Umicom Studio IDE – Initial Roadmap and Setup

Project Scaffolding (Folder Structure & Templates)

To kickstart the IDE project, we should implement a project initialization routine that creates a standard directory structure and template files. Many modern tools offer this kind of scaffolding. For example, Rust's Cargo and Zig provide commands to scaffold a "Hello World" project with needed files [1][2], and V offers `v init` / `v new` to set up a basic project [3]. We will follow a similar approach:

1. **Define Standard Structure:** Create a base folder (project root) with subfolders like **`src/`** for source code and **`include/`** for headers. For multi-language support, we might also include language-specific folders (e.g. `asm/` for assembly, `scripts/` for Python, etc.) as needed.
2. **Generate Template Files:** Populate the new project with minimal files that compile or run immediately. For example, a C/C++ project might include `src/main.c` (with a simple `main()` function) and a build configuration file (like a **CMakeLists.txt** or Makefile). We can also include a README.md with project info.
3. **Use Utility Functions:** The Umicom engine already provides file and directory helpers (e.g. `mkpath()` to create directories recursively, and `write_text_file_if_absent()` to create files if they don't exist [4]. We can leverage these in an `init` command to create the scaffolding. (In the current codebase, `cmd_init` seeds a book project [4]; we will adapt this to code project scaffolding.)
4. **PowerShell Support:** For Windows developers, we can also provide a PowerShell script to do the same steps from the command line. This script will create directories and files using PowerShell commands (ensuring compatibility with Windows 11).

Example (Scaffolding a C project with an Assembly snippet):

- **PowerShell commands:** To scaffold manually (for testing), one can run in PowerShell:

```
New-Item -Type Directory -Path .\MyProject\src -Force
# create src/ folder
Set-Content -Path .\MyProject\src\main.c -Value $@"
#include <stdio.h>
extern int getValue();
int main() {
    printf("Value from assembly: %d\n", getValue());
    return 0;
}
"@
Set-Content -Path .\MyProject\src\getvalue.s -Value $@"
    .text
    .globl getValue
getValue:
    movl $42, %eax    # return 42 in EAX/RAX
    ret
"@

Set-Content -Path .\MyProject\CMakeLists.txt -Value
```

```
"cmake_minimum_required(VERSION 3.16)
project(MyProject C ASM) add_executable(MyProject
src/main.c src/getvalue.s) "
```

The above creates a **`src/main.c`** (C source calling an assembly function), an **`src/getvalue.s`** (x86-64 assembly returning a value), and a minimal **CMakeLists.txt**. This ensures the project is ready to build and run.

- **In-Engine command:** Ultimately, the IDE will include an `init` command (or menu action) to perform the same steps. For instance, after integrating the above logic, running `umicomstudio init <ProjectName>` can create a "Hello World" project that **"builds immediately"** [4]. This mirrors how tools like Cargo (`cargo new`) create a new package with a default main and config [1].

Compilation & Build Automation (C/C++ with Assembly)

Automating the build process is crucial. We should support common build systems and compilers outof-the-box, with the ability to add others via plugins or configuration:

- **Cross-Platform Build System:** Use CMake as a default build generator for C/C++ (and ASM) projects. CMake is widely used in open-source C/C++ projects (e.g. wxWidgets and many libraries) and can generate build files for different compilers. Deepin-IDE, for example, supports CMake, Ninja, Maven, Gradle, etc., acknowledging that developers use various build tools [5]. By adopting CMake, we allow flexibility: on Windows it can invoke MSVC or MinGW/Clang, and on Linux/macOS it can use GCC/Clang seamlessly.
- **Initial Compiler Integration:** Focus on GCC and Clang first. Ensure the IDE can detect and call these on Windows 11 (for instance, using LLVM's `clang` or a MinGW GCC installation). The build script or CMake config should be set up to compile both C/C++ and assembly files. For example, the CMakeLists above declares the project languages as C and ASM, so CMake knows how to handle `.s` assembly files.
- **PowerShell Build Script:** Provide a PowerShell script to automate build steps on Windows. For example, a **`build.ps1`** could configure and invoke CMake:

```
cd MyProject
# Configure the project (out-of-source build in 'build\' directory)
cmake -S . -B build -G "Ninja" -DCMAKE_BUILD_TYPE=Debug
# Or use "MinGW Makefiles" generator if Ninja isn't installed.
# Build the project cmake --build
build --config Debug
```

