Usage of Modern IDEs for developing robotics applications

1st Tobias Madlberger
Software Engineering
HTL St. Pölten
St. Pölten, Austria
tobias.madlberger@htlstp.ac.at

2nd Leander Klik
Electrical Engineering
HTL St. Pölten
St. Pölten, Austria
leander.klik@htlstp.ac.at

***Abstract*—When it comes to programming software, the choice of the correct Integrated Development Environment, short IDE, is crucial. This is also the case when it comes to coding robotic applications. Choices are very limited when developing on a Wombat. This paper examines the difficulties related to switching to a different IDE, specifically from the standard KISS IDE to a more recent one. By looking into the setup procedure, which includes cross-compiling for Raspberry Pi and strategies for deployment, we explain the complexities of combining powerful IDE features with robotics projects. Based on practical experiences and problem-solving approaches, we offer insights into overcoming certain obstacles encountered during the transition.**

***Index Terms*—Robotics Application Development, IDE, Cross-Platform Development, Harrogate**

I. Introduction

A. Background

Botball, organized by the KISS Institute for Practical Robotics (KIPR), stands out as a leading educational robotics competition engaging middle and high school students. Teams undertake the challenge of designing, building, and programming autonomous robots to tackle dynamic tasks on a game table. Unlike remote-controlled robots, Botball places a strong emphasis on autonomy, nurturing creativity, and problem-solving skills. The competition's rules evolve annually, ensuring fresh challenges. Teams compete on a global stage with some of the best teams from around the globe at the Global Conference on Educational Robotics (GCER), where they showcase their skills [11].

B. The provided controller and its integrated IDE

Writing the code for the tasks faced in the competition can be done with the editor provided by the included controller, which is named the Wombat. It is responsible for exposing a simple interface to control the included hardware. The controller is built out of a Raspberry PI 3b+ with a custom circuit board, where different parts of the hardware can be plugged in when building the robot for the competition. The device itself has software preinstalled by the organization selling it, including the editor called the KISS IDE. It's a very minimalistic code editor, only providing a very limited feature set. Most people use this supplied IDE, as the code editor provides all the functionality needed to write a working program. It's very easy to learn, making it a great starting point for beginners and especially students new to the field of robotics.

C. Benefits of an IDE

An Integrated Development Environment (IDE) is a code editor that provides a comprehensive toolkit for programmers to develop software with it. It typically combines several tools and functionalities into a single graphical user interface, making the process of writing, testing, and debugging code more efficient and productive [8].

Some common features found in modern IDEs include:

- **Code Editor**: IDEs offer a text editor with features like syntax highlighting, code completion, and code formatting, making it easier for developers to write and manage code.
- **Build Automation Tools**: IDEs often integrate build automation tools like compilers, linkers, and build systems to help developers compile and build their projects efficiently.
- **Debugging Tools**: Debugging tools integrated into IDEs allow developers to identify and fix errors in their code more effectively. They typically include features like breakpoints, step-through debugging, and variable inspection.
- **Version Control Integration**: Many IDEs provide integration with version control systems like Git, allowing developers to manage code changes, collaborate with team members, and track project history within the IDE.
- **Project Management**: IDEs often include project management features such as project templates, file organization, and project navigation tools, helping developers organize and manage complex software projects.

D. The need for a different IDE

The shift from the integrated IDE to a different one isn't just about surface changes but about strategically improving the development process and productivity. This change is driven by various factors listed in *Section I-C*. Also, the rise of cross-platform development requires IDEs that can work across different hardware and operating systems.

For more experienced pupils, the KISS IDE might not be advanced enough to meet their needs. This editor, for instance, makes it difficult to have a multifile project and limits the teams overall productivity by hiding syntax errors until the code is compiled. Such problems and many others might put many students into a challenging situation — While using the integrated IDE might be straightforward, it lacks the capabilities of a more advanced IDE.

When it comes to deciding whether your team uses the KIPR Software Suite for developing their robot or if they bring their own IDE, it can vary. An important factor is the familiarity with the code editor. When most of the team has little to no knowledge about the chosen editor, it might be hard to write code in it.

Overall, IDEs play a crucial role in modern software development, providing developers with the tools and workflows necessary to create high-quality software efficiently.

II. State of The Art

The KISS IDE has been designed to be straightforward to use, and that's why it only provides basic features. For example, the integrated editor only supports basic syntax highlighting. However, it's missing the feature to show syntax errors upfront.

While the KISS IDE has undoubtedly earned its place, exploring alternative solutions can uncover valuable insights. According to this market share estimate taken in 2018, the commonly used IDEs are Visual Studio, Vim, Qt Creator, Visual Studio Code and CLion [2]. While the top four IDEs (and heavily modified text editors) appear to be free to use, CLion is a paid professional product. But they offer a free license for students [3]. Therefore, it's a great opportunity for students to get to know a professional grade IDE and its advanced features.

