPositionScalar(ServoCollection servos, int inputMin, int inputMax, int input)`	Arduino `map()` for servos: maps the input range onto each servo's own min..max
`SetServoPositionScalar(ServoCollection servos, float inputMin, float inputMax, float input)`	Float overload, so `0..1` with `0.5` means halfway
`SetServoIncrement(ServoCollection`	

`servos, int steps)` / `(ServoDescriptor, int)`	
`GetServoPosition(ServoCollection servos)` / `(ServoDescriptor)`	The last **commanded** position, not a reading. For a bi-directional servo use `GetServoPositionRealtime()`
`SetReleaseServo(ServoCollection)` / `(ServoDescriptor)`	Release from holding
`GetReleaseServo(ServoCollection servos)`	Whether the first servo is released
`ReleaseAllServos()`	Every servo on every board, and raises the panic event

Moving many servos in sync

One command per board rather than one per servo, which keeps a multi-servo move together:

```
var buffers = EZBManager.CreateServoBuffers();

EZBManager.TryBufferServoPosition(buffers, boardIndex, new ServoPositionItem(port,
position));
// ... buffer more ...

EZBManager.DispatchServoBuffers(buffers);
```

`TryBufferServoPosition` silently ignores a board index that is out of range, so a project referencing a missing controller moves the servos it can rather than throwing.

The raw servo API

`ezb.Servo` when you need a specific port directly. Classes are in `ARCx.Core.EZBController.Classes`.

Method	**Notes**
`SetServoPosition(ServoPortEnum port, int position)`	
`SetServoPosition(ServoPortEnum port, int position, int velocity)`	
`SetServoPosition(ServoPortEnum port, int position, int velocity, int acceleration)`	
`SetServoPosition(ServoPositionItem[] servos)`	One command for many servos on this board
`SetServoPositionScalar(port, servoMin, servoMax, clientWidthMin, clientWidthMax, clientPosition, bool invert)`	`int` and `float` overloads
`GetServoPosition(ServoPortEnum port)`	Last commanded
`GetServoPositionRealtime(ServoPortEnum port)`	Reads the servo. Bi-directional servos only
`SetServoVelocity(port, int)` / `(ServoPortEnum[], int)` / `GetServoVelocity(port)`	

`SetServoAcceleration(port, int)` / `(ServoPortEnum[], int)` / `GetServoAcceleration(port)`	
`SetServoMin` / `GetServoMin` / `ResetServoMinLimits`	**Hard limits held on the controller.** `RobotConfigService` pushes the robot's body limits down through these, and every `SetServoPosition` is clamped by them
`SetServoMax` / `GetServoMax` / `ResetServoMaxLimits`	
`SetServoFineTune` / `GetServoFineTune` / `ResetServoFineTune`	Per-servo trim
`ReleaseServo(port)` / `(ServoPortEnum[])` / `ReleaseAllServos()`	
`bool IsServoReleased(port)`	
`double GetNumberOfSecondsSinceLastMove(port)`	
`ResetAllServoVelocities()`	

Events on `ezb.Servo`, for a skill that mirrors or logs servo traffic: `OnServoMove`, `OnServoMoveDetailed`, `OnServoMoveDetailed2` (includes the `EZB`), `OnServoVelocity`, `OnServoAcceleration`, `OnServoRelease`, `OnServoGetPosition`.

---

5. Digital, ADC, PWM

Store the port in a descriptor (`DigitalPortDescriptor`, `ADCPortDescriptor`, `PWMPortDescriptor` in `ARCx.Config.Helpers`) edited by the pickers in the UI manual.

Digital - ezb.Digital

`DigitalPortEnum`: `D0`..`D255`, plus `NA`.

Method	Notes
`void SetDigitalPort(DigitalPortEnum port, bool status)`	
`void SetDigitalPort(DigitalItem[] items)`	Many ports in one command
`bool GetDigitalPort(DigitalPortEnum port)`	Reads the hardware
`bool GetLastDigitalPortSet(DigitalPortEnum port)`	Last known, no traffic
`int GetDigitalPortAsInt(DigitalPortEnum port)`	
`bool Toggle(DigitalPortEnum port)`	
`event OnDigitalPortChanged(EZB ezb, DigitalPortEnum port, bool state)`	Fires on an **actual change** only, however it arrived - a write, a toggle, or a read that came back different. Lets a skill mirror digital I/O without polling

ADC - ezb.ADC

`ADCPortEnum`: `ADC0`..`ADC255`, plus `NA`.

Method	Notes
`int GetADCValue(ADCPortEnum port)`	0-255 (8 bit), relative voltage over 0-5V. 10 and 12 bit firmware readings are scaled down. **Throws** `NotSupportedException` when the firmware has no ADC capability
`int GetADCValue12Bit(ADCPortEnum port)`	0-4095
`float GetADCVoltage(ADCPortEnum port)`	Volts
`float GetADCVoltageFromValue(int adcValue)`	Convert a raw value you already have

PWM - ezb.PWM

`PWMPortEnum`: `D0`..`D99` only (the digital ports that can output PWM).
`PWM.PWM_MIN` is 0, `PWM.PWM_MAX` is 100.

Method	Notes
`void SetPWM(PWMPortEnum port, int dutyCycle)`	Clamped to min/max; `0` calls `StopPWM()`. Does nothing when disconnected. **Throws** when the firmware lacks `CAP_PWM_DUTY`
`void SetPWM(PWMItem[] ports)`	
`int GetPWM(PWMPortEnum port)`	
`void StopPWM(PWMPortEnum port)` / `bool IsPWMStopped(port)`	
`event OnPWMChanged(EZB ezb, PWMPortEnum port, int dutyCycle)`	

---

6. I2C, UART and ping sensors

I2C - ezb.I2C

Method	Notes
`void Write(byte deviceAddress, params byte[] data)`	
`void WriteBinary(byte deviceAddress, byte b7, ... byte b0)`	Eight bits as separate arguments
`byte ReadByte(byte deviceAddress)`	
`byte[] Read(byte deviceAddress, byte expectedBytesReturn)`	
`void SetClockSpeed(uint rate)`	

UART - ezb.Uart

Two families. **Bit-banged serial out of any digital port** (send only):

```
ezb.Uart.SendSerial(Digital.DigitalPortEnum.D5, Uart.BAUD_RATE_ENUM.Baud_9600, "hello");
```

Overloads take `string`, `char[]`, `byte`, or `byte[]`. `SetBaudClock(BAUD_RATE_ENUM, int clockSpeed)`
adjusts the timing.
The hardware UART expansion ports (bidirectional, buffered):

Method	Notes
`void UARTExpansionInit(int port, uint baudRate)`	Call before anything else
`void UARTExpansionWrite(int port, byte[] data)`	
`int UARTExpansionAvailableBytes(int port)`	
`byte[] UARTExpansionRead(int port, int bytesToRead)`	
`byte[] UARTExpansionReadAvailable(int port)`	
`int MaxUARTReceiveBuffer`	Default 4000

HC-SR04 ultrasonic - ezb.HC_SR04

Member	Notes
`int GetValue(Digital.DigitalPortEnum triggerPort, Digital.DigitalPortEnum echoPort)`	
`int MinPoolTimeMS`	Default 200. The sensor cannot be read faster than this
`static readonly int MAX_VALUE`	255

EZB speaker - ezb.Audio

`ARCx.Core.EZBController.Audio`. Streams 8 bit mono PCM to an EZ-B v4 speaker.

Member	Value	Notes
`AUDIO_SAMPLE_BITRATE`	14700	The playback sample rate
`RECOMMENDED_PACKET_SIZE`	256	
`RECOMMENDED_PREBUFFER_SIZE`	20000	The EZ-B buffer is 50k, so never exceed that
`PACKET_SIZE`, `PREBUFFER_SIZE`		Adjustable
`OnAudioDataChanged`, `OnVolumeChanged`, `OnStopPlaying`		Events

`ezb.MusicSynth` makes synthesized music on the same speaker.

---

7. Serial COM ports

`ARCx.Core.Scripting.Common.ComPortManager` (static). A shared registry of open COM ports, so a
skill and a script can share one port and the Script Manager can close them all.

Method	Notes
`string[] AvailablePorts()`	
`void PortOpen(string portName, int baud)` / `(string, int, string description)`	
`bool IsPortOpen(string port)` / `void PortClose(string port)` / `void CloseAllPorts()`	
`void PortWrite(string port, byte data)` / `(string, byte[])` / `(string, string data, bool newLine)` / `PortWriteLine(string, string)`	
`int Available(string port)`	Bytes waiting
`byte PortReadByte(string port)` / `string PortReadChar(string)` / `byte[] PortReadBytes(string, int)` / `string PortReadStr(string, int)` / `string PortReadLine(string)` / `string PortReadAll(string)`	
`void PortClearInputBuffer(string port)`	
`ConcurrentDictionary<string, ComPortDefinitionCls> GetPorts`	The open ports, each with its `SerialPort` and `Description`

A skill that owns a device on a port (a servo bus, a lidar) normally opens its own `SerialPort`
in `Initialize()` and closes it in `DisposeOverride()`. Use `ComPortManager` when the port should
be shared with scripts.
Files have an equivalent registry: `ARCx.Core.Scripting.Common.FileManagerService` (`FileReadLine`, `FileReadAll`, `FileReadByte`, `SetFileReadPosition`, `FileClose`, `FileCloseAll`, and an `OnCollectionChanged` event).

---

8. Video routing

`ARCx.Services.VideoRouting.VideoRouteService`. The meeting point between skills that **produce**
frames (cameras) and skills that **consume** them (vision, tracking, recording).

Producing frames

```
// Initialize()
_sourceGuid = _videoRouteService.AddSource("My Camera", "USB camera on /dev/video0", 30);

// per frame, with an OpenCV Mat
_videoRouteService.NotifySubscribers(this, _sourceGuid, matImage);

// DisposeOverride()
_videoRouteService.RemoveSource(_sourceGuid);
```

Member	Notes

`Guid AddSource(string sourceName, string sourceDescription, double sourceFPS)`	Keep the guid
`void NotifySubscribers(object sender, Guid guid, Mat matImage)`	Call for every frame
`void RemoveSource(Guid guid)`	In `DisposeOverride()`
`void UpdateSourceDetails(Guid guid, string sourceDescription = null, double FPS = 0, string sourceName = null)`	Leave a parameter null/0 to keep it
`VideoRouteSource[] VideoSources`	All registered sources
`VideoRouteSource GetSource(string sourceName)` / `(Guid)`	

`VideoRouteSource` carries `SourceName`, `SourceDescription`, `FPS` (as declared, not measured),
`FrameCount` and `LastUpdated`.
Save the source name in a configuration, never the guid - guids change every session.

Consuming frames

Event	Notes
`OnNewImageBeforeProcess`	The raw frame. **This is where detection goes.** Set `MarkupItems` here and ARCx draws them for you
`OnObjectsDetected`	What the detection skills found on this frame, as `VideoObjectsDetectedEventArg`. **This is where tracking and reacting to detections goes.** See below
`OnNewImageAfterProcessed`	After markup has been drawn. For display or recording. **Do not alter the images here**
`OnVideoSourceAdded` / `OnVideoSourceRemoved`	A source appeared or went away

Each subscriber is invoked on its own, so one skill that throws cannot stop the others from seeing the frame or take the capture thread down.
`VideoRouteSourceEventArg`:

Member	Notes
`OriginalMat`	The original. **Never alter it.** Use it as the detection source
`ProcessedMat`	What you may draw on, from `OnNewImageBeforeProcess` only. When nothing subscribes to that event and no markup is pending, this **is** `OriginalMat`, not a copy
`ComputerVisionResizedMat`	A smaller copy for detection, built lazily on first read. Size comes from the `ARCxCompute`

	`rVisionImageScaledWidthPixels` preference (default 320)
`ComputerVisionResizedScaleFactor`	Multiply a coordinate in the resized mat by this to get the coordinate in `OriginalMat`. It is `1.0` until the resized mat has been read
`MarkupItems`	See below

The images are borrowed. The service disposes the argument after both events and the browser
refresh; subscribers must not dispose it and must **clone anything they need to keep** beyond
the handler.

`VideoSourceMarkupItem` - let ARCx draw the overlay rather than doing it yourself:

Member	Default	Notes
`Text`	-	Empty for no text
`TextPosition`	-	OpenCV `Point`, in `ProcessedMat` pixels. OpenCV places the **bottom-left** of the text here
`TextColor` / `RectangleColor`	-	OpenCV `Scalar`, **BGR order**
`TextSize`	`1`	HersheyPlain font scale
`TextThickness`	`2`	
`Rectangle`	-	OpenCV `Rect`, in `ProcessedMat` pixels - scale detections made on the resized mat first
`RectangleThickness`	`1`	`0` for no rectangle
`FramesToDisplayFor`	`1`	Decremented after each frame is drawn. `1` is right when detecting every frame
`TreatAsObjectDetected`	`false`	Tells a consuming skill this is a detection it may track or follow. Set it on the detector's **primary** detection (i.e. the largest face)
`TrackingType`	`"unknown"`	The kind of detection: `"glyph"`, `"face"`, `"color"`, `"object"`...
`TrackingTypeValue`	-	What was detected within that kind: `"Glyph 1"`, `"red"`, `"person"`. Empty when there is nothing more specific
`Confidence`	`null`	0 to 1, when the detector has such a measure

`Attributes`	`null`	`Dictionary<string, object>` of details only this kind of detection has (a glyph's `Rotation`). Numbers, strings or booleans, PascalCase keys. A detector that knows the distance sets `DistanceM` (`DetectedObject.ATTRIBUTE_DISTANCE_M`) in metres

Following detections

A detection skill only detects and adds markup items. Following the object with servos or the movement panel, scripts that run when something appears, and the variables describing it belong to the **Object Detection Script** robot skill, which works for every detection skill. Do not duplicate them in a detection skill: fill in `TrackingType`, `TrackingTypeValue` and, where you have them, `Confidence` and `Attributes`, and it takes care of the rest.

A skill of your own that reacts to detections subscribes to `OnObjectsDetected`. It is raised for every frame of every source, after `OnNewImageBeforeProcess`, with an **empty list** when nothing was detected so you can tell an object has gone. Every markup item with an area that names its `TrackingType` or is flagged `TreatAsObjectDetected` becomes a `DetectedObject`:

Member	Notes
`TrackingType` / `TrackingTypeValue`	As set on the markup item. The type is `"unknown"` when not set; the value is never null
`Name`	The value, or the type when there is no value
`Rectangle`	In pixels of the original frame (`FrameWidth` x `FrameHeight` on the argument)
`IsPrimary`	`TreatAsObjectDetected`
`Confidence`, `Attributes`, `Label`	As set on the markup item. `Attributes` is never null. On a source a depth camera is aligned with, ARCx adds the measured `DistanceM`, `ForwardM`, `LeftM`, `HeightM`, `RangeM`, `Angle` and `DepthSensor` itself (section 26)

The argument also carries `VideoSource`, `FrameWidth`, `FrameHeight`, `FrameNumber` and `Time` (`Environment.TickCount64`). It holds no images and never changes, so keep it as long as you like. It is raised on the capture thread, so hand servo moves and scripts to a thread of your own.

