THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

The PiSharp IDE for Raspberry PI

Bo Si

This thesis is submitted in partial fulfillment of the requirements for the Degree of Master of Science at the University of Waikato.

August 2017

Abstract

The purpose of the PiSharp project was to build an IDE that is usable for beginners developing XNA-like programs on a Raspberry-Pi. The system developed is capable of

1. Managing and navigating a directory of source files
2. Display a file in a code text editor
3. Display code with syntax highlight
4. Automatically discovering program library structure from code namespaces
5. Compiling libraries and programs automatically with recompilation avoided if source code has not been updated
6. Compiling and running from the IDE
7. Editing more than one file at a time
8. Providing code compilation for global and local names of variables, methods and classes.

Acknowledgements

Firstly, I would like to thank to Bill Rogers. I am very grateful for his kindness, concern and endless help over this project.

Secondly, I would like to thank to Lichao Wang and James Boud for their previous contributions inspired this project.

Last but not the least, I would like to thank to my family and friends for their support and help.

Contents

Chapter 1. Introduction

Today, “computer” is no longer only the PC, but also smartphones, video game consoles, tablets, wearable electronics, and any embedded digital devices with programmable computing resources. They are participating in numerous activities in people’s work, study, entertainment and daily lives. Almost everyone roughly knows about how to interact with them. Most people, especially the younger generations that are digital natives after a short-term familiarity can simply operate their computers, e.g. check emails, browse web pages and open up a picture or a movie onto the display by clicking a mouse or swiping a finger. People believe computers are smart because they can do very fast and accurate computation and carry out diverse tasks. When they enjoy the convenience of using featured software to interact with their computers, most are only curious about a computer’s capability rather than how a computer actually works, because it is a basic part of people’s lives, just like a television or a microwave. In fact, computers are not as smart as most people think. Programming process develops all software, from simple applications to complicate operating systems. Computers only attempt to achieve users’ expectations or complete pre-assigned when they receive a set of instructions. In short, the “smart” of computers comes from careful and specific programs, and not computer components.

When mentioning the phrase “computer graphics”(CG), the first thing that most people will associate would be animation, movies effects or video games. With the rapid development of computer graphics technology, a great number of movies and games are fully computer animated. They are becoming one of the most common entertainment forms. Aside from these, CG technology is also used in a board field, involving images and charts of data in a presentation, architectural design and modeling, urban planning, animated product demonstration, advertising animation, scientific simulation and so on. Compared to text or static picture, CG technology uses give more intuitive feedback to people. There is no doubt that, with the highly improved graphics processing unit, computer graphics still have enormous potential uses that are waiting for more people to develop and program.

1.1 Raspberry Pi

The purpose of this project is to port an existing IDE from Windows platform that helps people easily learn and write 3D graphics programs on a Raspberry Pi. The Raspberry Pi(R-Pi) is a single board microcomputer that has been developed by the Raspberry Pi Foundation in the United Kingdom. With a display and input devices like a mouse and a keyboard, the R-Pi is sufficient for most general-purpose things that a typical PC can do, like games, high-definition videos and the Internet access. According to the Raspberry Pi official documentation, the organization mainly intended to promote the teaching and learning of computing fundamentals in education institutions over the world so that more people of all ages would be able to take part in learning programming and understanding how computers work in a low-cost way [1].