This script sets up the project and compiles it. (Users would need CMake and a compiler in PATH. In the future, the IDE can auto-detect these tools or prompt the user.) - **Immediate Build Verification:** The scaffolded "Hello World" project should compile *without errors*. For instance, using the files from the previous section, running the above build commands would produce an executable that prints `Value from assembly: 42` . This verifies that both the C code and x86 assembly were compiled and linked correctly. - **Extensible Compiler Support:** The architecture will allow adding other compilers or build systems via plugins or settings. For example, if a user prefers MSVC, the IDE could invoke *msbuild* or *cl.exe* with appropriate flags. By designing a plugin interface for compilers, users could add support for

languages like Zig or Rust (which have their own build tools) easily. Initially, we focus on GCC/Clang for C/C++ due to their broad availability and consistent CLI interface.

LLM-Driven Code & Markdown Editing

One distinguishing feature planned for Umicom Studio is AI-assisted coding and documentation. This means integrating Large Language Models (LLMs) to help generate or edit code and markdown content:

- **Code Generation & Explanation:** The IDE will provide an AI assistant that can write boilerplate code, suggest improvements, or explain code sections in natural language. For instance, you could ask the assistant to "generate a function to compute factorial," and it would insert the code. This is similar in spirit to GitHub Copilot's suggestions within editors [6], but we aim to allow more interactive, high-level requests (and support local models).
- **Natural Language Commands:** We can draw inspiration from Anthropic's *Claude Code*, which lets you use natural language commands to perform coding tasks in a project [7]. For example, Claude Code can commit code changes or refactor functions when you type instructions in plain English [8]. In our IDE, a user might select a piece of code and ask the LLM to optimize it or add comments, and the IDE would apply the edit.
- **Markdown Documentation Edits:** The AI assistant can also help with documentation. For example, if there's a README or code comments, the model could refine language or generate usage examples. Given the engine's origin as an authoring tool (with functions for packing Markdown chapters into a book draft [9] [10]), this is a natural extension. The user could maintain project docs with AI assistance, ensuring consistency and clarity.
- **Integration in Editor UI:** The IDE should have an **"AI Edit"** or **"Ask AI"** feature in the editor. This might be a sidebar or modal where the user enters a prompt (e.g., "Explain what the following function does" or "Convert this code to use iteration instead of recursion"). The assistant then returns an answer or code diff. The result can be reviewed and applied by the user.
- **Safety and Control:** When editing code via AI, we'll implement a review step. Edits could be presented as a diff that the user can accept or reject. This aligns with best practices to keep the developer in control of the code changes (similar to how Copilot's **edit mode** works interactively [11]).
- **Local vs Remote Models:** We plan to support both local and remote LLMs for these features. Initially, connecting to OpenAI's API (for models like GPT-4) is straightforward for high-quality suggestions. We have a module (`llm_openai.c`) to handle API calls (likely using HTTPS). Simultaneously, we prepare for local model integration – for example, using **LM Studio** or similar. *LM Studio* is an open-source app that lets users run local LLMs on their own hardware [12], and it even provides a Python SDK [13]. Umicom Studio could interface with such a local runtime (or directly with libraries like llama.cpp) to enable offline AI assistance. This dual approach (remote API or local model) will be abstracted so that the "chat" and "edit" features work regardless of backend.

Debugging and Terminal Integration

A modern IDE needs to integrate debugging tools and provide a built-in terminal for convenience:

- **Debug Adapter Protocol (DAP):** We will implement debugging support using the Debug Adapter Protocol, which is a standardized way to interface with debuggers. This approach is used by VS Code and others, and Deepin-IDE also adopts DAP for debugging modules [14]. By using DAP, Umicom Studio can support multiple debuggers (gdb, lldb, or even Windows CDB) through

a common interface. For example, the IDE can launch a DAP server for GDB, allowing features like breakpoints, step execution, and variable inspection in a uniform way.

- **UI for Debugging:** The IDE will have a **Debug Panel** where you can start/stop the program, step through code, set breakpoints, and watch variables. Initially, we might integrate a simple GDB frontend – for instance, invoking GDB with the program and allowing basic controls via commands. As we integrate DAP, this will become more robust and GUI-driven (with call stacks, memory view, etc.).
- **Terminal Emulation:** Provide an **integrated terminal** within the IDE so developers don't need to switch out for command-line tasks. This could be a terminal emulator control on Windows (PowerShell or CMD by default) and a Bash/Zsh on Unix. Many IDEs and editors embed a terminal for running build commands or Git operations. We'll do the same, likely by spawning a PTY and embedding its output in a panel. On Windows 11, we can leverage the modern terminal APIs or simply run PowerShell in a hidden window and capture I/O.
- **PowerShell Integration:** Given priority to PowerShell (as the user's environment is Windows 11), we ensure the terminal opens with PowerShell by default. This allows running any commands, including the build script or git, directly in the IDE. For example, after scaffolding a project, the user can press a **"Open Terminal"** button in the IDE and execute `cmake --build` or even Python scripts right there.
- **Testing the Debug Flow:** As a simple test, we can compile the example project and then use GDB to run it. In the IDE, the user would set a breakpoint at `main` and hit "Debug": the IDE should pause at `main` and allow inspecting `getValue()` returns. Using DAP, the common debugging actions (continue, step, etc.) will function across supported languages and platforms [1].