Enumerating cameras

`ARCx.Core.ComputerVision.VideoDeviceManager.GetDeviceList()` returns the cameras

attached to
this computer with the modes each supports. Store the device **`Identity`** to find it again on reload, and the mode's `VideoDeviceResolution.ID` (which carries FPS, width and height). Enumeration is synchronous and can take a moment. Windows and Linux only; anywhere else it logs
and returns an empty array rather than throwing.

---

9. Audio

`ARCx.Services.AudioServices.AudioService`. Audio in and out, mixed across every skill.

Constant	Value
`AudioService.OUTPUT_SAMPLE_RATE`	22050 - everything you play
`OUTPUT_WAVE_FORMAT`	22050Hz, 16 bit, mono
`AudioService.INPUT_SAMPLE_RATE`	16000 - the microphone, as raised by `OnAudioInput`

Everything you play must match `OUTPUT_WAVE_FORMAT`, or it is noise. The microphone runs at a
different rate from the speaker: 16kHz is what speech recognition and voice activity detection
are built for, so a speech engine can usually take `OnAudioInput` as it is without resampling. Never assume the two rates are the same.

Where it comes out

Do not write a "play on the PC" path and a "play on the EZB" path. There is one output mix,
and where it goes is a setting: this machine's preference in Options, Preferences, overridden by
the project's setting on the EZB Connection robot skill. Queue your audio in the format above and
it plays out of the computer's speakers or is converted, resampled and streamed to an EZB's speaker without your skill knowing which. A skill that offers its own speaker choice is asking the user the same question twice and will disagree with the rest of the project.

Member	Notes
`AudioOutputTargetEnum OutputTarget`	What the project is set to - `PCSpeaker` or `EZBSpeaker`
`int OutputEZBIndex`	The board when `OutputTarget` is `EZBSpeaker`
`bool IsPlayingThroughEZB`	What is actually happening. False while the chosen board is disconnected - audio then plays out of the PC rather than being lost
`string GetOutputDescription()`	For a window or a log entry: "PC speaker", "EZB 1 speaker"
`event Action OnAudioOutputDestinationChanged`	The setting changed
`int OutputVolume`	Master volume, 0 to 200, applied to the mix before either speaker

Read them if you want to *show* the user where the sound is going. Do not branch on them to

decide how to play something.

Playing a finite clip

Register a source, play through it, deregister on dispose:

```
// Initialize()
_audioGuid = _audioService.RegisterInputSource();

// Play
using var stream = ...;
using var waveReader = new WaveFileReader(stream);

var resampled = new WdlResamplingSampleProvider(waveReader.ToSampleProvider(),
AudioService.OUTPUT_SAMPLE_RATE);

_audioService.StopAudio(_audioGuid);
_audioService.AppendPlayAudioForGUIDSource(_audioGuid, resampled);

// DisposeOverride()
_audioService.DeregisterInputSource(_audioGuid);
```

Member	Notes
`Guid RegisterInputSource()` / `void DeregisterInputSource(Guid)`	**Deregister in `DisposeOverride()`**
`void AppendPlayAudioForGUIDSource(Guid, ISampleProvider)`	Reads **all** available audio into a queue. For clips with an end: a file, generated sound, speech. Not for live streams
`int QueueLength(Guid sourceGuid)`	Samples still pending in ARCx. Note that an EZB has its own buffer behind this
`void WaitForSourceToFinish(Guid, int stallTimeoutMs = 3000)`	Blocks until the audio has actually been heard, including an EZB's buffer. Use this rather than polling `QueueLength` yourself. Throws `TimeoutException` when nothing is draining the queue
`void StopAudio(Guid source)` / `void StopAllAudio()`	Also discards what has been sent towards an EZB but not played
`long GetPlayedSamples(Guid sourceGuid)`	How many samples of the source have gone to the speaker since it was registered. Only grows, never counts audio thrown away by `StopAudio()`, and steps one mix buffer (about 90ms) at a time. The sound card or EZB's own buffering comes after it. 0 for an unknown source

Streaming live audio

`AddSourceToMixer(ISampleProvider)` / `RemoveSourceFromMixer(ISampleProvider)`. The mixer
pulls from the provider as needed, in packets, so nothing is read into memory up front. Use this for live streams and for very large files.

Input and devices

Member	Notes
`event Action<float[]> OnAudioInput`	Every frame of input audio: mono floats, -1 to +1, at `INPUT_SAMPLE_RATE`, 1024

	samples (64ms) a frame. This is how a speech engine gets the microphone. **Subscribe only while you want the audio** - the microphone stream is stopped while nothing is subscribed, which is what keeps an idle robot's CPU free
`IDisposable HoldInputOpen()`	Keeps the microphone stream running without subscribing, for something that only reads `InputLevel`, such as a level meter. Dispose it when done
`float InputLevel`	The loudest recent microphone sample, 0 to 1. Only moves while the microphone stream is running - see above
`event Action OnAudioDeviceInitialized`	At start-up, and when an audio device preference changes
`AudioDeviceItem[] GetInputDeviceList()` / `GetOutputDeviceList()`	Only the devices that will open in the format ARCx uses (mono, at `INPUT_SAMPLE_RATE` / `OUTPUT_SAMPLE_RATE`). `Name` includes the audio system (MME, DirectSound, ALSA...) when there is more than one. `Id` is PortAudio's index and changes when devices are plugged in or removed; `Key` does not
`int GetSelectedInputDevice()` / `GetSelectedOutputDevice()`	The PortAudio index of the device in use, else the one that would be tried first, else `-1` when there is none
`string GetInputDevicePreference()` / `GetOutputDevicePreference()`	The chosen device's `Key`, or `AudioService.DEFAULT_DEVICE_KEY` for the system default. May name a device that is not plugged in
`void SetInputDevicePreference(string key)` / `SetOutputDevicePreference(string key)`	Saves the choice and reopens the devices. When the chosen device will not open, ARCx tries the same device through another audio system, then the system default, then anything else, and the Diagnostics page says so

Saving audio files

A WAV is NAudio's `WaveFileWriter`. For MP3, ARCx has its own fully managed encoder, `ARCx.Services.AudioServices.Encoders.Mp3Encoder`. It needs no native library, so the same code works on every platform ARCx runs on; never ship LAME or call the operating system's encoder.

```
using ARCx.Services.AudioServices.Encoders;

// A whole clip: float samples, -1 to +1, interleaved when stereo
byte[] mp3 = Mp3Encoder.Encode(samples, 22050, channels: 1, bitrateKbps: 64);

// Or a stream, written as the audio arrives
using (var encoder = new Mp3Encoder(fileStream, AudioService.INPUT_SAMPLE_RATE, 1)) {

  encoder.Write(frame);          // float samples
  encoder.WritePcm16(pcmBytes);  // or 16 bit little endian PCM, such as a TTS result
```

```
}                                   // Dispose() calls Finish(), which sends the tail

// Anything with an end, resampled when MP3 cannot have its rate
Mp3Encoder.Encode(waveReader.ToSampleProvider(), outputStream);
```

Member	Notes
`SUPPORTED_SAMPLE_RATES` / `IsSupportedSampleRate(int)`	8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000. `OUTPUT_SAMPLE_RATE` and `INPUT_SAMPLE_RATE` are both on the list. Any other rate throws from the constructor; `GetClosestSampleRate()` says what to resample to
`GetSupportedBitrates(int sampleRate)`	32 to 320kbps at 32000Hz and up, 8 to 160kbps below. A bitrate not on the list is rounded to the nearest one, and `BitrateKbps` says which
`GetDefaultBitrate(int sampleRate, int channels)`	What `0` means: 128/192kbps (mono/stereo) at 32000Hz and up, 64/96 at 16000 to 24000Hz, 32/48 below
`Write(ReadOnlySpan<float>)` / `WritePcm16(ReadOnlySpan<byte>)`	May be split anywhere, even between channels or between the two bytes of a sample
`Finish()`	Sends the samples still in the filterbanks and the frames held back for the bit reservoir. Nothing can be written after it. Without it the file is cut short
`SamplesWritten`, `BytesWritten`	Samples per channel in, MP3 bytes out

The result is constant bitrate MPEG-1, 2 or 2.5 layer III (whichever the sample rate belongs to) that every player and browser plays. It is a simple encoder, with no psychoacoustic model and no joint stereo: speech and robot sounds are clean at the default bitrates, but music at a low bitrate sounds worse than from LAME. The decoded audio starts about 1057 samples (a few tens of milliseconds) late and ends with a little silence, because there is no LAME tag to tell players to trim it. An instance is not thread safe.
NAudio types (`ISampleProvider`, `WaveFileReader`, `WdlResamplingSampleProvider`) come from the
copy of NAudio that ARCx ships. Add `NAudio.Core` to the References item group of the skill's
`.csproj` to use them (see rule 16); note that NAudio.Core 3.x
`Read()` takes a `Span<float>`.

---

10. Text to speech

`ARCx.Services.TextToSpeech.TextToSpeechService`.

Speaking

Method	Notes
`void Say(string text)`	Returns immediately
`void SayWait(string text)`	Blocks until the speech has finished being heard

`void StopTalking()`	
`string GetSpeakerDescription()`	Where speech will be heard, in words
`bool IsSsml(string text)`	Whether the text is an SSML document rather than plain text
`string GetTextToSpeechEngineWindowTitle()`	The project's TTS engine skill, or null when there is none and Say will throw
`bool IsSpeaking`	Speech is queued that has not all gone to the speaker yet

There is no PC version and EZB version of these. Speech goes into the output mix like anything
else, and the project's audio output setting decides which speaker it reaches - see 9. Nothing that speaks needs an EZB index, and nothing that speaks should offer the user a choice of speaker.
The project's TTS engine is whichever robot skill implements `TextToSpeechEngineInterface`; if
there is none, the Say methods throw.

Following speech

A robot skill that moves with the voice - a talking mouth, lights that pulse while the robot speaks - subscribes to `OnSpeaking`. It is raised for every Say and SayWait, from any script, robot skill or AI agent, once the speech has been synthesized and queued, and it carries the speech itself, so the skill works from the real sound rather than guessing from the text.

Member	Notes
`event Action<TextToSpeechSpeakingEventArgs> OnSpeaking`	Raised on the thread that called Say, before Say returns. Copy what you need and do the work on your own thread. An exception in a handler is logged and does not stop the speech. Unsubscribe in `DisposeOverride()`
`long GetPlaybackPosition()`	Where the speaker is in the speech audio, in samples at `AudioService.OUTPUT_SAMPLE_RATE`. Steps a mix buffer (about 90ms) at a time while speech plays, so interpolate between steps with your own clock for smooth movement. Stops where the voice stopped when speech is cut short

`TextToSpeechSpeakingEventArgs`: `Text` (as passed to Say, possibly SSML), `IsSsml`, `Samples` (mono floats, -1 to +1, shared by every subscriber - read, do not change), `SampleRate`, `StartPosition` (the `GetPlaybackPosition()` value at which the first sample is heard; in the future when earlier speech is still playing) and `Duration`.
`GetPlaybackPosition() - StartPosition` is how many samples of that speech have been played: negative before it starts, past `Samples.Length` once it has finished. When `IsSpeaking` is false and the position has stopped moving short of the end, the speech was stopped. What is heard trails the position by the sound card's or EZB's own buffering, so give the user an offset to tune. The Talk Servo robot skill is a complete example.

Being the engine

Implement `TextToSpeechEngineInterface` on the worker:

```
public TextToSpeechResultCls OnTextToSpeech(TextToSpeechRequestCls request) {

  // request.SampleRate, request.SampleSize (Bits8 / Bits16), request.Channels (always 1),
  // request.TextToSpeak (plain or SSML - use TextToSpeechService.IsSsml())

  return new TextToSpeechResultCls() {
    Success = true,
    Audio = pcmBytes,       // RAW PCM, no file header, in exactly the format requested
  };
}
```

Called **synchronously on the caller's thread**. Return `Success = false` (or null `Audio`) when
the text could not be synthesized. Honour `request.SampleRate` and `request.SampleSize` exactly;
any other format produces noise. In practice speech is always asked for as 16 bit at `AudioService.OUTPUT_SAMPLE_RATE`, because it goes into the output mix and the mix is converted to
whatever the speaker needs - an engine has nothing to say about where its voice comes out.
Only one TTS engine skill is allowed in a project.

---

11. Speech to text

`ARCx.Services.SpeechToText.SpeechToTextService` routes between skills that **provide** recognition and skills that **consume** it. The service does no recognition itself.
An engine says what it **can do**, not what it is **for**. Whether an engine is the wake word listener, the command listener or the one behind Wait For Speech is decided by whoever wires the
project together. Everything an engine cannot do natively is filled in by the service:

Gap	**What the service does**
Engine cannot hold recognition open (`CanListenContinuously` false)	A `Continuous` request is served by starting it again after every result. The consumer sees a stream of utterances
Engine must not be left listening (`RequiresTrigger` true)	Never starts it for a `Continuous` request. It runs for a `Once` request, for the awake window after its wake word, or after `TriggerWake()` - and otherwise not at all
Engine should wait for a wake word	Holds it asleep while another engine listens for the wake phrase, then runs it for the awake window
Two consumers, one engine	A `Once` request suspends a `Continuous` one and resumes it afterwards. A second `Continuous` request is refused and logged

Those first two are separate, and an engine billed per phrase wants both. `CanListenContinuously`
false alone still leaves the service re-arming it forever, which is exactly how a metered service
burns through a daily allowance on an empty room.
`SpeechToTextModeEnum` is a property of the **request**, not of the engine:

Value	**Meaning**

`Once`	Listen for one utterance and stop. The Wait For Speech popup, `WaitForSpeech()` scripts, a `GetText` control command. Bypasses any wake word - asking is the trigger
`Continuous`	Keep recognizing until the session is cancelled

Providing an engine

Add **one** subclass of `SpeechToTextEngineAbstractEntity` to the skill's assembly (not on the
worker itself). `ProjectToolsService` finds it when the skill is added, creates it with dependency
injection, and assigns it to the worker's `SpeechToTextEngine` property.

Member	**Notes**
`RobotSkillWorker`	Set for you, after construction. Build anything that needs it lazily
`SpeechToTextCapabilitiesCls GetCapabilities()`	Return a fresh instance; it is read every time the service or a config dialog needs it
`SpeechToTextTriggerCls GetTrigger()`	Optional. Return the trigger saved in the skill's configuration, or null
`void OnStartSpeechRecognition(SpeechToTextRequestCls)`	Replace whatever is running. `Mode` is never `Continuous` for an engine that said it cannot
`void OnStopSpeechRecognition()`	Must be safe when nothing is running
`void Dispose()`	The host only disposes the **worker**, so dispose the engine from `DisposeOverride()`

`SpeechToTextCapabilitiesCls`:

Field	**Meaning**
`CanListenContinuously`	False for a record-upload-answer engine. The service re-arms it instead
`RequiresTrigger`	True when the service must never start it on its own. A `Continuous` request leaves it quiet; only a `Once` request, its wake word, or `TriggerWake()` runs it. **Set this on anything billed or rationed per phrase**
`SupportsPhraseGrammar`	False for a dictation or cloud engine that always returns free text
`DetectsSpeechOnly`	True for a detector - a voice activity detector that knows somebody is talking but not what they said. Kept out of the engine lists consumers pick from, offered as a trigger source
`ReturnsConfidence`	False when `Confidence` is a constant, so

	consumers know not to filter on it
`Description`	One line for the user, beside the engine in a drop-down

Report results back through the service. There is no session to track - the service stamps each
message with whichever session the engine is running for:

```
_speechToTextService.UpdateSpeechDetectedMessage(worker, speechResultCls);
_speechToTextService.UpdateSpeechHintMessage(worker, speechHintCls);     // "speak louder", etc
_speechToTextService.UpdateAudioStateMessage(worker, audioStateCls);      // silence / speech / stopped
```

Report **failures** as well as successes (`Success = false`, `ErrorString`). The service closes a
`Once` session, re-arms and puts gated engines to sleep on the strength of these messages, and a
consumer waiting on one utterance would otherwise wait out its whole timeout.