The concepts for Raspberry Pi's early prototypes started from Atmel Corporation manufactured microcontrollers in 2006[2]. Later, due to the more and more mature processors for mobile devices that were powerful enough to provide excellent multimedia, the first formal Raspberry Pi(Model B revision 1) was launched in early 2012. Few months later, an upgraded revision of Model B was released with as twice as much memory space than the previous one. So far, the Raspberry Pi family members increased to about 20, including Model A series, Model B series, Compute Module series, Zero series. Raspberry Pi 2 and Raspberry Pi 3 are both in Model B series that represent the second and the third generation of Raspberry Pi. The Figure 1 from Rasp.tv shows a photo of the Raspberry Pi family up to February 2017. Among them, the Model A family, Compute Module family, and Zero family are stripped-down devices. They cut down the power of the processor, RAM, or other components to reduce the size of the board for low power embedded projects or industrial applications. Compared with the same period of the stripped-down models, Model B series are the higher-spec variants of Raspberry Pi allowing more complex embedded projects and also general computer use. The Raspberry Pi 2 Model B is used for this project. Figure 2 shows the layout of Raspberry Pi 2 Model B board. It has a Broadcom BCM2836 System-on-a-Chip(SoC), which integrates an ARMv7 processor, GPU block and 1GB RAM.

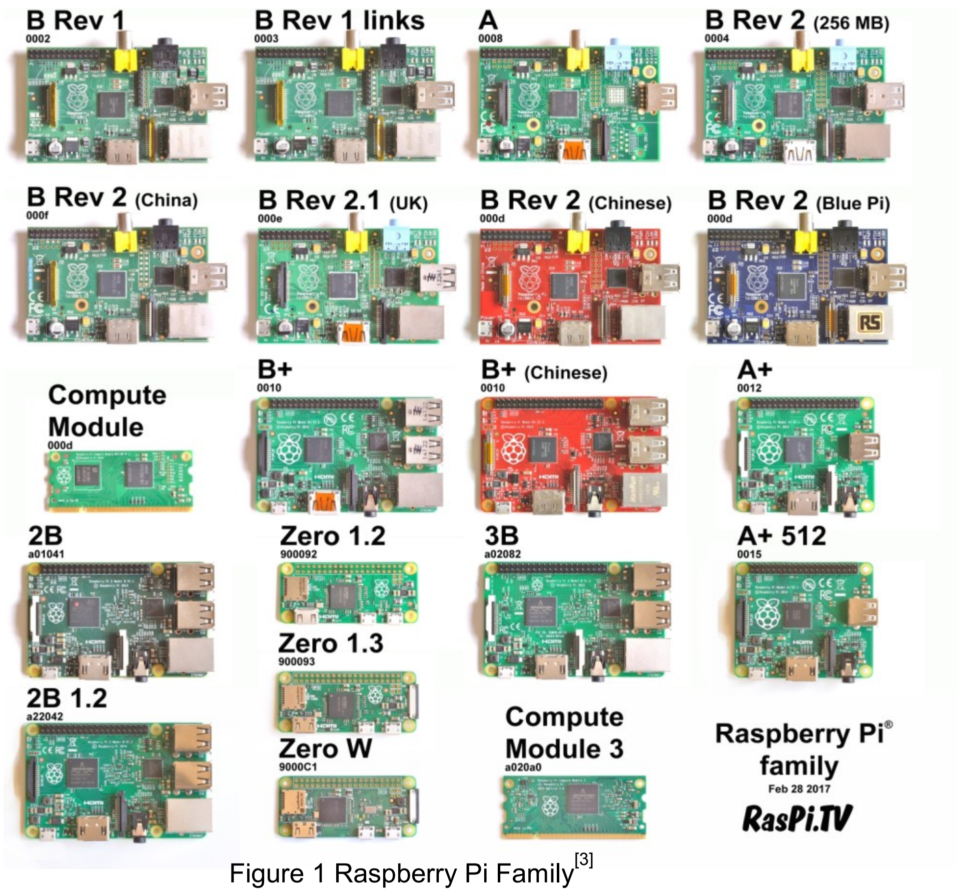


Figure 1 Raspberry Pi Family[3]