When looking at these IDEs in detail, it becomes clear that they offer a lot of features that the KISS IDE doesn't. Some of them even have extensions, making development for robotics even easier. For example, Visual Studio Code is widely used among the robotics development community, as it's very flexible when it comes to customization through plugins. Developers can easily install extensions to support specific languages or frameworks, making it adaptable for robotics applications. The environment includes debugging capabilities, essential for robotics development, though the interface and functionality can vary based on the language and extensions used. This flexibility allows for customization to meet unique requirements and preferences [10]. It also supports debugging, which is a crucial feature when it comes to developing robotics applications [6].

III. Concept

Initially, a basic C-Project setup is needed. This can be generated with the IDE of choice or by simply creating the necessary files, to define dependencies and run the compilation.

Essential to the project is the inclusion of the Kipr Library, which provides the required functionalities for our application. Incorporating this library allows the interaction with various components of the Wombat effectively. To get the library, it has to be copied from the Wombat to the project environment, as shown in *Figure1*. The libwallaby header files and libwallaby library binary can be found in the '/usr/include' and '/usr/lib' directories, respectively.

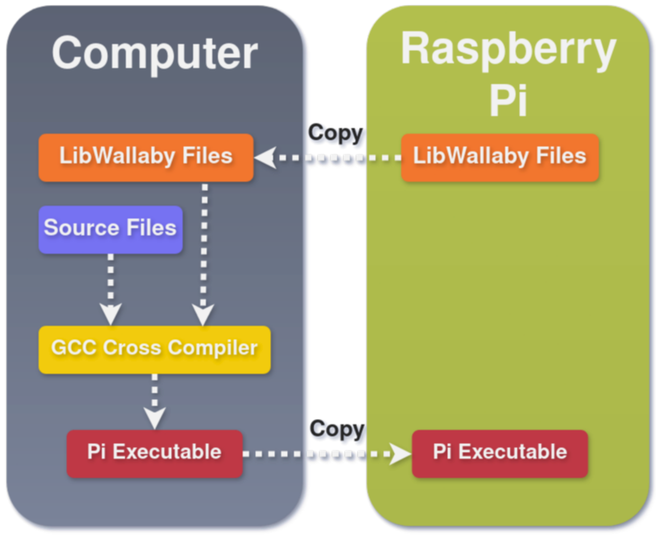

Fig. 1. Flow chart for the cross compilation setup

When it comes to compiling the binary, a special build process has to be used since the final binary shall be executed on the Raspberry Pi. The Wombat, the target the project has to run on, uses a Raspberry Pi 3b+ for running the operating system [4]. These hardware limitations require the C code to be compiled specifically for this architecture, making this a cross-compilation process [9].

An automated deployment pipeline is needed to upload the binary to the target device, once it has been compiled. This ensures that any changes to the code are transferred to the Wombat in a speedy manner, so that iteration and testing are possible without delay.

Furthermore, the configuration of the execution environment on the Wombat ensures that the program can be started from the robot's UI. The program must be able to start by using either the touchscreen of the controller or the mouse.

IV. Implementation

A. *Establishing Connection to the Wombat*

Connecting to the Wombat necessitates access to its dedicated network, which the device autonomously generates. Network credentials, including the SSID and password, are retrievable from the 'Advanced → Network Settings' page and are also more simply available on the About page.

Upon successfully establishing a network connection, it is often sufficient to directly initiate an SSH session with the Wombat. This can be done using the `ssh` command, assuming the Wombat's IP address is `192.168.125.1` and the username is `kipr`. Authentication is completed by entering the provided password, conventionally set as `botball`.

For enhanced security and to facilitate passwordless SSH authentication for future connections, the user can generate an SSH key pair if it has not already been done. This involves navigating to the `ssh` directory, located at `/home/[username]/.ssh` on an Ubuntu machine, and using the tool `ssh-keygen` to create an RSA-formatted key pair. The public key content, identifiable by the `.pub` extension, should then be copied.

Instead of manually integrating the copied public key into the `authorized_keys` file within the `.ssh` directory of the user's home folder, the `ssh-copy-id` command can be used for convenience. This command copies the public key to the Wombat's authorized keys file automatically. For example:

```
ssh-copy-id kipr@192.168.125.1
```

Note: The instructions provided assume default settings. Actual configurations may vary depending on user modifications.

Note: The instructions provided assume default settings. Actual configurations may vary depending on user modifications.

B. *Setting Up the Development Environment*

The setup starts with the installation of an appropriate IDE. Follow the installation instructions described in the documentation of the editor to ensure a smooth installation process.

After the successful installation of the wanted editor, a new project has to be created. The optimal configuration would contain a CMakeLists file, containing instructions to build a C++ Executable with the appropriate C++ Language standard, whereby Standard 20 is preferred.