Triggers - the wake word

A trigger belongs to the **gated engine's own robot skill**, so the wiring follows that skill around the project and every consumer of the engine inherits it.

1. Put a `SpeechToTextTriggerCls` on the skill's `Configuration` and save it as usual.
2. Drop `<UCSpeechToTextTrigger @bind-Trigger="_configuration.Trigger" EngineWindowTitle="@(CurrentRobotSkillWorker?.Properties?.WindowTitle)" />` into the skill's config dialog. The component lists only engines whose capabilities report `CanListenContinuously`, because a one-utterance engine cannot listen for a wake phrase all day.
3. Return it from `GetTrigger()`.
4. Call `SpeechToTextService.NotifyTriggerChanged(windowTitle)` from `OnConfigWindowClosed`, so a wake word the user just set takes effect on recognition that is already running rather than at the next project load.

That is the whole feature on the engine's side. The service arms the trigger source, starts the gated engine when the phrase is heard, refreshes the awake window on every result, and stops it when `StayAwakeMs` lapses. `StayAwakeMs` of 0 means one utterance per wake, which is the setting for an engine with a daily allowance. An engine may be its own trigger source, in which case the service swaps the grammar back and forth.

An engine that reports `RequiresTrigger` with **no** usable trigger gets a gate that never wakes by itself: it stays quiet for `Continuous` requests, and `TriggerWake()` still works so a listen button or a `Wake` control command can run it for one utterance.

Waking on any voice. `SpeechToTextTriggerCls.WakeOnAnySpeech` wakes the gated engine as soon as anybody talks, with no phrase to say. The trigger listener then asks its source for dictation rather than a one-phrase grammar, and any successful result wakes the gate.

That is what makes a **voice activity detector** useful. Write one as an ordinary speech engine with `CanListenContinuously = true` and `DetectsSpeechOnly = true`, and signal either

way:

- publish a result with `Success = true` (the text may be empty), or
- publish `UpdateAudioStateMessage` with `AudioStateEnum.Speech`.

Either wakes whatever the detector is the trigger for, so a detector that never produces text at
all still works. The service forces `WakeOnAnySpeech` for a detector source whatever the saved trigger says, since there is nothing for a phrase to match, and `GetSpeechToTextTriggerSources()` is the list that includes detectors - `GetSpeechToTextRobotSkills()` leaves them out, because they have no text to give a consumer.

Consuming recognition

Member	Notes
`SpeechToTextSessionCls StartSpeechRecognition(SpeechToTextRequestCls)`	**Does not throw.** Returns null, having logged why, when the engine is missing or busy. Keep the session
`void CancelSpeechRecognition(SpeechToTextSessionCls)`	Safe with null, and twice
`event OnSpeechDetected(object, SpeechToTextResultCls)`	Compare `e.SessionId` against your session's `Id`. Sender is the engine's worker
`event OnSpeechHint(object, SpeechToTextHintCls)`	Same `SessionId`
`event OnAudioStateChanged(object, SpeechToTextAudioStateCls)`	Same `SessionId`
`event OnTriggered(object, SpeechToTextTriggeredCls)`	A gated engine woke (`Awake` true) or slept. Hang a script off it; the recognition is already being driven
`string[] GetSpeechToTextRobotSkills()`	Window titles of every engine that can produce text - populate a drop-down with these. Detectors are left out
`string[] GetSpeechToTextTriggerSources()`	Window titles of everything that can serve as a wake source: engines that can listen continuously, detectors included
`string[] GetSpeechToTextRobotSkills(bool canListenContinuouslyOnly)`	Pass true for only the engines that can hold recognition open
`bool DoesSpeechToTextRobotSkillExist(string)`	Never throws
`RobotSkillWorkerAbstractEntity GetSpeechToTextRobotSkill(string)`	Throws when not found
`SpeechToTextCapabilitiesCls GetCapabilities(string)`	Null when the engine is not in the project
`SpeechToTextTriggerCls GetTrigger(string)`	What wakes that engine, or null
`bool TriggerWake(string)` / `TriggerSleep(string)`	Wake or sleep a gated engine by hand. False when it has no trigger

`void NotifyTriggerChanged(string)`	Re-read that engine trigger and re-arm everything listening through it. A speech engine skill calls this when its config window closes
`bool IsAwake(string)`	Whether that engine is listening right now

Filter on `SessionId`, never on `OriginWindowTitle` alone. Two consumers can be sharing one
engine, and the window title does not tell you whose speech a result is.
Do not build a wake word state machine in a consumer. Ask for `Continuous` and subscribe to
`OnTriggered` if you want to know when the robot starts and stops listening.
The engine the Wait For Speech popup and script commands use is named in `ProjectToolsService.CurrentProjectProperties.WaitForSpeechEngineWindowTitle`.

---

12. Movement

`ARCx.Services.Movement.MovementManagerService`. The abstraction between "go forward" and the
motors.
Anything can call `Go*()`. The **movement panel** - the one skill in the project implementing `MovementPanelInterface` - subscribes to the events and actually drives the hardware.
`MAX_SPEED` is 255, `MIN_SPEED` is 0.

Member	Notes
`GoForward()` / `(byte speed)` / `(byte speedLeft, byte speedRight)`	
`GoReverse()`, `GoLeft()`, `GoRight()`	Same three overloads each
`GoStop()`	
`GoUp()`, `GoDown()`, `GoRollLeft()`, `GoRollRight()`, `Takeoff()`, `Land()`	Flying robots
`GoEmergency()`	
`GoCustom(string customMovementID)` / `(string, byte speed)` / `(string, byte left, byte right)`	For a movement the enum does not cover
`Go(MovementDirectionEnum direction)`	
`SetSpeed(byte)` / `(byte left, byte right)` / `SetSpeedLeft(byte)` / `SetSpeedRight(byte)`	
`GetSpeed`, `GetSpeedLeft`, `GetSpeedRight`	
`GetCurrentDirection`, `GetLastDirection`	
`GetCustomMovementId`	Read this from an event handler when the direction is `Custom`
`LocomotionStyle`	`Undefined`, `Wheels`, `Tank`, `Drone`, `Gait`, `Helicopter`, `Plane`, `Rocket`, `Submarine`, `Boat`. Set by the movement panel

`MovementDirectionEnum`: `Stop`, `Forward`, `Reverse`, `Left`, `Right`, `Up`, `Down`, `RollRight`, `RollLeft`, `Takeoff`, `Land`, `Emergency`, `Custom`.
Events:

Event	Signature
`OnMovement`	`(MovementDirectionEnum direction)`
`OnMovement2`	`(MovementDirectionEnum direction, byte speedLeft, byte speedRight)` - what a movement panel subscribes to
`OnSpeedChanged`	`(int speedLeft, int speedRight)`

---

13. Kinematics

`ARCx.Services.Kinematics.KinematicsService`. A registry and router: a **provider** skill publishes named chains ("Left Arm", "Camera Gimbal") and does the maths; everything else moves
them by name without knowing who owns them. The service does no kinematics itself.

Consumer side

```
// Configuration: UCKinematicChainSelector, save the name
var result = _kinematicsService.MoveTo(_Configuration.ChainName, x, y, z);

if (!result.Accepted || !result.Reachable)
  UCLogBuffered.AddEntry(this, $"Move refused: {result.Message}");
```

Member	Notes
`KinematicChain[] Chains` / `string[] ChainNames`	Snapshot. Present by **name**, never save the guid
`KinematicChain GetChain(string name)` / `(Guid)` / `bool ChainExists(string)`	Null when unknown
`MoveTo(name, x, y, z, speed = -1, mode = Joint, requestedBy = "")`	Absolute position, provider chooses the orientation
`MoveToPose(name, x, y, z, roll, pitch, yaw, ...)`	Absolute position and orientation
`MoveRelative(name, dx, dy, dz, ...)`	Offset from where it is now
`Move(KinematicMoveRequest request)`	The one path every move takes
`KinematicPose GetPose(string name)`	Runs the provider's forward kinematics on the **live servo positions**, so it reflects moves made by other skills. Throws for an unknown chain
`KinematicPose GetLastKnownPose(string name)`	Cached, no provider call. Null when unknown or never reported. Cheap - use for displays
`double[] GetJointAngles(string name)`	Degrees. Empty array when the provider does not expose them
`bool IsMoving(string name)` / `void Stop(string name)` / `void StopAll()`	

`bool WaitForMoveComplete(string name, int timeoutMs, CancellationToken ct)`	True when idle, false on timeout. `timeoutMs <= 0` waits forever. Polls, because a provider that moves synchronously has already finished
Events	`OnChainAdded`, `OnChainRemoved`, `OnChainUpdated`, `OnPoseChanged`, `OnMoveRequested`, `OnMoveStateChanged`

An unknown chain name throws - that is a configuration error the caller needs to see. A provider that cannot reach the target does **not** throw; check `KinematicMoveResult.Accepted` and `.Reachable`.

`KinematicMoveResult`: `Accepted`, `Reachable`, `Message`, `SolvedPose`, `JointAnglesDegrees`, `ResidualErrorMm`, plus the factories `Ok(...)`, `Rejected(...)`, `Unreachable(...)`.

`KinematicPose`: `X`, `Y`, `Z`, `Roll`, `Pitch`, `Yaw`, `HasOrientation`, `Timestamp`, and `DistanceTo()`, `Offset()`, `Clone()`, `ToPositionArray()`, `ToPoseArray()`.

`KinematicMoveModeEnum` selects `Joint` (default) or linear interpolation.

Provider side

- `AddChain(KinematicChainRegistration)` in `Initialize()` or after your configuration loads.
  Keep the returned `Guid`. Throws on an empty name, a duplicate name (case-insensitive) or a
  missing required handler - catch per chain so one bad name does not sink the others.
- `RemoveChain(guid)` in `DisposeOverride()`. Safe with an unknown or already-removed guid.
- `UpdateChainDetails(guid, name, description, providerName, jointCount, supportsOrientation, supportsLinearMoves)` -
  leave a parameter null (or `-1` for the joint count) to keep it.
- `NotifyPoseChanged(guid, pose)` **whenever the end effector moves**, including every step of an
  interpolated move and when you notice another skill moved your servos. Displays, the published
  variables and `OnPoseChanged` all hang off this.
- `NotifyMoveStateChanged(guid, bool isMoving)` when a move starts and ends, or `WaitForMoveComplete()` never returns.

`KinematicChainRegistration` handlers: `OnMoveRequested` (`Func<KinematicMoveRequest, KinematicMoveResult>`), `OnPoseRequested` (`Func<KinematicPose>`), `OnStopRequested` (`Action`),
`OnIsMovingRequested` (`Func<bool>`), `OnJointAnglesRequested` (`Func<double[]>`). Plus `Name`,
`Description`, `ProviderName`, `JointCount`, `SupportsOrientation`, `SupportsLinearMoves`, `PublishVariables` (default true).

With `PublishVariables`, poses are published as global script variables prefixed `Kinematic` + the chain name with non-alphanumerics removed ("Left Arm" becomes `KinematicLeftArmX`, `KinematicLeftArmY`, ... `KinematicLeftArmIsMoving`).

---

14. Navigation messaging (NMS)

`ARCx.Services.NMS.NMS2Dv1.NMSService`. Routes 2D positioning and scanning data between sensor
skills and a navigator (SLAM) skill.

Three levels: sensors report, the service collates, a navigator consumes. Outdoors a fourth

part joins them: a GPS robot skill reports satellite fixes and ties the NMS map to the earth, so a navigator can work in latitude and longitude (see GPS: tying the map to the earth).

Reporting

Method	From
`void UpdatePose(Pose pose)`	A positioning skill: wheel encoders, a tracking camera. `X` and `Y` are **metres** from where tracking began; `Yaw` is **radians** likewise relative
`void UpdateScan(ScanPoint[] scanPoints)`	A distance sensor: lidar, ultrasonic. **Use the array overload** when reporting many points
`void UpdateScan(ScanPoint scanPoint)`	One point. Slow for a full sweep

`ScanPoint`: `Range` (metres, `0` means nothing detected), `Angle` (radians relative to the robot, -PI to PI: 0 straight ahead, growing counter-clockwise, so PI/2 is left and -PI/2 is right),
`Confidence` (0 = none, 255 = certain).
`Pose`: `X`, `Y` (metres), `Yaw` (radians, -PI to PI), `Confidence`, plus `Frame` and `TimeStamp` which the service assigns.
The NMS is ROS's frame and units (REP 103): metres and radians. `+X` is forward (the way the
robot faced when tracking began), `+Y` is left, and `Yaw` grows counter-clockwise, as the robot **turns left**, from 0 at the start. `ScanPoint.Angle` grows the same way from straight ahead, so a
scan point at range `r` lands on the map at bearing `h = Yaw + Angle`: `X + r * cos(h)`, `Y + r * sin(h)`. A ROS pose or LaserScan maps straight across with no conversion at all. Report
angles normalised to -PI..PI (`NMSAngles.NormalizeRadians()`), as ROS does, and accept any angle
you are given. `NMSAngles.ToRadians()`/`ToDegrees()` convert for display.
The old `Pose.DegreesHeading`, `ScanPoint.Distance` (cm) and `ScanPoint.Degree` remain as
`[Obsolete]` members that convert, so robot skills built before this change keep loading. Do not
use them in new code.
Wheel odometry in this frame, per reading, with each wheel's travel `dL`, `dR` (metres) and the distance between the wheels `b`: the yaw changes by `(dR - dL) / b` radians (positive turning
left), and the robot moves `(dL + dR) / 2` along the yaw half way through that change.
Every robot skill uses this frame for any angle or position around the robot, not only NMS data,
and uses metres and radians for anything a script reads or passes (variables, control command
parameters and results): `Turn(1.5708)` turns a quarter turn left, and an arc "from A to B" runs
counter-clockwise. Settings a person types and readouts on screen stay in degrees (and whatever
distance unit reads naturally), converted at the UI.
Use the same names as the other robot skills for the same things, so a script written for one reads naturally against another:

What	Name
Position variables	`<prefix>PoseX`, `<prefix>PoseY` (metres), `<prefix>PoseYaw` (radians)
Distance driven since the origin	`<prefix>OdometryDistance` (metres)
Read the position	control command `GetPose`, returning `[x, y, yaw]`
Make here the origin	control command `ResetPose`
Drive and turn by an amount	`DriveDistance(metres)`, `Turn(radians)` (positive turns left)
A bearing to something	a name ending in `Angle` (radians), never `Degree` or `Heading`
A sideways offset setting	measured to the left, like ROS's y (e.g. "Left of centre")

Consuming

Member	Notes
`event OnNewPoseAndScanPoints(PoseAndScanPoints)`	Fired on every pose update, carrying that pose and **every scan point collected since the previous one**, sorted by `Angle`
`event OnNavigationStatusChanged(NavigationStatusEnum)`	An obstacle sensor asking the navigator to pause or continue
`void ChangeNavigationStatus(NavigationStatusEnum)`	
`NavigationStatus`	Current status
`X`, `Y`, `Pose`, `Yaw`, `FrameCnt`	Last reported values (metres, radians)
`void Clear()`	Drop the stored scan points
`MAX_PENDING_SCAN_POINTS`	Scan points are only held while something subscribes to `OnNewPoseAndScanPoints`, and only until the next pose. With no positioning skill reporting, the newest 100,000 are kept and older ones dropped

A navigator that works its position out from the scans alone still needs poses to arrive, because
that is what releases the scan points. A distance sensor skill can publish a placeholder pose (the
last known `X`, `Y` and `Yaw`) after each sweep when it notices no other skill is
reporting one - the RPLidar skill does this, and compares `FrameCnt` before and after its own `UpdatePose()` to tell whether anything else is posting.