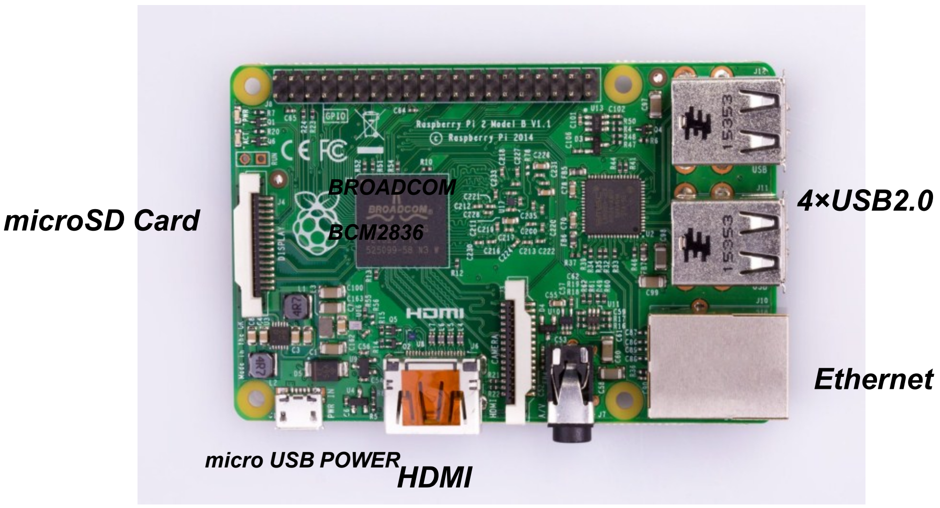


Figure 2 Raspberry Pi 2 Model B [4]

Unlike the majority of processors, for example, Intel and AMD as found on most desktop PC and laptops using the x86 or x86-64 Instruction Set Architecture(ISA), BCM2836 uses ARM architecture. The ARM architecture today is not designed for the typical PC, but it is very commonly utilized in modern mobile devices because ARM is a simple reduced instruction set computer with low power consumption. An instruction set architecture is the form of machine code that a processor can execute. Because different ISAs determine different specifications of machine codes and native commands, a program compiled for the x86 ISA, will not be able to executed by an ARM processor. For this reason, the ARM-based Raspberry Pi will be not compatible with major software for typical x86-based PC.

Raspberry Pi Foundation claimed that the GPU on Raspberry Pi "provides OpenGL ES 2.0, hardware-accelerated OpenVG, and 1080p30 H.264 high-profile encode and decode. The GPU is capable of 1Gpixel/s, 1.5Gtexel/s or 24GFLOPs of general purpose compute and features a bunch of texture filtering and DMA infrastructure" [5]. It means that the GPU generates 1*109 display pixels/second, 1.5*109 texture access/second, 24*109 floating-point calculations/second. Therefore, Raspberry Pi is capable of running a well-maintained 3D graphics program.

Table 1 lists three generations of Raspberry Pi Model B specifications: the original Raspberry Pi, Raspberry Pi 2 and Raspberry Pi 3. Compare with the original Raspberry Pi board, the second and third generations mainly upgrade the chip that improves the performance on CPU, GPU or RAM. The chip on Raspberry Pi 2 provides a 900MHz quad-core 32-bit ARMv7 CPU with 1GB RAM, and the one on Raspberry Pi 3 is upgraded to 1.2GHz 64/32-bit ARMv8 CPU and a more powerful GPU. In addition, both R-Pi2 and R-Pi3 board are equipped with 4 USB2.0 ports for mouse, keyboard or other USB-port external devices connection. They add more General Purpose Input-Output(GPIO) headers up to 40 pins and integrate both composite video and audio into a 3.5mm phone jack as a video output method. Besides the Ethernet, Raspberry Pi 3 also supports Wireless and Bluetooth for network access.