To incorporate 'libkipr' into the project, the user establishes an 'include' folder within the project structure, housing the essential kipr header files. These files are retrieved from Wombat using appropriate commands, ensuring accessibility within the project environment.

C. *Cross-Compiling for the Raspberry Pi*

The first approach to this was to send all sources onto the Raspberry Pi and compile the project there. It turned out that this is a terrible idea. The first issue is that a lot of files have to be transmitted over the network, which itself might already take very long. Not just the transition is very tedious, the compilation and linkage itself is extremely slow. As the Raspberry Pi only has very limited resources, it takes very long to compile a simple project.

Due to these circumstances, another approach had to be tried. This time, the problem has been approached by setting up a docker container, emulating the Raspberry Pi with all its installed packages and structure. This turned out to work quite well, at least when it came to just building the project. It compiled the project fine and was fast while doing so. The built sources were even runnable on the pi, but linking the kipr library failed with many issues. It was missing the libx11 and libpthreads library, which was fixable by installing appropriate dev packages for these libraries, making the linker find the required files.

Even though docker was able to build the project, the need for a solution that doesn't require 39 seconds *[Table I]* to compile lead to the actual solution in use. At first, the arm64 compiler for C++ was installed, and a cmake toolchain was set up and configured. The configuration process included defining the correct linker and compilers. Then, when running the cmake tooling, the newly created toolchain has been supplied as a console argument. This made it possible to compile the whole project on a much faster computer than directly on the Raspberry Pi.

D. *Cross Compilation Benchmark*

To evaluate the effectiveness of each approach, a benchmark was conducted comparing the performance of compiling the project using three different strategies: compiling directly on the Raspberry Pi, using a Docker container emulating the Raspberry Pi environment, and cross-compiling with a dedicated toolchain on a separate machine. The benchmark was performed on a set of ten trials for each strategy, with the compilation time recorded for each trial. The results are summarized in *TableI* and visualized in *Figure2*.

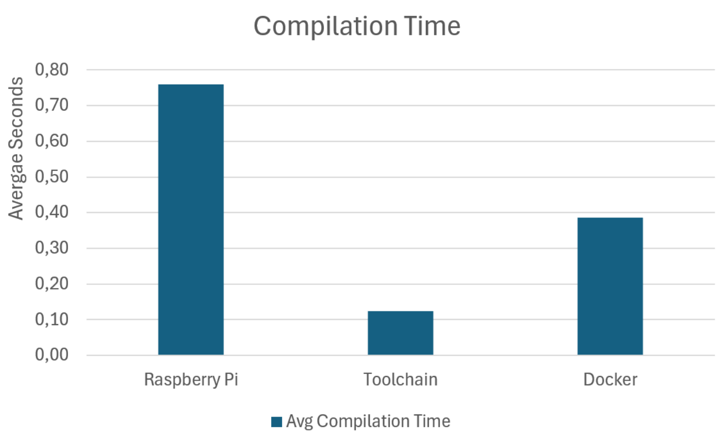

Fig. 2. Benchmark Results of Cross Compilation Strategies

TABLE I
BENCHMARK RESULTS OF CROSS COMPILATION STRATEGIES

Strategy	**Raspberry Pi**	**Toolchain**	**Docker**
Average (seconds)	75.88	12.38	38.55

From the benchmark results, it's evident that compiling directly on the Raspberry Pi is the slowest approach, with an average compilation time of 75.88 seconds. This method suffers from the limitations of the Raspberry Pi's hardware.

Using Docker to emulate the Raspberry Pi environment significantly reduces compilation time compared to compiling directly on the Raspberry Pi, with an average time of 38.55 seconds. However, issues with library dependencies hindered the usability of the compiled binaries.

The most efficient approach proved to be cross-compiling with a dedicated toolchain on a separate machine, which had an average compilation time of 12.38 seconds. This method not only drastically reduces compilation time but also overcomes the limitations of the Raspberry Pi's hardware and avoids dependency issues encountered with Docker.

E. Deploying the binary

With the previously configured and now running build process, the only thing left to do is build the deployment part of this pipeline. This is needed to upload the compiled binary onto the robot, which then enables the execution of it. This can be achieved by using the secure copy tool to upload a file onto the target device. The targeted folder is important here: To run the program from the UI on the Wombat, a C-Project has to be created using the project window on the Wombat first, as an automatic creation isn't possible because otherwise the program won't show up in the UI. To achieve this, open the KISS IDE and create a new project.

After the project has been created, the binary can be uploaded to the project's build directory. When running the KIPR OS version 30.2.5 on the Wombat, this folder is located at '/home/kipr/KISS/projects/[projectName]/build,' whereby '[projectName]' shall be replaced with the name of the recently created project.

When the file has been uploaded, clicking the run button on your robot will execute the deployed binary.