GPS: tying the map to the earth

The pose is metres from where tracking began, which says nothing about where on the earth that is. A GPS robot skill (BN-880 GPS Compass, for one) ties the two together, so any robot skill can turn a latitude and longitude into a point on the map and back, without knowing which GPS is fitted or how it is wired.

Member	Notes
`void UpdateGeoFix(GeoFix fix)`	A GPS skill reports each fix the receiver gives, including "no fix". It does not move the pose; the positioning skill still reports that with `UpdatePose()`
`event OnNewGeoFix(GeoFix)`	Raised on the reporting skill's thread at the receiver's rate. Return quickly
`GeoFix GeoFix`	The newest fix, a copy, or null. Check `HasFix` and `TimeStamp`
`void SetGeoReference(GeoReference reference)`	Where the local origin (0, 0) is on the earth and which compass direction `+X` points. `null` unties the map
`event OnGeoReferenceChanged(GeoReference)`	The tie was set, moved or removed (null)
`GeoReference GeoReference`, `bool IsGeoReferenced`	The current tie, a copy, or null
`bool TryGeoToLocal(lat, lon, out x, out y)`	Latitude/longitude (degrees) to map metres. False when nothing has tied the map to the earth
`bool TryLocalToGeo(x, y, out lat, out lon)`	Map metres to latitude/longitude

`GeoFix` is the receiver's own report, like ROS's `sensor_msgs/NavSatFix`: `Latitude`, `Longitude` (degrees, WGS84), `Altitude` (metres above sea level), `FixType` (`NoFix`, `DeadReckoning`, `Fix2D`, `Fix3D`, `Differential`, `RtkFloat`, `RtkFixed`, in that order so `>=` compares quality), `Satellites`, `Hdop`, `HorizontalAccuracy` (metres, one sigma), `Speed` (m/s), `Course` (degrees clockwise from true north, the direction of travel), `UtcTime` and `Source`. A number the receiver did not report is `NaN`, never 0. `HorizontalAccuracyOrEstimate` falls back to HDOP when the receiver gives no error estimate.

`GeoReference` holds `Latitude`, `Longitude` and `Altitude` of the origin and `Bearing`, the compass bearing of `+X` (degrees clockwise from true north): 90 is ROS's east-north-up, 0 is north up. `+Y` is always a quarter turn counter-clockwise from `+X`, so the frame stays right handed. It converts with `ToLocal(lat, lon)`, `ToGeo(x, y)`, `YawToBearing(yaw)` and `BearingToYaw(bearing)`. The local frame is a flat plane touching the earth at the origin: exact there and good to a few centimetres within a kilometre or two. `GeoMath` has the earth geometry (WGS84 geodetic/ECEF/east-north-up conversions, `Distance`, `Bearing`, `Destination`).

The tie can move while the project runs. When the GPS skill is also the pose source it only changes with the origin, but when wheel encoders or a tracking camera report the pose, their frame drifts and the GPS skill keeps sliding the reference so the drifting frame still lands in the right place on the earth (ROS's map to odom correction). So convert at the moment of use, and do not cache converted points. A navigator that keeps places should store them as latitude and longitude, and convert each pose into a flat frame of its own (`GeoMath`, or a frame around a fixed origin) rather than planning in the NMS frame, which is what the Outdoor Navigator robot skill does.

The Robot Bus publishes both as `/arcx/nms/fix` (`arcx/NavSatFix`) and `/arcx/nms/geo_reference` (`arcx/GeoReference`).

A GPS skill, reporting fixes and tying the map to the earth:

```
// For every epoch the receiver sends, fix or not, so subscribers can tell the position went
stale
_nmsService.UpdateGeoFix(new GeoFix() {
  Latitude = lat,
  Longitude = lon,
  Altitude = alt,
  FixType = GeoFixTypeEnum.Fix3D,
  Satellites = sats,
  Hdop = hdop,
  HorizontalAccuracy = accuracyMetres,   // NaN when the receiver does not say
  Speed = metresPerSecond,
  Course = degreesFromTrueNorth,
  Source = Properties.WindowTitle,
});

// When this skill also reports the pose: the origin of its frame and which way +X points.
// Set it once the first good fix fixes the origin, and again only when the origin moves.
_nmsService.SetGeoReference(new GeoReference() {
  Latitude = originLat,
  Longitude = originLon,
  Altitude = originAlt,
  Bearing = 0,                           // +X north (north up); 90 for ROS east-north-up
  Source = Properties.WindowTitle,
});

// DisposeOverride(): untie the map only if this skill tied it
if (_nmsService.GeoReference?.Source == Properties.WindowTitle)
  _nmsService.SetGeoReference(null);
```

A navigator, reading where the robot is on the earth:

```
void nmsService_OnNewPoseAndScanPoints(PoseAndScanPoints frame) {

  var reference = _nmsService.GeoReference;

  // No GPS in the project, or no fix yet
  if (reference == null)
    return;

  var (lat, lon) = reference.ToGeo(frame.Pose.X, frame.Pose.Y);
  double heading = reference.YawToBearing(frame.Pose.Yaw);   // degrees clockwise from true
north

  // A scan point at angle a from the robot is at angle a from the heading in any flat frame,
so
  // sweeps can be placed in the navigator's own frame directly from the robot's position
there
}

// How good is the position right now?
var fix = _nmsService.GeoFix;
double errorMetres = fix?.HasFix == true ? fix.HorizontalAccuracyOrEstimate : double.NaN;
```

Rules for GPS skills:

- **One geo reference.** Only the skill that knows how the NMS frame sits on the earth sets it: the GPS skill. Several GPS skills in one project fight over it, as two pose sources fight over the pose.
- **The pose and the tie agree.** A GPS skill that reports the pose sets a fixed reference for its own frame. A GPS skill that leaves the pose to another skill (wheel encoders, a tracking camera) works out where that skill's frame is and keeps the reference following it; it never reports a pose of its own.
- **Report fixes as the receiver gives them.** `UpdateGeoFix` is the raw report, not a filtered position; the fused position travels as the pose.
- **Use `NaN`, not 0,** for anything the receiver did not report. The Robot Bus sends it as `null`.

---

15. Auto Position

`ARCx.Core.AutoPosition.AutoPositionService`. The engine behind the Auto Position skill: named
frames (servo poses) combined into actions (sequences).

Member	Notes
`AutoPositionConfig Config`	The frames and actions
`string Name`	Unique name for this instance
`void SetVelocity(byte value)`	0 slow to 255 fast, for frames with `UseVariableVelocity`. Takes effect on the next step, so it can change mid-action
`bool IsRunning`	
`ExecAction(...)` / `MoveToFrame(...)`	Movements run on a background thread; **only one runs at a time** and starting a new one cancels the current
`event OnStartAction(sender, AutoPositionAction)`	Raised on the worker thread
`event OnStartFrame(sender, AutoPositionFrame, AutoPositionActionFrame)`	Each frame, including pauses and repeats
`event OnComplete(sender)`	When the queue drains, whether it finished or was stopped

Derives from `DisposableBase` - dispose it when the skill closes, which stops any movement and
unhooks it from the `EZBManager` connection events.

---

16. Scripting, control commands and variables

Control commands: exposing your skill to scripts

Declare them, then handle them.

```
readonly string cmd_setCounter = "SetCounter";

public override List<ControlCommandItem> GetSupportedControlCommands() {

  var items = new List<ControlCommandItem>();

  items.Add(
    new ControlCommandItem() {
      CommandName = cmd_setCounter,
      Description = "Set the counter to the specified value",
      ReturnType = typeof(int),
      Parameters = [

ControlCommandParameter.AddParameter(ControlCommandParameter.ParameterTypeEnum.Number,
"value"),
      ],
    });

  return items;
}

public override object ControlCommand(ControlCommandArgs args) {
```

```
  if (args.Command.Equals(cmd_setCounter, StringComparison.InvariantCultureIgnoreCase)) {

    // The user's script decides what is actually passed, so always verify
    if (args.Parameters.Length != 1)
      throw new Exception($"The {cmd_setCounter} control command expects 1 parameter
(value)");

    _Cntr = Convert.ToInt32(args.Parameters[0]);

    RefreshUI();

    return _Cntr;
  }

  // The base raises the "no matching control command" exception for the calling script
  return base.ControlCommand(args);
}
```

`ControlCommandItem`:

Member	Notes
`CommandName`	`ScriptStart`, `Enable`, `SetCounter`
`Description`	What the command means and when to use it. It no longer has to spell out the return value
`ReturnType`	A real `System.Type`: `typeof(bool)`, `typeof(int)`, `typeof(string[])`. `typeof(void)` says explicitly that nothing useful comes back. **Leave null only when the return type genuinely is not predictable** - do not declare one that is true only some of the time. Shown in the script editor's command list, given to Athena, and used in the generated documentation
`Parameters`	`List<ControlCommandParameter>`. Values arrive in `args.Parameters` in the same order
`ReturnTypeName` / `ReturnTypeDescription`	Read-only. `"Integer"` / `"Integer [-2,147,483,648 to 2,147,483,647]"`

`ControlCommandParameter.AddParameter(ParameterTypeEnum, string name)` and `AddDefaultValue(ParameterTypeEnum, object defaultValue)`.
`ControlCommandArgs`: `Command`, `Parameters` (`object[]`), `CancellationToken` - observe the token in a long-running command.

Running scripts

Create executors through `ScriptManagerService` rather than `new Executor()`, so the Script
Manager UI can list, monitor and stop them, and so they are disposed when the project is cleared.

```
// Initialize()
_executor = _scriptManagerService.CreateExecutor($"{Properties.WindowTitle} script");

// Read back CompilerName - a duplicate name gets a number appended
_executor.EnableConsoleDebugLogging();
```

```
// Run
_executor.StartScriptASync(_Configuration.OnDetectedScript);

// DisposeOverride()
_scriptManagerService.RemoveAndDisposeExecutor(_executor);
```

`Executor` member	Notes
`StartScriptASync(ScriptDefinition)` / `(CancellationToken, ScriptDefinition)`	Background, on the engine's own worker thread. Stops whatever the executor was running or had queued first. Stopped by `StopScript()` or by the script calling `halt()`. **Use the token overload when starting on behalf of another script** (i.e. from `ControlCommand()`), so stopping the caller stops this
`RunScriptAsync(ScriptDefinition)` / `(CancellationToken, ScriptDefinition)`	Same as `StartScriptASync`, but returns a `Task<ScriptRunOutcomeEnum>` that completes when the run ends (`Completed`, `Error`, `Cancelled`, `Skipped`). Never faults; errors still arrive through `OnError`. Holds no thread while awaited
`StartScriptBlocking(ScriptDefinition)` / `(CancellationToken, ScriptDefinition)`	Runs on the calling thread. Does nothing if the executor is already running a script - it never stops or waits for one
`RunScriptBlocking(CancellationToken, ScriptDefinition)`	Same as `StartScriptBlocking`, but returns a `ScriptRunOutcomeEnum`; `Busy` when it was refused
`ExecuteScriptSingleLine(ScriptDefinition)` / `(CancellationToken, ScriptDefinition)`	Returns the result as a string. Refused (raising `OnError`) while the same engine is running a script
`ExecuteScriptSingleLineAsync(ScriptDefinition)` / `(CancellationToken, ScriptDefinition)`	Same, on a thread-pool thread - for Blazor components
`StopScript()`	Stops the running script and any script queued to start
`IsScriptRunning`, `IsCancelRequested`, `Script`, `CompilerName`, `CompilerUniqueIdentifier`	
`EnableConsoleDebugLogging()`	Routes output and errors to the ARCx log automatically
Events	`OnStart(compilerName)`, `OnDone(compilerName, TimeSpan)`, `OnResult(compilerName, text)` (the `print()` output), `OnError(compilerName, message)`, `OnCombinedLog(compilerName, EventTypeEnum, text)`

`ScriptManagerService`: `CreateExecutor(string compilerName)`,

`RemoveAndDisposeExecutor(Executor)` (safe with null or an already-removed executor), `Executor[] GetExecutors`, events `OnAdded` / `OnRemoved`.
`ScriptDefinition` (`ARCx.Config.Helpers`) holds `BlocklyXML`, `Javascript` and `Python` slots -
only one at a time; the editors clear the others when the user switches language. `IsEmpty`, `GetScriptEditType` (checked Blockly, then JavaScript, then Python), `Title`, `Description`.

Global variables

`ARCx.Core.Scripting.Common.VariableManager` (static). One program-wide set of named values
shared by robot skills, JavaScript, Python, Blockly, Custom UI screens and the AI agent. It is how
a skill publishes a reading that anything else can use without a control command: `FrontPingDistance`, `KinematicLeftArmX`, and so on.
Names are plain (`Direction`, not `$Direction`) and case-insensitive. A leading `$` - the old EZ-Script convention - is accepted and removed, so `"$Direction"` still finds `Direction`, but do
not use it in new names. `NormalizeName(name)` gives the canonical form.
Values are stored as the raw object. Numbers, strings and booleans (real `bool`, not 1/0) are
stored as they are; an enum is stored as its name. Arrays, lists and dictionaries are stored by
reference - scripts use `setVarObject()` to store a copy.