	Original Raspberry Pi	Raspberry Pi 2	Raspberry Pi 3
Released Date	April-June 2012	February 2015	February 2016
Price	US $35		
System-on-a-Chip	Broadcom BCM2835	Broadcom BCM2836	Broadcom BCM2837
CPU	700MHz ARM1176JZF-S	900MHz ARM Cortex-A7	1.2GHz ARM Cortex-A53
No. of Cores	1	4	
Architecture	32-bit ARMv6	32-bit ARMv7	64/32-bit ARMv8
GPU	Broadcom VideoCore IV @ 250 MHz OpenGL ES 2.0, 24 GFLOPS		3D part of GPU @ 300 MHz, Video part of GPU @ 400 MHz OpenGL ES 2.0, 28.8 GFLOPS
Memory(RAM)	512MB*	1GB	
Onboard Storage	SD	Micro SD	
Onboard Network	10/100 Mbit/s Ethernet		10/100 Mbit/s Ethernet; Wireless(150 Mbit/s); Bluetooth(24 Mbit/s)
No. of USB2.0	2	4	
No. of GPIO	26 pins	40 pins	
Video Output	HDMI; raw LCD panels via MIPI display interface (DSI); composite video(R-Pi: RCA jack/R-Pi2, R-Pi3: 3.5mmTRRS jack)		
Audio Output	3.5mm phone jack; HDMI		
Power Source	5V via Micro-USB or GPIO header		

*Early launched Raspberry Pi Model B board just had 256MB RAM

Table 1 Raspberry Pi Models specifications[6]

To keep the weight and size low, all Raspberry Pi models are designed to use an SD card/micro SD card for long-term storage and booting rather than a hard drive(HD) or solid-state drive(SSD). The Raspberry Pi 2 provides three different approaches to video output: a 3.5mm TRRS jack for composite video, an HDMI(High-Definition Multimedia Interface) port for plugging into a high-quality display like a TV or a monitor with high resolution and a DSI(Display Serial Interface) for the flat-panel displays typically used in tablets and smartphones.

1.2 Operating Systems(OS) on Raspberry Pi

An operating system is used to manage hardware and to provide a number of libraries allow software running on a computer. Windows and OS X operating systems are dominant in the majority of household and family desktop and laptop PCs. However, the official Raspberry Pi OS uses neither of them, because they are close sourced which is difficult to be modified to run on ARM-based Raspberry Pi. In contrast, Linux system is an open-sourced operating system widely used in servers and embedded systems. The Raspberry Pi Foundation promotes and maintains its official operating system based on one of Linux distributions called "Debian" Wheezy and names it as "Raspbian". The "Raspbian" refers to the combination of "Raspberry Pi" and "Debian". It is specially designed for the Raspberry Pi. The Programming languages installed with Raspbian on Raspberry Pi are Python, C, C++, Java, Scratch and Ruby. More detail of Raspbian will be described in Chapter 3 – "Underlying System".

There are several third-party distributions ported to the Raspberry Pi, such as Ubuntu MATE(based on regular Ubuntu ARM Hard-Float distribution is exclusive to Raspberry Pi 2 and Raspberry Pi 3), Windows 10 IOT Core(a version of Windows 10 for embedded devices), RISC OS(a non-Linux distribution but widely used on ARM chipset), and OSMC(open-source media center) [7], and so forth. The Windows 10 IOT Core also called "Windows 10 Internet of Things Core" is available for Raspberry Pi 2 and 3. It is mainly designed to allow development of programs for embedded devices using Visual Studio on Windows, with remote running and debugging via network connection. However, "it does not include the user interface('shell') or the desktop operating system"[8]. Google's popular Android platform, which is widely used by mobile devices, in particular smartphones and tablets around the world, is developed on a Linux core. This implies Android is possible for Raspberry Pi. Although a few versions of Android for R-Pi are available from the Raspberry Pi community and R-Pi enthusiasts, those operating systems are not stable enough for everyday uses.

1.3 Alternatives to Raspberry Pi

The mainline Raspberry Pi board upgrade cycle is relatively long, since the foundation launched the original Raspberry Pi Model B in early 2012. Many Raspberry Pi users in forum have expressed a desire for a new fast Raspberry Pi with upgraded features like a new processor, more memory or a GIGABIT Ethernet. However, the Raspberry Pi makers believed the earlier Raspberry Pi boards(Model B, B+) had sufficient features for users, and that improvements could raise the price. So the Raspberry Pi upgraded gradually to retails at the $35.