Plugin System and Live Preview

To keep Umicom Studio extensible and versatile, we will design it with a **plugin architecture**. Features like language support, build tools, and even UI components (like a live preview) can be added or removed as plugins:

- **Language as Plugins:** Following a modular approach, each programming language or domain can be handled by a plugin. Deepin-IDE uses this strategy – C/C++, Java, Python support are all via plugins, communicating with the core via language protocols [7]. We will do similarly. For example, an **Assembly plugin** could provide syntax highlighting and compiler integration for assembly files, a **Python plugin** could enable running and debugging Python scripts, etc. The plugin would register the language's file extensions, how to compile or run them, and what LSP server to use for smart features.
- **Language Server Protocol (LSP):** Rather than writing our own code analysis for each language, we'll integrate existing LSP servers. LSP is an open standard that many languages support for IDE features (autocomplete, go-to-definition, linting). Using LSP decouples the IDE from language specifics [1]. For instance, we can bundle or prompt to install the C/C++ language server (such as clangd) for C-family code intelligence, Rust Analyzer for Rust, etc. The plugin's job is to launch the LSP server and forward requests. This approach means from day one, we can support many languages (C/C++, Python, Zig, Rust, etc.) with rich editing features by leveraging their LSPs.
- **Build/Tool Plugins:** Similarly, support for different build systems or tools can be plugin-based. A plugin could define how to invoke a particular compiler or packager. For example, a **Rust plugin** might call `cargo build` under the hood, a **Zig plugin** could wrap `zig build`, etc. By abstracting "build tasks," the core IDE can remain generic.

- **Live Preview:** Live preview is especially useful for web development or markup editing. We plan to have a plugin (or core module) for live preview of HTML/Markdown and possibly GUI layouts:
- **Web/HTML Preview:** For web projects, the IDE could include an embedded browser view that auto-reloads the served page when files change. If the user is editing an HTML file or a markdown document, a split-view can show the rendered output. For Markdown, we can use a library (or even our own engine's Pandoc/Markdown integration if available) to render to HTML and display it. The engine already has code to generate HTML from Markdown chapters [19] that can be repurposed for on-the-fly rendering of docs.
- **GUI Preview:** In the future, for languages with GUI toolkits, we might allow designing interfaces visually. For now, a simpler target is to support previewing outputs of code – e.g., if the code produces an image or graphical output, show it in the IDE.
- **Example – Live Markdown Preview:** We could have a **Markdown Preview** panel that watches `.md` files. When a file is saved, the plugin uses our `pack_book_draft` or a similar function to produce HTML [23] then refreshes the preview pane. This gives real-time feedback for documentation writing.
- **Plugin Management:** We'll maintain a config file or directory where plugins are listed. Plugins could be implemented as dynamic libraries (for performance-critical tasks in C/C++), or as scripts (Python or Lua scripts controlling the IDE via an API). In the initial stage, some "plugins" might be baked into the application (for core languages like C/C++ and assembly), but by structuring them as if they were external, we keep the door open for later separation. Users should be able to enable/disable certain language support or features by adding/removing plugins without recompiling the whole IDE.

AI Assistant – Chat Window Integration

As discussed, the AI assistant will play a significant role. To make it user-friendly, we'll incorporate a **Chat Window** in the IDE where the user can converse with the LLM:

- **Dedicated Chat Panel:** This will be a panel (docked, say, on the right side) with a conversation view. The user can ask questions or give instructions in a text box (e.g. "How do I use this library?" or "Review my code for errors"). The assistant's responses are displayed in a chat transcript above.
- **Contextual Awareness:** The assistant will be aware of the project's context. For example, if the user highlights a block of code and opens the chat, the assistant could automatically see that code to answer questions about it or to perform transformations. Anthropic's Claude Code tool can "understand your codebase" when answering queries [8] we aim for similar capability. Our backend code (`serve.c` and `llm_*.c`) can be extended to feed relevant files or symbols to the LLM as needed.
- **Local vs Remote Model Selection:** In the chat interface, the user could choose which model to use (e.g., a dropdown for "OpenAI GPT-4" vs "Local LLaMA 2 13B"). For local models, we integrate with a backend like LM Studio or directly load a model via a library. *LM Studio* for instance can run models like Mistral or Llama2 on consumer hardware [21] We might run LM Studio in headless mode or use its Python API [22] to get responses. The chat panel will abstract this – it sends the prompt to whichever LLM backend is configured and streams the answer back for display.
- **Usage Example:** A developer could type: *"LLM, explain the function in* main.c *that calls assembly."* The assistant, seeing `main.c` and `getValue()` implementation, might respond: *"This function `getValue` is implemented in assembly. It simply returns the constant value 42, which `main` then*

- *prints. The assembly uses the EAX register to set the return value 24. Such an answer is generated by analyzing code context (note: our engine might pass the assembly file content to the model for analysis).*
  AI Code Actions: Beyond Q&A, the chat should allow triggering actions. For example, *"Generate a C function to reverse a string"* could insert code into the editor. Or *"Find bugs in this file"* could result in a list of potential issues. Initially, this will be manual (the user copies suggestions), but eventually we can integrate these as one-click fixes or refactorings (with user approval).
- **Ethical & Safety Measures:** Since the assistant can make code changes or suggestions, we'll include usage guidelines and possibly filters (especially for remote models like OpenAI which have their own content policies). Logging interactions (locally) and providing a way to undo AIapplied changes will be implemented to ensure user trust and the ability to back out if needed.

Multi-Language Support (Assembly, C/C++ first, then Python, Zig, Rust, etc.)

From the outset, Umicom Studio will be multilingual. The core editor will not be tied to one programming language. Here's the plan for supporting multiple languages smoothly:

- **Assembly (x86/x64):** Assembly is a first-class citizen. Many IDEs have limited support for assembly, but we will include syntax highlighting and build integration for assembly language. Using the plugin system, we'll treat assembly like any other language: e.g., an **ASM plugin** can provide highlighting rules and perhaps integrate an assembler (NASM/YASM) if needed. However, since we already plan to compile assembly via GCC/Clang or as part of C projects, basic support is already in place. The example earlier with `getvalue.s` shows how we compile assembly alongside C. We will ensure both Intel and AT&T syntax files can be recognized. As a bonus, we can allow the user to write inline assembly in C/C++ and still get it recognized by the IDE.
- **C/C++:** As the backbone of systems programming, C/C++ support will be robust. We use Clang's language server (clangd) for code intelligence, and compilers as discussed for building. The IDE should be able to handle single-file scripts as well as multi-file projects. Features like code completion, error squiggles, and go-to-definition will be enabled via clangd. We will also allow configuring different C/C++ compilers through settings (e.g., MinGW, MSVC, Clang).
- **Python:** Python is hugely popular and will be supported early. We'll integrate the Python LSP (pylsp or Microsoft's PyLance, depending on availability) for code intelligence. The build/run for Python is simply executing the script with the interpreter, which we can do easily (the IDE can call `python main.py` and capture output). We can include a basic virtual environment manager or at least respect one if present. Python plugin would handle running scripts and possibly debugging (using ptvsd or debugpy through DAP).
- **Zig:** Zig is a newer systems language we want to support. Zig has its own build system and package manager built-in [25][26]. We can support Zig by calling the Zig compiler directly. For instance, to build a Zig project, the IDE might run `zig build` (after `zig init` if scaffolding a new project). We saw that `zig init-exe` creates a project with a `build.zig` and source file . Our plugin could invoke these commands under the hood. Code intelligence can come from the community Zig LSP (if one exists; if not, basic syntax highlighting still helps).
- **Rust:** Rust will be included as well, given its popularity. We'll rely on `rust-analyzer` for smart editor features, as recommended by Rust developers [27][28]. Building/running Rust code will use `cargo` commands. A new Rust project can be made via `cargo new` (which produces a