Member	Notes
`void SetVariable(string name, object value, string description)`	**Give it a description.** It is shown wherever variables are listed and given to the AI assistants. Null or empty keeps the existing description, so it is *sticky* - describe the variable once and the thousands of value updates that follow (`SetVariable(name, value)`) will not erase it
`object GetVariable(string name)`	Throws `KeyNotFoundException` when it does not exist
`object GetVariable(string name, object defaultValue)`	`defaultValue` when it does not exist
`T GetVariable<T>(string name, T defaultValue)`	Converted to `T` (numbers, numeric strings, booleans, enums; invariant culture). `defaultValue` when missing or not convertible
`bool TryGetVariable(string name, out object value)`	
`bool DoesVariableExist(string name)`	False for an invalid name, never throws
`GlobalVariable GetEntry(string name)` / `TryGetEntry`	The whole entry, or null: `Name`, `Value`, `Type` (`VariableType`: Empty, Boolean, Number, String, Object), `Description`, `UpdatedUtc`. Immutable - a snapshot
`string GetDescription(string name)`	Empty when none or missing
`GlobalVariable[] GetVariables(string`	Snapshot sorted by name, optionally filtered

`namePrefix = null)`	by prefix
`IReadOnlyDictionary<string, GlobalVariable> Variables`	Live view, case-insensitive keys. Read only - change variables through the methods so events are raised
`bool ClearVariable(string name)`	Deletes one; false when it did not exist
`void ClearVariables()`	Deletes all. ARCx does this when a project closes or loads
`string FormatValue(object value)`	Display text used everywhere variables are shown: invariant numbers, `true`/`false`, JSON for collections. `GlobalVariable.ToString()` returns this
`event OnVariableChanged(VariableChangedEventArgs e)`	Every set, delete and clear. `e.Kind` (`Set`, `Deleted`, `Cleared`), `e.Name` (null for `Cleared`), `e.Value`, `e.Variable`, and `e.Affects(name)` - true when the change is about that name or everything was cleared

`OnVariableChanged` is raised on the thread that made the change - often a script or another skill's worker thread - on every set, even when the value is the same. Filter first and keep the
handler cheap; marshal to the UI with `InvokeAsync`. An exception thrown by a handler is logged and
does not reach the code that set the variable.

```
// When the skill starts: describe it once
VariableManager.SetVariable("FrontPingDistance", 0, "Distance in cm from the front ping
sensor");

// Every reading: the description is kept
VariableManager.SetVariable("FrontPingDistance", cm);

// Reading one another skill publishes, with a fallback
var tracking = VariableManager.GetVariable("CameraIsTracking", false);

// Watching one
VariableManager.OnVariableChanged += e => {

  if (!e.Affects("FrontPingDistance"))
    return;

  var cm = e.Value;   // null when it was deleted or cleared
};
```

Scripts see the same variables: `getVar("Name")`, `getVar("Name", default)`, `setVar("Name", value, "optional description")`, `setVarObject(...)` (stores a copy), `varExists("Name")` and
`deleteVar("Name")`, identically in JavaScript and Python. In a Blockly workspace, a variable whose name starts with `$` is a global variable (the `$` is removed in the generated code); other
Blockly variables are local to the script. Custom UI labels show one with `{Name}`.

MCP tools and resources: exposing your skill to AI clients

ARCx is a Model Context Protocol (MCP) server (`ARCx.Services.Mcp.McpServerService`, at `/mcp`). AI clients the user connects - Claude, Cursor, VS Code Copilot, ChatGPT, agent frameworks - see the project through it. **Every control command is already offered to them**

through the generic `send_control_command` tool, with the description and parameters you declared,
and the configuration is readable and editable through the schema of section 19. Nothing is needed
for a skill to work with an AI.
Override `GetMcpTools()` when a command deserves a first class tool: named and typed arguments
instead of a positional list, a longer description, or a result that is not a string - an image above all, which lets the AI see what the robot sees. Override `GetMcpResources()` for data the AI
reads by URI (a latest reading, a status document, an image).

```
using ARCx.Services.Mcp;

public override List<McpToolItem> GetMcpTools() {

  return [
    new McpToolItem() {
      Name = "look",
      Description = "The next camera frame as an image you can look at. Nothing is saved.",
      ReadOnly = true,
      InputSchemaJson = "{\"type\":\"object\",\"properties\":{\"maxWidth\":{\"type\":
\"integer\",\"description\":\"Widest, in pixels. Defaults to 640.\"}},
\"additionalProperties\":false}",
      Handler = args => Task.Run(() => {

        using var mat = waitForFrame(args.CancellationToken);

        return McpToolResult.Image(encodeJpeg(mat, args.GetInt("maxWidth", 640)), "image/
jpeg", $"{mat.Width}x{mat.Height}");
      }),
    },
    new McpToolItem() {
      Name = "set_counter",
      Description = "Set the counter and optionally start or stop its timer.",
      InputSchemaJson = McpToolItem.InputSchemaFromType(typeof(SetCounterArguments)),
      Handler = args => {

        var a = args.GetArguments<SetCounterArguments>();

        _Cntr = a.Value;

        return Task.FromResult(McpToolResult.Json(new { counter = _Cntr }));
      },
    },
  ];
}

public override List<McpResourceItem> GetMcpResources() {

  return [
    new McpResourceItem() {
      Uri = "status",
      Name = "Counter status",
      MimeType = "application/json",
      Reader = ct => Task.FromResult(McpResourceContent.Json(new { counter = _Cntr })),
    },
  ];
}
```

`McpToolItem`:

Member	Notes
`Name`	snake_case, letters, digits and underscores. ARCx prefixes it with the window title as a slug, so the AI sees

	`camera_snapshot__look`. Keep it stable once published
`Title`	Optional, for people. The AI reads `Description`
`Description`	Written for the model: what it does, what the arguments mean, units, pitfalls, what comes back
`InputSchemaJson`	JSON Schema of the arguments object. Defaults to no arguments. `McpToolItem.InputSchemaFromType(typeof(T))` generates it from a class, with the `///` comments as descriptions, exactly like the configuration schema; read it back with `args.GetArguments<T>()`
`ReadOnly`	True when the tool changes nothing. Clients run read only tools without asking the user, and ARCx offers them even when the user has turned **Allow the AI to control the robot** off. Leave false for anything that moves, speaks, saves or changes a setting
`Destructive`	True when it cannot be undone (deleting files). A servo move is not destructive
`Handler`	`Func<McpToolCallArgs, Task<McpToolResult>>`. Runs on a thread pool thread and may block. Throwing returns the message to the AI as an error result it can read and act on

`McpToolCallArgs`: `Arguments` (`JsonElement`), `CancellationToken` (the client gave up, the 120 second call timeout elapsed, or the skill is being removed), `ToolName`, and the tolerant getters `GetString`, `RequireString`, `GetInt`, `GetDouble`, `GetBool`, `GetArray`, `Has`, `GetArguments<T>()`.

`McpToolResult`: `Text(string)`, `Json(object)` (text and structured content; anonymous types work), `Image(byte[] jpegOrPng, mimeType, caption)`, `Error(message)` (the AI reads it and tries something else, which beats an exception for anything the model can fix), `.WithText(...)` to add a caption. `McpContent.FromText/FromImage/FromAudio` for anything else.

`McpResourceItem`: `Uri` (relative; ARCx prefixes `arcx://skill/<slug>/`), `Name`, `Description`, `MimeType`, `Reader` returning `McpResourceContent.Text`, `.Json(object)` or `.Blob(bytes, mimeType)`.

Rules:

- Both overrides are called whenever a client lists tools or calls one, so keep them quick and return the same tools each time. When what you offer changes at run time (a tool per trained object, say), call `NotifyMcpChanged()` so connected clients list again.
- Never put an API key or anything secret in a tool result or a resource; it goes to a model.
- Keep results small: they are tokens in somebody's conversation. Scale images down

(640 pixels
wide is plenty), cap lists.
- A tool that calls a paid model or moves the robot is not `ReadOnly`, however harmless it seems.

`McpCoreTools.GrabFrameJpegAsync(videoRouteService, sourceName, maxWidth, processed, ct)` (a
static on the DI singleton `ARCx.Services.Mcp.McpCoreTools`) returns the next frame of a video
source as JPEG bytes and is what `get_camera_image` uses; a skill that wants the same thing can
call it rather than subscribing to the video events itself.
The user controls all of this in Options → Preferences → AI connections: the server on or off, an
access token, and whether a connected AI may control the robot and change the project. See
`Manuals\mcp.md` for the user's side.

---

17. Talking to other robot skills

`ARCx.Services.ProjectTools.ProjectToolsService`.
Look up another robot skill only in `OnProjectLoadCompleted()`. In `Initialize()` or `OnSetConfiguration()` the other skills may not be loaded yet.

Member	Notes
`RobotSkillWorkerAbstractEntity[] GetRobotSkills()`	A snapshot, safe to enumerate from a background thread while skills are added or removed
`FindRobotSkillByWindowTitle(string)`	**Null-check it** - the user may have renamed or removed the skill your configuration points at
`FindRobotSkillByInstanceGuid(string)`	An instance guid survives a rename, so it is the better thing to remember
`object SendControlCommand(string windowTitle, string command, params object[] parameters)`	Throws when no skill matches, or when the target throws. Case-insensitive title
`object SendControlCommand(CancellationToken ct, string windowTitle, string command, params object[] parameters)`	Pass your token for long-running commands
`List<ControlCommandCollection> GetAllControlCommands()`	Every command of every skill, grouped by skill - the same list the script editor shows
`Task<RobotSkillWorkerAbstractEntity> AddRobotSkillAsync(string storeGuid)`	Installs from the store if needed, then adds it with a unique window title and raises its `OnSetConfiguration()` and `OnProjectLoadCompleted()`. Find a store guid with `PluginManagerService.GetAllRobotSkills()`
`void RemoveRobotSkill(string windowTitle)`	Throws when no skill has that title
`event OnRobotSkillRemoved(string)` /	

`event OnWorkspaceChanged`	

Reading and writing another skill's configuration programmatically (this is how Athena does it):

Member	Notes
`string GetRobotSkillConfigJson(string windowTitle)`	
`string GetRobotSkillConfigSchemaJson(string windowTitle, bool indented = false)`	The annotated JSON Schema
`string PatchRobotSkillConfigJson(string windowTitle, string patchJson)`	Merge a partial change
`void SetRobotSkillConfigJson(string windowTitle, string configJson)`	Replace wholesale

---

18. The project

`ProjectToolsService.CurrentProjectProperties` (static) is `ProjectProperties`.

Member	Notes
`MaxServoPositions`	Default 180. This is what `Servo.SERVO_MAX` resolves to
`ServoPortDescriptions`, `DigitalPortDescriptions`, `ADCPortDescriptions`, `PWMPortDescriptions`	`Dictionary<string, string>` keyed by `"board.PORT"`. The port pickers read and write these, and `RobotConfigService` fills them from the robot picture
`WorkspaceTitles`	The desktops, default `["Main"]`
`StartupScript`, `StartupScriptTimeoutSeconds`	
`WaitForSpeechEngineWindowTitle`	The project's Single speech to text engine, used by the Wait For Speech popup and script commands
`CustomUIs`, `CustomUIExitPassword`	The user's Custom UI screens
`RobotConfig`	The robot picture and its port markers
`Description`, `Notes`, `TagsList`, `TitleImage`, `YouTubeURL`, `ManualURL`, `IsPublic`	Project metadata
`CreatedDateTime`, `LastModifiedDateTime`	

`ARCx.Core.Constants` holds the well-known folders: `DEFAULT_FOLDER` (`My Documents\ARCx Data`, or `~/ARCx Data` on Linux and macOS), `DEFAULT_APPLICATION_FOLDER`, `DEFAULT_PROJECT_FOLDER`, `ROBOT_SKILL_PROJECT_FOLDER`, `PLUGINS_FOLDER`, `SERVO_PROFILES_FOLDER`, `PYTHON_MODULES_FOLDER`, `DEFAULT_USER_LOGS_FOLDER`,
and the file extensions (`.arcx`, `.AutoPositionX`, `.servoX`, `.blocklyX`).
An `.arcx` project file is a zip:

```
projectProperties.json
robotskills/<WindowTitle>/properties.json    RobotSkillProperties
robotskills/<WindowTitle>/config.json        whatever GetConfigOnProjectSave wrote
```

A skill whose `GetConfigOnProjectSave` throws aborts the whole save, so catch what you can and
log it. `GetConfigOnProjectSave` is also called at any time by the AI agent to inspect the configuration, so keep it cheap and free of side effects.

---

19. Configuration and the JSON schema

Save and load with whatever format you like - the stream is yours - but **JSON is the convention**:

```
public override void OnSetConfiguration(OnSetConfigurationParams configurationParams) {

  if (configurationParams.IncludesConfig)
    _Configuration =
JsonSerializer.Deserialize<Configuration>(configurationParams.ConfigStream)!;
}

public override void GetConfigOnProjectSave(Stream configStream, JsonSerializerOptions
suggestedJSONSerializerOptions) {

  JsonSerializer.Serialize(configStream, _Configuration, suggestedJSONSerializerOptions);
}

public override Type GetConfigurationType() {

  return typeof(Configuration);
}
```

`GetConfigurationType()` is what lets the **Athena AI agent** (and any other programmatic editor)
read and change the skill's settings - "make my robot track faces" works because the agent can
see what each setting means. `ConfigSchemaBuilder` builds an annotated JSON Schema from the type:

- Descriptions come from, in order: a `[Description]` attribute, a `[RobotSkillReference]`
  attribute, or **the `///` summary comment on the property**. That is why the template project
  builds with `GenerateDocumentationFile` on: the XML ships beside the DLL and the schema reads
  it. **Write a `///` summary on every configuration property, for someone who has never seen
  the skill.**
- `[Range]` becomes `minimum` / `maximum`.
- Integer-backed enums list their names compactly (`"D0..D255=0..255, NA=512"`), because the JSON
  holds the number and the number alone means nothing.
- Byte arrays and get-only properties are marked so an editor leaves them alone.

`[RobotSkillReference("kind")]` (`ARCx.Config.Schema`) marks a string property that holds
another robot skill's window title:

```
/// <summary>
/// The camera robot skill this skill takes its video frames from.
/// </summary>
[RobotSkillReference("camera source")]
```

```
public string SourceName { get; set; } = string.Empty;
```

It adds `"x-arcx-reference": "robotSkillWindowTitle"` to the schema, so an agent knows the value
must match an existing skill rather than being free text. The constructor without arguments means
any robot skill.
`ConfigSchemaBuilder.Build(Type)` returns a `JsonNode`; `BuildJson(Type, bool indented = false)`
returns a string. Schemas are cached per type. A skill can call it on any type.
Return `null` from `GetConfigurationType()` to opt out of programmatic editing.

---

20. Logging

Two destinations, for two audiences.

ARCLogService - errors and diagnostics

`ARCx.Core.ARCLog.ARCLogService` (static). Goes to the console, the Log Monitor, and the user's
usage log file on disk.

```
try {

  doTheWork();
} catch (Exception ex) {

  ARCLogService.AddLogEntry(this, LogEntryCls.SeverityEnum.Error,
MethodBase.GetCurrentMethod(), ex);
}
```

Overloads:

```
AddLogEntry(RobotSkillWorkerAbstractEntity origin, SeverityEnum type, MethodBase m, Exception
ex = null)
AddLogEntry(RobotSkillWorkerAbstractEntity origin, SeverityEnum type, MethodBase m, string
message, Exception ex = null)
AddLogEntry(object origin, SeverityEnum type, MethodBase m, Exception ex = null)
AddLogEntry(object origin, SeverityEnum type, MethodBase m, string message, Exception ex =
null)
AddLogEntry(string origin, SeverityEnum type, MethodBase m, string message, Exception ex =
null)
AddLogEntry(EZB origin, SeverityEnum type, MethodBase m, string message, Exception ex = null)
```

`SeverityEnum`: `Info` (0), `Warning` (1), `Error` (2).
Passing the worker as `origin` records the skill's type, the method name and the skill's **window
title**, so the user can tell which instance logged it.
`AddLogEntry()` never blocks and never throws - entries are queued and written on a thread pool
task, so they may appear slightly after the call. **Do not log from a tight loop**; every entry is written to disk.