The advent of the Raspberry Pi is not only for people who do not require super-fast PC, but has also inspired microcomputer enthusiasts to create some amazing embedded projects. While Raspberry Pi has been a pioneer in the industry of tiny hardware for development, a rapidly increasing number of rival products now compete with Raspberry Pi. There are many single-board microcomputers available today, such as UDOO, HummingBoard, Banana Pi, BeagleBone and so forth. They have been released with fast chips and also have competed on price and performance. Similar to Raspberry Pi, most of them are designed to use on ARM architecture and run with Linux distributions. Among them, some boards offer great components and performance. In this section, some well-known microcomputers are similar to Raspberry Pi will be briefly introduced.

1.3.1. UDOO

UDOO is a powerful microcomputer based on a prototyping board that is suitable for software and embedded projects development. It was released in April 2013. The board equipped with two different types of processors: A 1GHz ARM i.MX6 Freescale processor comes with either a dual or quad-core for both Android and LINUX distributions, and the ARM SAM3X same as the chip on the Arduino Duo board, running with 1GB of on-board DDR3 RAM. The dual-core board provides Vivante GC 880 and Vivante GC 320 while the quad-core board offers Vivante GC 2000, Vivante GC 355 and Vivante GC 320 as the integrated graphics[9]. According to the summary of UDOO's specification on Wikipedia, each processor provides three separate accelerators for 2D, OpenGL ES2.0 3D and OpenVG[10]. In addition, the things UDOO beats the original Raspberry Pi in is that an UDOO has more GPIO pins, a GIGABIT Ethernet, a camera connector, and even the quad-core version has a Wi-Fi module and a SATA port. However, an UDOO is more expensive than Raspberry Pi; that extra power priced from $115 to $135. Figure 3 shows the basic layout of UDOO Quad.

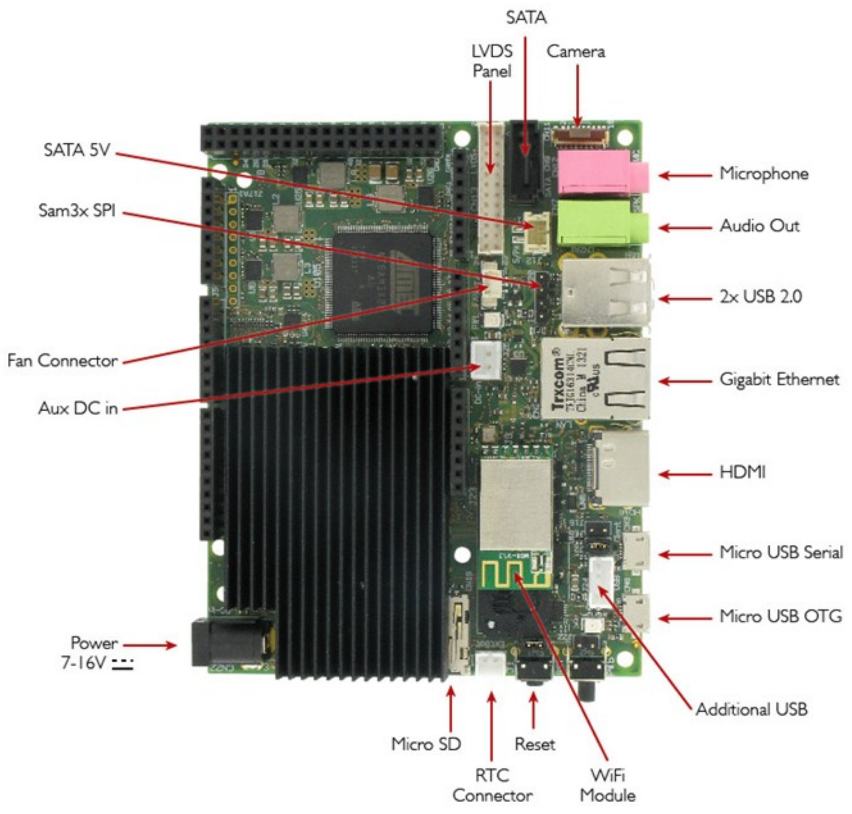


Figure 3 UDOO Quad[11]