- Cargo.toml and a hello-world main [29]). The IDE can automate that, similar to how we scaffold C projects. Debugging Rust can go through DAP with lldb or gdb (Rust's tools integrate with those).
- **Other Languages:** We won't stop at the above. The architecture should handle *any* language, provided there's an LSP server or at least a syntax file and a way to run or compile it:
- **Web Languages:** HTML/CSS/JS – integrate with web development workflows. Possibly include a browser preview (as noted) and use the LSPs for HTML/CSS and TypeScript/JavaScript.
  Go, Java, etc.: These can come later via their LSPs and build tools (Go's `go build`, Java's Maven/Gradle integration).
- **Experimental languages:** The list given (Carbon-lang, C3, Gravity, etc.) shows interest in new languages. We can keep an eye on these. For instance, Carbon and C3 are C++/C successors in development [30] [31] – as they mature, adding support in our IDE early could attract communities. Since Carbon is LLVM-based, it might reuse some C++ tooling; C3 has its own compiler we could call. Gravity is a small language (embeddable); we might include its interpreter for script running. The key is our modular approach – adding a language is a matter of dropping in the right server and configuring build/run commands.
- **Testing Multi-language Projects:** The IDE should allow a single project to contain multiple language source files (e.g., a C++ core with Python scripts for high-level logic, or a Rust library used via FFI in C). With LSPs, each file can be handled by the appropriate language server concurrently. We'll treat the project as a "workspace" containing possibly multiple modules. This is another advantage of using LSP – it's designed to work in multi-root workspaces and different file types gracefully.

In summary, **Umicom Studio's initial implementation** will set up the groundwork: a Windows-friendly development environment with **project scaffolding** for C/C++ and assembly, automated builds via **CMake and PowerShell**, an **AI-assisted editor** for code and docs, integrated **debugging and terminal**, a flexible **plugin system**, and broad **language support from day one** (Assembly, C/C++, Python, Zig, Rust, etc.). By learning from existing open-source projects and standards – from Deepin-IDE's architecture [16] [14] to Claude-Code's AI workflow [7] – we can implement these features in a coherent way. Each piece we add (be it a CMakeLists.txt template or an LSP integration) will be committed to the repository as we go, so the IDE grows organically with a clear history. The next steps will involve coding these components, testing on Windows 11 via PowerShell, and iterating based on feedback.

Sources:

- Deepin IDE official presentation – modular architecture, plugin system, LSP/DAP usage [16] [14] [32]
- Rust in VS Code – using `cargo new` to scaffold projects (Hello World) [1]
- Zig language getting started – project init and build commands [2]
- V language tutorial – `v init` / `v new` project scaffolding (with prompts) [3] [33]
- Anthropic Claude Code – AI coding assistant capabilities in terminal/IDE [7]
- LM Studio documentation – running local LLMs on your PC (for offline AI) [12]
- Umicom Engine Code – existing helpers for file system and project init (e.g., `cmd_init` for minimal project) [4]

---

[1] [27] [28] Rust in Visual Studio Code
https://code.visualstudio.com/docs/languages/rust

[2] [25] [26] Getting started with the Zig programming language - LogRocket Blog

- https://blog.logrocket.com/getting-started-zig-programming-language/

3 33 Vlang Tutorial: A Practical Guide : Part 1 | by mohammed alaa | Medium
https://medium.com/@m.elqrwash/vlang-tutorial-a-practical-guide-65da80b8b4ac

4 20 main.c
file://file-AJCvVgrmdDAQGvtA1Ux9zi

5 14 15 16 17 32 Official presentation: deepin-IDE – Deepin Technology Community
https://www.deepin.org/en/2023-9-4/

6 GitHub Copilot in VS Code
https://code.visualstudio.com/docs/copilot/overview

7 8 GitHub - anthropics/claude-code: Claude Code is an agentic coding tool that lives in your terminal, understands your codebase, and helps you code faster by executing routine tasks, explaining complex code, and handling git workflows - all through natural language commands.
https://github.com/anthropics/claude-code

9 10 18 19 23 24 fs.c
file://file-F6e4Z9UFacg62YFuhC88yV

11 Multi-file editing, code review, custom instructions, and more for ...
https://github.blog/changelog/2024-10-29-multi-file-editing-code-review-custom-instructions-and-more-for-github-copilot-invs-code-october-release-v0-22/

12 LM Studio - Local AI on your computer
https://lmstudio.ai/

13 21 Get started with LM Studio | LM Studio Docs https://lmstudio.ai/docs/app/basics

22 Model Catalog - LM Studio
https://lmstudio.ai/models

29 Creating a new project - The Rust Edition Guide
https://doc.rust-lang.org/edition-guide/editions/creating-a-new-project.html

30 Carbon Language's main repository: documents, design ... - GitHub https://github.com/carbon-language/carbon-lang

31 C-like language C3 is an evolution, not a revolution | Interesting: find ... https://www.x-cmd.com/blog/240801/