UCLogBuffered - what the user reads

```
UCLogBuffered.AddEntry(this, "Connected to COM3 at 115200");
```

See the UI manual, section 8. A caught exception goes to `ARCLogService`; a status line the user
wants to see goes to `UCLogBuffered`. Most user-visible actions should leave a line there.

---

21. Persistence and runtime state

Three stores, for three lifetimes.

Store	Lives	Use it for
The skill's `Configuration` class	In the user's project file	Anything the user sets. **This is the default**
`CustomRegistryManagerService`	On disk, across restarts, per signed-in account	A skill's own settings that are not part of a project: an API key, a remembered device
`RuntimeStateService`	In memory, for the life of the ARCx process	Values shared between skills for one session

CustomRegistryManagerService

Namespaced key/value, encrypted on disk with the signed-in account's email.

```
_customRegistry.SetRegistryEntry("MySkill", "ApiKey", key);
var key = _customRegistry.GetRegistryEntry("MySkill", "ApiKey", string.Empty);
```

`SetRegistryEntry(ns, name, value)` and `GetRegistryEntry(ns, name, defaultValue)` overload for
`bool`, `int`, `long`, `byte`, `decimal`, `double`, `DateTime` and `string` (`GetRegistryEntryAsLong` for `long`). `DeleteRegistryEntry(ns, name)`.

RuntimeStateService

```
_runtimeState.Set("mySkill.lastScan", scanPoints);

if (_runtimeState.ContainsKey("mySkill.lastScan"))
  var points = _runtimeState.Get<ScanPoint[]>("mySkill.lastScan");
```

`Get<T>` **throws** **`KeyNotFoundException`** when the key is missing *and* when it exists but is not
a `T`. Check `ContainsKey()` first. Keys are case-sensitive. `Remove(key)` returns whether anything was removed.

ConfigRegistryManagerService

ARCx's own preferences (log intervals, font sizes, audio devices, computer-vision scale width,
`RobotSkillUIQueuedUpdateMS`). A skill reads these to respect the user's preferences; it does not
store its own settings here.

---

22. Custom UI and robot configuration services

CustomUIService

`ARCx.Services.CustomUI`. The screens the user lays out for their robot, which one is showing, and
the breadcrumb behind it. The screens themselves live in the project properties, so they save and
load with the project.

Member	Notes
`const string WINDOW_TITLE`	`"Custom UI"` - the pseudo robot skill scripts address with `ControlCommand("Custom UI", "Show", "My screen")`
`List<CustomUIScreen> Screens`	In display order, never null

`CustomUIScreen GetScreen(string guid)` / `FindScreen(string titleOrGuid)`	
`event OnScreensChanged`	Screens added, removed, renamed, reordered or edited
`event OnNavigated(CustomUINavigateEventArgs)`	`Screen` and `Breadcrumb` (home first, current last)

Navigation is one shared state, so a script, a control command and a phone all agree on which
screen is up.
A skill contributes widgets through `GetCustomUIWidgets()` on the worker - see the UI manual,
section 16.

RobotConfigService

`ARCx.Services.RobotConfig`. The picture of the robot and the ports marked on it.
It copies each marker's name into the project's port description dictionaries, so every port selector shows "V3 (Left elbow)" without knowing the picture exists, and it **pushes the servo
hard limits down to the controllers**, which clamp every position in `EZB.Servo.SetServoPosition`.
No skill, script or Auto Position frame can drive a servo past what the robot's body allows. The
limits are re-pushed whenever a controller connects.

Member	Notes
`RobotConfiguration Configuration`	Never null
`List<RobotPortMarker> Ports`	Every marked port, in the order added
`List<RobotPortMarker> GetPorts(RobotPortTypeEnum)`	All the servos, say
`RobotPortMarker FindPort(RobotPortTypeEnum, int boardIndex, string port)`	Null when not on the picture
`event OnChanged`	A port added, removed, moved, renamed, relimited, or the picture replaced

Limits set by a script or a robot skill directly are not tracked here and are left alone.

---

23. Utilities

ONNX Runtime

ARCx ships Microsoft ONNX Runtime (the CPU build) for skills that run neural networks - image
embeddings, detection, speech, voices. ARCx itself does not use it; it is there so every skill shares
one copy. Reference `Microsoft.ML.OnnxRuntime.dll` from the ARCx folder with `Private=false` (the
line is in the template's References comment). Add no `Microsoft.ML.OnnxRuntime*` NuGet package
and copy no `onnxruntime` native library: the one beside ARCx.exe is already the right one for the

platform.
Create sessions through `OnnxAccelerationService`, not with `new InferenceSession` directly. The
user chooses the hardware once, for every skill, in Options > Preferences > Acceleration, and the
service puts the session there:

Accelerator	Hardware	Where it comes from
`WebGpu`	Any graphics card: DirectX 12 on Windows, Vulkan on Linux, Metal on macOS	Ships with ARCx (plugin in `onnxruntime-ep/webgpu`)
`Cuda`	NVIDIA, GTX 16 / RTX 20 series and newer	Downloaded from the Preferences page; needs NVIDIA CUDA 13 and cuDNN 9
`CoreML`	Apple GPU and Neural Engine	Built into the macOS runtime
`Cpu`	The processor	Always there

```
var acceleration = ServiceRequestor.GetRequiredService<OnnxAccelerationService>();

// Uses the user's choice (Auto picks the fastest that works), and falls back to the
processor
// when the graphics card cannot start. configure runs once per attempt.
var created = acceleration.CreateSession(modelPath, (options, accelerator) => {

  options.IntraOpNumThreads = 4;
});

_session = created.Session;                // yours to dispose
log($"Running on {created.Description}"); // "WebGPU (NVIDIA GeForce RTX 4070)" or "CPU"

if (created.FallbackReason != null)
  log(created.FallbackReason);
```

Pass `accelerator: OnnxAcceleratorEnum.Cpu` to keep a model off the graphics card (small models,
or a skill setting that forces the processor). For `SessionOptions` you build yourself, call `ResolveAccelerator()` and then `AppendAccelerator(options, accelerator)`. Never call `AppendExecutionProvider_DML` or `AppendExecutionProvider_CUDA`: the runtime ARCx ships has neither
built in, and graphics cards only work through the service's plugins. The choice is read when a
session is created, so a skill picks up a change the next time it loads its model.

Platform	Native library
Windows x64 / ARM64, Linux x64 / ARM64, macOS ARM64	Shipped
macOS x64	None - upstream dropped it. `OnnxAccelerationService.RuntimeError` says so and creating a session throws `DllNotFoundException`; catch it and tell the user

- Keep the ONNX Runtime types in a class of their own and create it inside a `try`. An older ARCx
  without ONNX Runtime then fails there with `FileNotFoundException` naming

Microsoft.ML.OnnxRuntime,
which the skill can report, rather than failing to load at all.
• Models are large. Download them on first use into a folder of your own under `Constants.DEFAULT_FOLDER`, which survives a skill update, and check a checksum before using one.
• Leave cores for the rest of ARCx: set `SessionOptions.IntraOpNumThreads` to a few, not all.
• Run inference on a thread of your own, never in a video event handler or a Blazor event. See
Background threads.

Background threads

Work that loops for as long as the skill runs - reading a serial port or a HID device, polling a sensor, running a model - gets a **dedicated thread with a name**. Thread pool threads all show up
as `.NET TP Worker` in a debugger's Threads window, a profiler and the OS task manager, so when ARCx
is busy nobody can tell which skill is using the processor. A named thread says so.

```
Thread _readThread;

public override void Initialize() {

  _readThread = new Thread(readLoop) {
    IsBackground = true,
    Name = $"{Properties.WindowTitle} - Reader",
  };

  _readThread.Start();
}

void readLoop() {

  while (!IsDisposing && !IsDisposed) {

    // An exception that escapes a thread of your own ends ARCx, not just the skill
    try {

      if (AllWorkersPaused) {

        Thread.Sleep(100);

        continue;
      }

      // ... the work
    } catch (Exception ex) {

      ARCLogService.AddLogEntry(this, LogEntryCls.SeverityEnum.Error,
MethodBase.GetCurrentMethod(), ex);

      Thread.Sleep(1000);
    }
  }
}
```

• Name it after the window title and the job, so two copies of the same skill are told apart.
• Always `IsBackground = true`, or the thread keeps ARCx running after the user closes it.
• Catch everything inside the loop. An unhandled exception on a `Thread` terminates the process.
• `Task.Factory.StartNew(loop, TaskCreationOptions.LongRunning)` also gets a thread of its own,
named `.NET Long Running Task`. Set `Thread.CurrentThread.Name` as the loop's first

line.
• **Do not loop forever in `Task.Run`.** It holds a thread pool thread that Blazor, timers and every other skill share, and it cannot be told apart from the rest.
• **Do not rename a thread pool thread** (inside `Task.Run`, a timer tick or after an `await`).
The pool keeps the name after your work has finished, so it then points at the wrong skill. Short work belongs on the pool, unnamed; find it with a CPU profiler, which groups by call stack.

TimerSmart

`ARCx.Core.TimerSmart.TimerSmart`, a `System.Timers.Timer` that does not re-enter: a tick that
arrives while the previous handler is still running is **dropped**, not overlapped.

```
public override void Initialize() {

  _timer = new TimerSmart();
  _timer.Elapsed += _timer_Elapsed;
  _timer.Interval = 1000;
}

protected override void DisposeOverride() {

  // Unsubscribe BEFORE disposing - an elapsed event can already be queued on a thread pool
  // thread and would otherwise run against a disposed worker
  if (_timer != null)
    _timer.Elapsed -= _timer_Elapsed;

  _timer?.Stop();
  _timer?.Dispose();
  _timer = null;
}
```

`Elapsed` **hides** the base event - subscribe to this one. `IsEventRunning`, `IsDisposing`, and the base `Enabled`. `Start()` is ignored while already running; both `Start()` and `Stop()` are ignored once disposing.
The handler's first two checks are always the same:

```
// The skill may have been removed while this event was queued
if (IsDisposing || IsDisposed)
  return;

// A configuration window is open somewhere, so hold off
if (AllWorkersPaused)
  return;
```

DisposableBase

`ARCx.Core.Disposable.DisposableBase`. `IsDisposing` is set before `DisposeOverride()` runs,
`IsDisposed` after. Implement `protected abstract void DisposeOverride()`.

ConcurrentList

`ARCx.Core.ConcurrentList.ConcurrentList<T>`. A thread-safe list, used throughout ARCx where a
collection is written from skill threads and read from Blazor circuits.

Common

`ARCx.Core.Common` (static). Serialization (`SerializeObjectString`, `SerializeObjectCompressed`,
`DeserializeObjectFile`, …), filename and path safety (`GetSafeFilename`, `GetSafeSanatizedString`,
`CombinePathSafe`, `CombinePath`), string parsing (`IsNumeric`, `IsInteger`,

`GetNumbersFromString`, `GetDecimalFromString`, `TitleCase`, `SplitOnlyOnce`, `GetTextBetween`),
networking (`GetIPAddresses`, `GetWebResponseV2`, `PostWebResponse`, `IsInternetConnected`), and `GetOperatingSystem()`.
`ARCx.Core.Extensions` adds `DateTimeEx`, `NumericEx`, `StringEx`, `MemoryStreamEx`, `StreamToByteArray`.

---

24. Gamepads, joysticks and HID devices

`ARCx.Services.InputDevices.InputDeviceService`. Game controllers (Xbox, PlayStation, Switch,
8BitDo, flight sticks, wheels - anything SDL recognizes, over USB or bluetooth) and raw HID devices (custom USB boards, sensors, button boxes), on Windows, Linux and macOS.
The service owns SDL3 and is the only thing in ARCx that touches it. **Do not ship SDL, or any SDL
binding, in a robot skill.** Every skill loads into its own assembly load context, so a skill's SDL
types would not match the host's, and a second `SDL_Init` in the process fights ARCx for the devices. The public API uses only ARCx types, so a skill needs no package reference at all and
stays one AnyCPU build.
The user can turn the service off (Views, Gamepads And HID). Check `IsRunning`, and listen to
`OnRunningChanged` if you show its state.

Controllers

```
// Initialize(): subscribe. DisposeOverride(): unsubscribe, or the plugin assembly is pinned.
_inputDeviceService.OnGamepadAdded += inputDevice_OnGamepadAdded;
_inputDeviceService.OnButtonChanged += inputDevice_OnButtonChanged;

// Save the DeviceKey in your configuration; the InstanceId changes every time it reconnects
var gamepad = _inputDeviceService.FindGamepad(_Configuration.DeviceKey);   // "" = first
connected

// From a timer: a copy of the current state, never waits on the controller
var state = _inputDeviceService.GetState(gamepad.InstanceId);
float forward = -state.GetAxis(GamepadAxisEnum.LeftY);                     // up is negative
bool jump = state.IsPressed(GamepadButtonEnum.South);                      // A / Cross / B
```

Member	Notes
`GamepadInfo[] GetGamepads()`	Every controller connected, in plug-in order. Includes joysticks that are not gamepads - check `IsGamepad`
`GamepadInfo GetGamepadByIndex(int)` / `GetGamepad(uint instanceId)` / `FindGamepad(string deviceKey)`	Null when not connected. Scripts address controllers by index; configurations should save `DeviceKey`
`GamepadState GetState(uint instanceId)`	A copy. Null when disconnected. Cheap enough for a 20 ms timer
`bool WaitForButtonPress(id, button, timeoutMs, token)`	Counts presses, so a tap between polls is not missed. A button already held does not count
`Rumble(id, low, high, ms)`,	Strengths 0 to 1. False when the controller

`RumbleTriggers(...)`, `StopRumble(id)`	cannot
`SetLed(id, r, g, b)`, `SetPlayerIndex(id, n)`	Light bar and player lights, where the controller has them
`SetSensorsEnabled(id, bool)`	Accelerometer and gyro, off by default (they report hundreds of times a second). Values appear in `GamepadState.Accelerometer` / `Gyro`
`AddGamepadMapping(string)` / `GetGamepadMapping(id)`	Teach SDL an unrecognized controller, in SDL's mapping string format
Events	`OnGamepadAdded`, `OnGamepadRemoved`, `OnButtonChanged`, `OnAxisChanged`, `OnRawButtonChanged`, `OnHatChanged`, `OnRunningChanged`

Buttons and axes are named by position, not by the letter printed on them: `South` is A on
Xbox, Cross on PlayStation and B on Nintendo. `GamepadNames.GetButtonLabel(button, type)` gives the
printed label for display; `GamepadNames.TryParseButton()` accepts both kinds of name. Sticks read
-1 to 1 with **up and left negative**, triggers 0 to 1.
`GamepadInfo`: `InstanceId`, `Name`, `IsGamepad`, `Type` (`XboxOne`, `PS5`, `SwitchPro`, ...), `JoystickType`, `VendorId`, `ProductId`, `Serial`, `DeviceKey`, `Connection`, `RawAxisCount`, `RawButtonCount`, `HatCount`, `HasRumble`, `HasTriggerRumble`, `HasRgbLed`, `HasAccelerometer`,
`HasGyro`. `GamepadState`: `Axes`, `Buttons`, `ButtonPressCounts`, `RawAxes`, `RawButtons`, `Hats`, `Accelerometer`, `Gyro`, `BatteryPercent`, `PowerState`, `LastInputUtc`.
Events are raised on the service's own dispatch thread, in order. A slow handler delays the
other handlers, not the controllers, but keep handlers short: start a script or set a flag. Use `RefreshUI()` from them as usual. `OnAxisChanged` fires hundreds of times a second while a stick
moves and is dropped when handlers fall behind - to drive something from a stick, poll `GetState()`
on a timer instead. A handler that throws is logged, rate limited, and does not stop the others.