F. Debugging the binary

Debugging the binary can be done by using the GDB debugger, as the binary needs to run on remote device. With the uploaded binary, the GDB server has to be started on the Wombat. As the Wombat doesn't come with it preinstalled, it has to be installed first using the package manager. Once installed, the debugger can be started by running the command 'gdbserver :[port] [pathToBinary]'. The port can be any free port on the Wombat and the pathToBinary is the path to the uploaded binary's path on the Wombat.

Once the debugger is running, the GDB client can be started on the development machine. Modern IDEs usually provide a very convenient way to start the debugger, as they have a built-in debugger. For example, when using CLion, a run configuration needs to be specified, containing the IP address of the Wombat and the port the debugger is running on. Once the run configuration has been set up, debugging is as easy as clicking the debug button, which will start the debugging session.

V. Results

Despite these challenges, the transition to a modern IDE proved to be a worthwhile endeavor. The more sophisticated IDE allowed us to work on complicated robotics projects more efficiently. Writing code is now much quicker and less error-prone due to the reasons stated in *Section I-C*.

VI. Conclusion

To wrap up, the study explored a setup that helps develop robots with a different IDE than the KISS IDE. Throughout our exploration, we encountered both benefits and challenges in transitioning to a modern IDE for robotics development.

One notable benefit of using a modern IDE is its comprehensive feature set, which significantly enhances developer productivity. For instance, the advanced tab completion and code analysis features helped a lot when it came to quickly identifying syntax errors and potential bugs in our codebase. Additionally, the seamless Git integration facilitated efficient version control, allowing for easy collaboration and code management among team members.

However, the transition was not without its challenges. One significant hurdle was setting up cross-compilation for the Raspberry Pi. Initially, the attempt to compile the project directly on the Pi turned out to be very slow and inefficient, due to its limited resources. This led to exploring alternative approaches, such as emulating the Raspberry Pi using a Docker container. While this approach showed promise, the linking of the KIPR library failed. Ultimately, using the arm64 compiler and a CMake toolchain file to compile the project on a faster machine did the job.

References

[1] Haziqa, "Text Editor vs. IDE: Which Is The Best For Beginners?", 2022. [Online]. Available: https://www.ultraedit.com/blog/text-editor-vs-ide-which-is-the-best-for-beginners/. Accessed: March. 4, 2024.

[2] Davide Coppola, "Market share of the most used C/C++ IDEs in 2018, statistics and estimates," blog.davidecoppola.com, 2018. [Online]. Available: http://blog.davidecoppola.com/2018/02/market-share-most-used-c-cpp-ides-in-2018-statistics-estimates/. Accessed: Feb. 28, 2024.

[3] "Education Tools for Students and Educators," JetBrains, 2024. [Online]. Available: https://www.jetbrains.com/community/education/#students. Accessed: March 5, 2024.

[4] "Wombat Development Toolkit Manual," GitHub. [Online]. Available: https://github.com/kipr/KIPR-Development-Toolkit/blob/master/Docs/WombatDevManual.pdf. Accessed on: March 5, 2024.

[5] KIPR, "Hardware & Software," www.kipr.org. [Online]. Available: https://www.kipr.org/kipr/hardware-software. Accessed on: March 6, 2024.

[6] Microsoft, "Visual Studio Code - Debugging", code.visualstudio.com, 2024. [Online]. Available: https://code.visualstudio.com/docs/editor/debugging. Accessed on: March 6, 2024.

[7] Gaurav Raturi, "Cross-Platform App Development: Exploring Emerging Trends," LinkedIn, 2023. [Online]. Available: https://www.linkedin.com/pulse/cross-platform-app-development-exploring-emerging-trends-raturi-vf4kf. Accessed on: March 6, 2024.

[8] Amazon Web Services, "What is an IDE (Integrated Development Environment)?", aws.amazon.com, 2024. [Online]. Available: https://aws.amazon.com/what-is/ide/. Accessed on: March 6, 2024.

[9] Anne Barela, "Cross Compiling: Compile C Programs on," March 2016. [Online]. Available: http://21stdigitalhome.blogspot.com/2016/03/cross-compiling-compile-c-programs-on.html. Accessed on: March 6, 2024.

[10] Daniel Biehl, "RobotCode - Language support for Robot Framework for Visual Studio Code," Visual Studio Code Marketplace, Available: https://marketplace.visualstudio.com/items?itemName=d-biehl.robotcode. Accessed on: March 6, 2024.

[11] "Botball," KISS Institute for Practical Robotics, 2024. [Online]. Available: https://www.kipr.org/botball. Accessed on: March 9, 2024.

[12] "Features of an IDE," adacomputerscience.org, 2024. [Online]. Available: https://adacomputerscience.org/concepts/soft_ide_examBoard=all&stage=all. Accessed on: March 9, 2024.