HID devices

```
// Look the device up by VID:PID each time; the path is not stable across ports and reboots
using var device = _inputDeviceService.OpenHidDevice(0x1234, 0xABCD);

device.Write(new byte[] { 0x00, 0x01, 0x02 });        // byte 0 is the report ID, 0 if
unnumbered
var report = device.Read(64, 100);                    // empty array on timeout
```

Member	Notes
`HidDeviceInfo[] GetHidDevices(vid = 0, pid = 0)`	Every HID device, or those matching. Throws when the service is not running
`HidDeviceConnection OpenHidDevice(string path)` / `(vid, pid, serial = null, usagePage = 0)`	Throws `IOException` when it cannot be opened - on Linux usually permission on `/dev/hidraw*` (INSTALL covers the udev

	rule)
`Write(byte[])`, `Read(byte[], timeoutMs)`, `Read(maxLength, timeoutMs)`	Read waits at most `HidDeviceConnection.MAX_READ_TIMEOUT_MS` (1000); loop for longer. On Windows the buffer must fit the device's longest input report
`SendFeatureReport`, `GetFeatureReport`, `GetInputReport`, `GetReportDescriptor`	hidapi semantics: the report ID is the first byte
`OnHidDevicesChanged`	Devices attached or removed. Only raised once something has used HID

Nothing in ARCx lists HID devices until something asks, because listing every HID device is what
can make macOS ask for Input Monitoring permission. **Dispose every connection in `DisposeOverride()`.** The service closes any still open when it stops, and they report `IsOpen == false` from then on.
A read and a write can run on different threads at once; two reads cannot. Do not do blocking
reads on a UI or event thread - give them a named thread of their own (see Background threads), and end its loop on `IsDisposing`.

---

25. Browser windows: web pages, dashboards, video calls and kiosks

`ARCx.Services.BrowserPresentation.BrowserPresentationService`. Chromium windows on the monitors
plugged into the computer ARCx runs on - a robot face, a dashboard, a WebRTC video call, a
kiosk screen - on Windows x64, Linux x64 and Linux ARM64. Not to be confused with the browser
showing the ARCx interface, which can be anywhere on the network.
On Linux no desktop environment is needed: a bare X server is enough, and the window is placed and
made fullscreen without a window manager. INSTALL, Browser windows, has the setup.
Do not ship CEF, CefSharp, CefGlue, WebView2 or any other Chromium embedding in a robot skill.
CEF can be initialized once per process, and ARCx runs it in a browser host process of its own
anyway (below). The public API uses only ARCx types, so a skill needs no package reference and
stays one AnyCPU build.

How it works

The service starts a **browser host process** (the ARCx executable again, with `--arcx-browser-host`) the first time a window is opened, and talks to it over a pipe. Chromium's
crashes stay in that process: the service restarts it and reopens every window, raising `OnWindowClosed` (`Reason = HostExited`, `WillReopen = true`) and then `OnWindowOpened` again. The
host stops two minutes after the last window closes.
Starting the host takes a second or two the first time (several on a Raspberry Pi). **Never call
`OpenWindow` from a Blazor render or event handler without `Task.Run`.** Every other call blocks

only for the few milliseconds the host takes to answer.

Opening a window

```
// Initialize(): nothing to create - windows open when you need them.
// Somewhere off the UI thread:
_window = _browserPresentationService.OpenWindow(new BrowserWindowOptions {
  Name = $"{Properties.WindowTitle} Face",          // global across ARCx - make it yours
  Url = "http://127.0.0.1:5000/_dynamic/" + Properties.InstanceGuid + "/face.html",
  DisplayMode = BrowserDisplayModeEnum.Kiosk,
  Monitor = _Configuration.Monitor,                 // -1 = primary
  AllowCamera = true,
  AllowMicrophone = true,
  OwnerTag = Properties.InstanceGuid,               // ARCx closes it if the skill is removed
});

_window.OnPageMessage += window_OnPageMessage;
_window.OnClosed += window_OnClosed;

// DisposeOverride(): unsubscribe, then close.
_window.OnPageMessage -= window_OnPageMessage;
_window.OnClosed -= window_OnClosed;
_window.Close();
```

`OpenWindow` with a name that is already open **re-points** that window (new Url, new options) and returns the same `BrowserWindow`, rather than opening a second one. Window names are shared by
every skill and script in ARCx, so prefix yours with the skill's window title.
A page the skill serves itself fits well with `OnDynamicResourceRequest` (UI manual, `UCDynamicImage`): serve `face.html` from `/_dynamic/[instance guid]/...` and point the window at
it. ARCx binds to 127.0.0.1 by default, which the browser host can always reach.

`BrowserWindowOptions`	**Default**	**Notes**
`Name`	required	The address scripts and the Browser Windows page use. Case insensitive
`Url`	required	http, https, file and data URLs. `example.com` becomes https, `localhost` and IP addresses http, an existing file path a file URL (`NormalizeUrl`)
`Title`	Name	The native window title
`DisplayMode`	`Windowed`	`Windowed`, `Fullscreen` (borderless over the monitor) or `Kiosk`
`Monitor`	-1	`BrowserMonitorInfo.Index`; -1 or a monitor that is gone means the primary
`X`, `Y`, `Width`, `Height`	0	Windowed only, relative to the monitor. 0 size = 1280x800 centered
`AlwaysOnTop`	false	Always true for Kiosk
`HideCursor`	false	CSS `cursor: none` on every

		page
`AllowCamera`, `AllowMicrophone`, `AllowScreenCapture`	false	getUserMedia / getDisplayMedia. **There is no prompt** - nobody is at a robot's screen to click Allow. A request for camera and microphone together is refused unless both are allowed
`AllowOtherPermissions`	false	Geolocation, notifications, clipboard, MIDI...
`PermissionOrigins`	empty	Restrict the permissions to these origins (`https://meet.example.com`, `*` wildcards). Empty = any
`AllowedUrlPatterns`	empty	Navigation allowlist (`https://dash.example.com/*`). Other main-frame navigations are cancelled and raise `OnNavigationBlocked`. Empty = anywhere
`PopupPolicy`	`OpenInSameWindow`	`window.open` / `target=_blank`: load it here, or `Block`
`IgnoreCertificateErrors`	false	For a device on your network with a self-signed certificate
`AllowDevTools`	true	F12 and `ShowDevTools()`. Never in Kiosk
`ZoomPercent`	100	25 to 500, kept across navigations
`Muted`	false	
`ReloadOnCrash`	true	Reload when the renderer dies; stops after 5 crashes in a minute
`RetryOnLoadError`, `RetryIntervalSeconds`	false, 10	Show a retry page that reloads the address. Always on for Kiosk
`ReopenWhenClosed`	false	Reopen when closed on the screen. Always on for Kiosk
`KeepDisplayAwake`	false	Stop the monitor blanking while open. Always on for Kiosk
`InjectCss`, `InjectScript`	empty	Added to every document the window loads, before the

		page's own scripts
`OwnerTag`	empty	Set it to `Properties.InstanceGuid`
`CloseWithProject`	true	Closed when the project closes or another loads

Kiosk is Fullscreen plus: always on top, no context menu, no keyboard shortcuts (Alt+F4 included), no developer tools, JavaScript `alert` logged and dismissed and `confirm` answered yes,
the retry page when offline, the display kept awake, and the window reopened if closed on the
screen. Close a kiosk with `Close()` - that one is final.

Controlling a window

`BrowserWindow` member	**Notes**
`Name`, `IsOpen`, `Options`, `State`, `Url`, `Title`, `IsLoading`	`Options` and `State` return copies
`Navigate(url)`, `GoBack()`, `GoForward()`, `Reload(ignoreCache)`, `Stop()`, `GoHome()`	`GoHome` returns to `Options.Url`
`WaitForLoad(timeoutMs, token)`	True when the page loaded, false on failure, timeout or close
`ExecuteScript(js)`	Fire and forget
`EvaluateScriptAsync(expression, timeoutMs)` / `EvaluateScript(...)`	The value as JSON (`"42"`, `"\"text\""`, `"{...}"`, `"null"`). Promises are awaited. Throws `BrowserPresentationException` with the page's error
`SendMessage(topic, data)`	To the page's `ARCx.on(topic, ...)`. `data` is serialized to JSON; a string holding JSON object/array text is passed through as that object
`WaitForMessage(topic, timeoutMs, token)`, `ClearMessages()`, `GetLastMessage(topic)`	Unread messages queue for 60 s (max 100); `WaitForMessage` takes the oldest. Events suit a skill better
`CaptureScreenshotAsync()`	PNG bytes
`SetDisplayMode(mode)`, `SetFullscreen(bool)`, `MoveToMonitor(i)`, `SetBounds(x, y, w, h)`, `SetAlwaysOnTop(bool)`	
`Show()`, `Hide()`, `Focus()`, `SetZoom(percent)`, `SetMuted(bool)`, `SetMediaPermissions(camera, mic)`	
`UpdateOptions(o => ...)`	Change any options at once; a changed Url navigates
`ShowDevTools()`, `Close()`, `GetConsoleHistory()`	Console history is the last 200 lines

`BrowserWindowState`: `Url`, `Title`, `IsLoading`, `CanGoBack`, `CanGoForward`, `IsVisible`, `DisplayMode`, `Monitor`, `X`/`Y`/`Width`/`Height`, `ZoomPercent`, `IsMuted`, `IsCameraInUse`, `IsMicrophoneInUse`, `LastHttpStatus`, `LastLoadError`, `CrashCount`, `OpenedUtc`, `OwnerTag`.

Service member	Notes
`OpenWindow(options)` / `OpenWindow(name, url)`	Opens or re-points. Throws `BrowserPresentationException` with a message for the user
`GetWindow(name)`, `GetWindows()`, `CloseWindow(name)`, `CloseAllWindows()`, `CloseWindowsOwnedBy(tag)`	`GetWindow` is null when not open
`GetMonitors(startHostIfNeeded)`	Only the host can see monitors: empty while it is not running unless you pass true
`IsAvailable`, `GetUnavailableReason()`, `Enabled`, `IsHostRunning`, `LastError`, `ChromeVersion`	Check `IsAvailable` before offering a screen feature in your UI
`EnsureHostRunning()`, `RestartHost()`, `StopHost()`, `ClearBrowsingData()`	Rarely needed; the service manages the host
Events	`OnWindowOpened`, `OnWindowClosed`, `OnWindowStateChanged`, `OnPageMessage`, `OnHostStateChanged`

`BrowserWindow` events: `OnStateChanged`, `OnLoadCompleted`, `OnLoadFailed`, `OnPageMessage`, `OnConsoleMessage`, `OnNavigationBlocked`, `OnCrashed`, `OnClosed`.

Events are raised on the service's own dispatch thread, in order. A slow handler delays the other handlers, never the browser, but keep them short and use `RefreshUI()` from them as usual. A handler that throws is logged, rate limited, and does not stop the others. Handle `OnClosed`: a window closed with `WillReopen == false` is gone, and its `BrowserWindow` stays closed - open it again for a new one.

The page side: window.ARCx

Every document the window loads - reloads, navigations, iframes - gets a `window.ARCx` object before its own scripts run, so a page can use it at the top level:

```
<script>
  // Robot to page: BrowserWindow.SendMessage("mood", "happy") / Browser.sendMessage(...)
  ARCx.on('mood', function (data) { document.body.className = data; });

  // Any topic
  ARCx.on('*', function (data, topic) { console.log(topic, data); });

  // Page to robot: raises OnPageMessage, answers WaitForMessage /
Browser.waitForMessage(...)
  document.getElementById('wave').onclick = function () { ARCx.send('command', { action:
'wave', speed: 5 }); };

  // Also delivered as a DOM event, for frameworks
  window.addEventListener('arcx-message', function (e) { console.log(e.detail.topic,
e.detail.data); });
```

```
</script>
```

`window.ARCx` only exists in an ARCx browser window; guard with `if (window.ARCx)` so the same page
still works in an ordinary browser. Data is anything `JSON.stringify` accepts. On the C# side,
`BrowserPageMessageEventArgs` has `Topic`, `DataJson`, `DataAsString` (a JSON string unquoted) and
`GetData<T>()`. Every frame in the window can call `ARCx.send`, including third-party iframes, so
treat page messages as user input.

Scripts, Blockly and the Browser Windows page

Everything here is also the `Browser` script namespace (JavaScript `Browser.open(...)`, Python
`Browser.Open(...)`) and the Browser category in Blockly, built on the same service - a skill and a
script can share a window by name. **Views, Browser Windows** lists the open windows with live
controls, the monitors, and the service settings (GPU acceleration, one renderer process, low end
device mode, the Linux X display, the remote debugging port). The Diagnostics page has a Browser Windows section, including the missing Linux
libraries when Chromium cannot start.

---

26. Depth cameras

`ARCx.Services.DepthSensing.DepthSensorService`. The meeting point between robot skills that **produce** depth (an Intel RealSense, Orbbec, OAK-D, Kinect, ZED, a ROS depth topic) and everything that wants to **use** it, the way `VideoRouteService` is for pictures and `KinematicsService` is for arms. A depth camera skill registers once and hands over every depth frame; any skill can then ask how far away something is, follow the depth live, or build its own maps from it, without knowing which camera it is.
It also does one thing for every vision skill at no cost to them: a detection made on a video source that a depth camera is **aligned** with (its pixels line up with the depth) is measured from the depth before `OnObjectsDetected` is raised. Face, colour, glyph and object detections then carry their real distance and position, and the Object Detection Script robot skill can stop the robot at a distance or drive to the object.

Frames, units and frames of reference

Type	What it is
`DepthFrame`	One depth image: `Width`, `Height`, `Data` (`ushort[]`, row by row, depth along the optical axis in units of `DepthUnitM` metres, `0` = no reading; the Z16 format every depth camera produces), `Intrinsics`, `FrameNumber`, `TimeUtc`. **Immutable once handed over**: give each frame its own array and never write to it again, because consumers keep frames and read them on other threads
`DepthIntrinsics`	The pinhole model: `Fx`, `Fy`, `Ppx`, `Ppy` for a `Width` x `Height` image, the numbers every

	SDK reports (RealSense `rs2_intrinsics`, ROS `CameraInfo.K`). `Scaled(w, h)` after decimating; `HorizontalFovRad`, `VerticalFovRad`
`DepthSensorMount`	Where the camera sits: `ForwardM`, `LeftM`, `HeightM` (from the floor), `PitchRad` (**positive tilted down**, ROS pitch), `YawRad` (counter-clockwise from the robot's front), `RollRad`. `OpticalToRobot(point)` turns a point from the camera's optical frame (+X right, +Y down, +Z out of the lens) into the robot's (+X forward, +Y left, +Z up from the floor); `OpticalDirectionToRobot(v)` does the same for a direction (an accelerometer reading). `HeightM = 0` means unknown
`DepthImuSample`	An inertial reading in the robot's frame: `Accel` (m/s², reads about +9.81 on Z when level), `Gyro` (rad/s, Z positive turning left), `HasAccel`, `HasGyro`, and `PitchRad` / `RollRad` from gravity (`PitchFromAccel()`, `RollFromAccel()` for your own vectors)
`DepthMeasurement`	Where something is: `DistanceM` (straight line from the lens), `DepthM`, `ForwardM`, `LeftM`, `HeightM`, `RangeM` (along the floor from the robot's centre), `Angle` (radians, counter-clockwise from straight ahead)

Everything follows the NMS frame of section 14: metres and radians, +X forward, +Y left, angles counter-clockwise.

Providing depth

```
// Once configured (and again with UpdateSensorDetails when the mount or the aligned source
changes)
_sensorGuid = _depthSensorService.AddSensor(new DepthSensorRegistration() {
  Name = Properties.WindowTitle,
  Description = "Orbbec Astra (serial 1234)",
  ProviderName = Properties.Title,
  AlignedVideoSourceName = $"{Properties.WindowTitle} Color",   // empty when the depth is
not aligned
  Mount = new DepthSensorMount() { HeightM = 0.35f, PitchRad = 0.2f },
  MinRangeM = 0.3f,
  MaxRangeM = 6f,
});

// Every frame, on the capture thread
_depthSensorService.UpdateDepthFrame(_sensorGuid, new DepthFrame() {
  Width = w, Height = h, Data = copyOfTheDepth, DepthUnitM = 0.001f, Intrinsics = k,
FrameNumber = n,
});

// DisposeOverride()
_depthSensorService.RemoveSensor(_sensorGuid);
```

Member	**Notes**
`Guid`	Throws on an empty or duplicate name

`AddSensor(DepthSensorRegistration)`	(case-insensitive). Keep the guid
`void RemoveSensor(Guid)`	Safe with an unknown guid
`void UpdateSensorDetails(guid, name, description, alignedVideoSourceName, mount, minRangeM, maxRangeM)`	Leave a parameter null (a range negative) to keep it; pass `""` as the aligned source to say nothing lines up any more
`void UpdateDepthFrame(Guid, DepthFrame)`	Stores the frame and raises `OnDepthFrame`
`void UpdateImu(Guid, DepthImuSample)`	For a camera with an inertial sensor. A few tens a second is plenty

Only register an aligned video source while the depth really is aligned to it pixel for pixel (same field of view; any resolution, the rectangle is scaled across). A detection on an unaligned source would be measured at the wrong place.

Turning depth into an NMS scan

`DepthScanBuilder.Build(frame, settings, treatCliffsAsObstacles)` is the shared way a depth camera feeds the navigation messaging system, so every camera behaves the same. It places every sampled pixel in the robot's frame with the mount, **removes the floor by height** (`MinObstacleHeightM`), keeps overhangs up to the robot's height that a lidar would miss (`MaxObstacleHeightM`, a table top), and with `DetectCliffs` finds the floor dropping away (stairs, ledges) and reports where the floor ends. The result holds the nearest obstacle and drop per slice of the circle (`ObstacleRangeM`, `CliffRangeM`, `SliceWidthRad`, `SliceOf()`, `SliceAngle()`, `NearestBetween(start, end, includeCliffs)`), the field of view edges, and `Points`, ready for `NMSService.UpdateScan()`. Without a mount height there is no floor to measure from and everything seen is reported, like a lidar. It is pure and thread safe; about 15 ms for 100,000 pixels on a desktop, and it samples to `TargetSamples` (30,000) by default.
A depth camera skill still publishes a placeholder pose when nothing else reports one, exactly as section 14 describes for a lidar.

Using depth

Member	**Notes**
`DepthSensor[] Sensors` / `string[] SensorNames`	Save the **name**, never the guid
`DepthSensor GetSensor(string name)` / `(Guid)` / `GetSensorForVideoSource(string sourceName)`	Null when unknown
`DepthSensor.LatestFrame`, `LatestImu`, `Mount`, `FPS`, `FrameCount`, `GetFreshFrame(maxAgeMs)`	A stopped camera leaves its last frame; `GetFreshFrame` returns null once it is older than you allow
`DepthMeasurement MeasurePixel(name, x, y, maxAgeMs)`	Depth image pixels; a 5 x 5 median so a single hole does not fail it
`DepthMeasurement MeasureRegion(name, x, y, w, h, maxAgeMs, nearPercentile)`	The median of the rectangle, or a nearer percentile for something thin in its box
`DepthMeasurement MeasureVideoRegion(name, x, y, w, h, videoWidth, videoHeight, ...)`	A rectangle on the aligned video, at its own resolution

`OnSensorAdded`, `OnSensorRemoved`, `OnSensorUpdated`	
`OnDepthFrame` (`DepthFrameEventArg`: `Sensor`, `Frame`)	On the provider's capture thread, every frame: keep it short
`OnImu` (`DepthImuEventArg`)	Often tens of times a second

`DepthFrame` also has `GetDepthM(x, y)`, `Deproject(x, y, depthM)`, `TryDeproject()`, `GetDepthInRectM()`, `GetValidFraction()` and `WritePly(stream, mount, step, colorBgr)` for a point cloud MeshLab, CloudCompare, Blender and Open3D open. `DepthVisualizer.Colorize(frame, minM, maxM)` makes the near-red, far-blue picture to publish as a video source.
The attributes a measured detection gains (`DetectedObject.ATTRIBUTE_*`): `DistanceM` (from the lens, replacing a detector's estimate), `ForwardM`, `LeftM`, `HeightM`, `RangeM`, `Angle` and `DepthSensor`. A detection with no depth reading inside its box keeps the attributes its detector gave it.

---

27. Robot Bus: topics, services and frames

`ARCx.Services.RobotBus.RobotBusService`. A publish/subscribe message bus, named request/response services and a tree of frames, shared by robot skills, scripts (the `Bus` namespace), the MCP server and programs outside ARCx (WebSocket and REST at `/api/bus`, Python and JavaScript clients at `/sdk/`). The user manual is `Manuals/Third-Party Integrations/robot-bus.md`.
Use it when what your skill produces or offers should be reachable by things you cannot name in advance: another skill, a Python program on another computer, a ROS node through the ROS Connector, an AI assistant. The typed core services above (video, NMS, kinematics, depth) stay the way to plug into ARCx's own features; ARCx already publishes their data on `/arcx/...` topics, so do not publish it again.

Conventions

- Names look like ROS names: `/my_skill/battery`. Put your skill's topics under `/` + `BusNames.Segment(Properties.WindowTitle)` so two copies of the skill do not collide.
- Payloads are JSON. Publish a record or anonymous object; it is serialized once with camelCase names. Give a `type` (free text, such as `my_skill/Battery`) and advertise the topic with a description.
- Units are the NMS's: metres, radians, +X forward, +Y left, +Z up, angles counter-clockwise (ROS REP 103).
- The source of everything you publish or provide is your window title.

Publishing and subscribing

```
// Initialize()
_topic = $"/{BusNames.Segment(Properties.WindowTitle)}/battery";
_robotBusService.Advertise(_topic, "my_skill/Battery", "Pack voltage (V) and charge (0..1)",
Properties.WindowTitle);

_subscription = _robotBusService.Subscribe("/arcx/speech/recognized", onHeard, new
BusSubscribeOptions() {
  Owner = Properties.WindowTitle,
  QueueDepth = 16,       // oldest dropped when the callback falls behind
  MaxRateHz = 0,         // above 0: the newest message per topic at most this often
  DeliverLatest = false, // true: the current latest message straight away
});

// Whenever there is something new, on any thread
_robotBusService.Publish(_topic, new { voltage = 12.1, charge = 0.82 }, "my_skill/Battery",
Properties.WindowTitle);
```

```
void onHeard(BusMessage message) {

  // On the subscription's own task, one message at a time. Never block for long.
  var text = message.GetField("text") as string;
}

// DisposeOverride()
_subscription?.Dispose();
```

`BusMessage`: `Topic`, `Type`, `Sequence` (per topic, from 1), `StampUtc`, `Source`, `Data` (a detached `JsonElement`), `DataJson`, `GetData<T>()`, `GetPlainData()`, `GetField("pose.x")`. Patterns: `*` one segment, `**` any number. `GetLatest(topic)`, `GetTopic(topic)` (rate, count, publishers), `GetTopics(pattern)` and `WaitForMessageAsync(pattern, timeout, ct)` read without subscribing.

Providing and calling services

```
_service = _robotBusService.RegisterService(
  $"/{BusNames.Segment(Properties.WindowTitle)}/calibrate",
  "Calibrate the sensor. Args: samples (int, default 50). Returns the offsets.",
  async (request, ct) => {

    var samples = request.GetInt("samples", 50);

    return await calibrateAsync(samples, ct);
  },
  Properties.WindowTitle,
  readOnly: false,
  argsSchema: "{\"type\":\"object\",\"properties\":{\"samples\":{\"type\":\"integer\"}}}");

// DisposeOverride()
_service?.Dispose();

// Calling one (never throws for a failed call)
var result = await _robotBusService.CallServiceAsync("/arcx/speech/say", new { text =
"Calibrated" }, Properties.WindowTitle, TimeSpan.FromSeconds(5), ct);

if (!result.Ok)
  Log(result.Error);
```

The handler returns any value (serialized to JSON) and throws to fail the call; the exception message is what the caller sees. Honour the token: it is cancelled when the caller's timeout runs out. `readOnly: true` marks a service that changes nothing, which external programs may call even when Preferences forbid them to control the robot. A service name has one provider; registering a taken name throws.

Frames

```
// A camera bolted 10 cm forward, 30 cm up, tilted 0.2 rad down
_robotBusService.SetTransform("base_link", "my_camera", 0.10, 0, 0.30, 0, 0.2, 0, isStatic:
true, source: Properties.WindowTitle);

// Where is it from the odometry frame?
if (_robotBusService.TryLookupTransform("odom", "my_camera", out var t, out var error)) {

  var (x, y, z) = t.Apply(1.5, 0, 0);   // a point 1.5 m in front of the camera, in odom
}
```

`BusTransform` is a translation and a unit quaternion: `FromEuler`, `FromQuaternion`, `Then`, `Inverse`, `Apply`, `Rotate`, `Roll`, `Pitch`, `Yaw`, `Distance`. ARCx keeps `odom → base_link` from the NMS pose and `base_link → <depth sensor>` from each depth camera's mount. Changes are published on `/tf` and `/tf_static`. Where `odom` is on the earth is not a frame but the NMS geo reference, published on `/arcx/nms/geo_reference` whenever a GPS skill sets or moves it, with each fix on `/arcx/nms/fix` (see section 14).

Recording

`_robotBusService.Recorder` records topics to `ARCx Data\Recordings` and plays them back (`StartRecording(patterns, name)`, `StopRecording()`, `ListRecordings()`, `StartPlayback(name, speed, loop, patterns, topicPrefix)`, `StopPlayback()`). A skill that consumes topics works on a replay exactly as on live data, which is the point: record once on the robot, debug at the desk.

---

28. Rules that will bite you

1. **Create nothing in the constructor.** Timers, executors, service registrations and event subscriptions all go in `Initialize()`.
2. **Unsubscribe from every event and dispose everything in `DisposeOverride()`.** A leaked handler pins the plugin assembly; ARCx then cannot overwrite the installed files and the skill will not rebuild until ARCx is restarted.
3. **Unsubscribe before disposing a timer**, not after. A queued elapsed event runs against a disposed worker otherwise.
4. **Check `IsDisposing` / `IsDisposed` and `AllWorkersPaused`** at the top of every timer tick and background loop.
5. **Look up other robot skills only in `OnProjectLoadCompleted()`**, never in `Initialize()` or `OnSetConfiguration()`.
6. **Null-check `TryGetEZB()` and `IsConnected` before every hardware call.** A project can reference a controller that is no longer there, and a disconnected board silently does nothing at best.
7. **Save names, not guids.** A video source guid and a kinematic chain guid change every session. Names and window titles survive a save and load.
8. **Never hold a `Mat` past the end of a video event handler.** Clone what you need.
9. **Deregister what you registered**: `RemoveSource()`, `RemoveChain()`, `DeregisterInputSource()`, `RemoveAndDisposeExecutor()` - all in `DisposeOverride()`.
10. **`RefreshUI()`, not `StateHasChanged()` on the worker.** It is throttled, thread safe, and reaches the Custom UI widgets too.
11. **Give every configuration property a `///` summary.** It is what Athena reads.
12. **Build AnyCPU.** A robot skill is platform-agnostic managed code and ARCx loads the same build on Windows, Linux and macOS. Never add a platform-specific code path or build: `dotnet build "<Project>.csproj" -c Debug -p:Platform=AnyCPU`. Name the `.csproj`: a solution file beside it (Visual Studio saves one) spells the platform "Any CPU", and `dotnet build` given only the folder picks the solution and fails with MSB4126.
13. **Do not log from a tight loop.** Every `ARCLogService` entry is written to disk.
14. **Do not ship SDL.** Controllers and HID devices come from `InputDeviceService`; a second copy of SDL in the process takes the devices away from ARCx and from every other skill.
15. **Do not ship a Chromium embedding** (CEF, CefSharp, CefGlue, WebView2). Browser

windows come
from `BrowserPresentationService`, which runs Chromium out of process for you. Open windows off
the UI thread, set `OwnerTag` to your instance guid, and close them in `DisposeOverride()`.
16. **Reference the ARCx libraries you use, and only those.** A skill compiles against the
References item group in its `.csproj`: ARCx, MudBlazor and DeepCloner to start with. For another library ARCx ships - `OpenCvSharp` for camera frames, `NAudio.Core` for audio, `Microsoft.ML.OnnxRuntime` for neural networks, `System.IO.Ports` for serial ports, `System.ServiceModel.Primitives` for ARCx's web service clients and the rest listed in the
template's comment - copy its line out of that comment. Each one is `Private=false`, so nothing
is copied into the output; ARCx already has it loaded. CS0246 or CS0234 on one of its types, or
CS0012 naming an assembly, means the reference is missing. A library ARCx does not ship is a
NuGet `PackageReference`.
17. **Do not ship ONNX Runtime.** ARCx ships it, native library included, for every platform - see
section 23. Only one onnxruntime native library can be loaded into a process,
so a skill carrying its own fights ARCx and every other skill that runs a model.
18. **Give a long-running loop its own named background thread**, never a `Task.Run` that never
returns. Name it after `Properties.WindowTitle` and the job so a profiler or debugger shows which
skill is using the processor, and catch everything inside it - see
Background threads.

Where to look next

- `ARCx.xml` - beside `ARCx.dll`, in the folder this copy of ARCx runs from. The full XML documentation for every public class, method, property and event, generated from the platform's
own source. It is around a megabyte, so search it rather than reading it: every entry is a `<member name="...">` element whose name is the fully qualified member, prefixed `T:` for a
type, `M:` a method, `P:` a property, `E:` an event, `F:` a field. Grep `T:ARCx` to list the types, `M:ARCx.Core.EZBManager` for the manager's methods, `VideoRoute` for everything about
cameras.
- `RobotSkill-UI-Components.md` - the visual side: the component base classes, layout, the `UC*`
components, Custom UI widgets.