1.3.2. HummingBoard

Another powerful alternative to Raspberry Pi is HummingBoard. Figure 4 shows the layout of a HummingBoard. HummingBoard is an ARM-based development board modelled after the Raspberry Pi. It was launched in July 2014. SolidRun Ltd manufactures and sells the boards in three models ranging from $45 to $120 with different level of configurations(components). All HummingBoard models are more powerful than the original Raspberry Pi equivalents, intended to provide better performance and flexibility to DIY projects. The board runs with a detachable 1GHz

ARMv7 processor and 512MB/1GB of RAM module rather than the built-in 700MHz ARMv6 and 512MB of memory of the original Raspberry Pi. It means users can upgrade their HummingBoard by switching a new module with a faster chip or multi-core chips and more memory in the future. The Hummingboard is also supported by a built-in graphics processor(GPU) that is capable of handling 3D graphics, or for a higher price a GPU with support for quad shading to give faster and more realistic 3D effects. The professional version of HummingBoard comes with an integrated onboard microphone and mSATA connector.

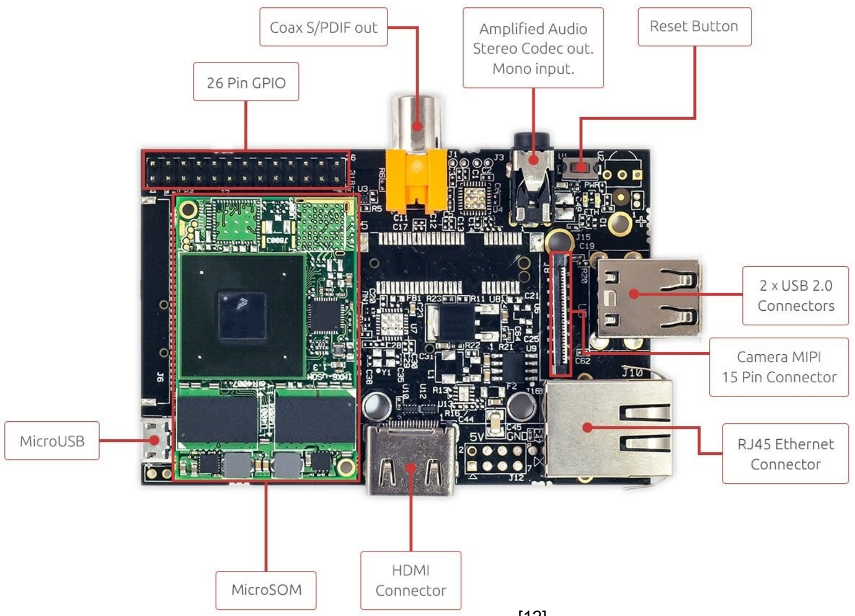


Figure 4 HummingBoard [12]

Similar to the Raspberry Pi, there are various choices of open source operating systems available to run on HummingBoard such as LINUX distributions of Ubuntu, Debian and ArchLinux, as well as Android OS and media center systems like XBMC.

1.3.3. BeagleBone Black

BeagleBone Black is a low-cost microcomputer board like the Raspberry Pi, which was released in April 2013 with $55, but more targeted to development experts and embedded project enthusiasts. Figure 5 shows the BeagelBone Black board. It has much more powerful on-board components than the original Raspberry Pi in the same era. The BeagleBone Black provides a single-core 1GHz ARM Cortex-A8 processor with 512MB of RAM. 4 GB on-board storage allows to boot with pre-installed Linux distributions or Android ROMs. Therefore, people do not have to pay extra for storage. In contrast, the drawback is that, the on-board storage also could be a limitation, which restricts the possibility of swapping operating system in a flexible and efficient way.

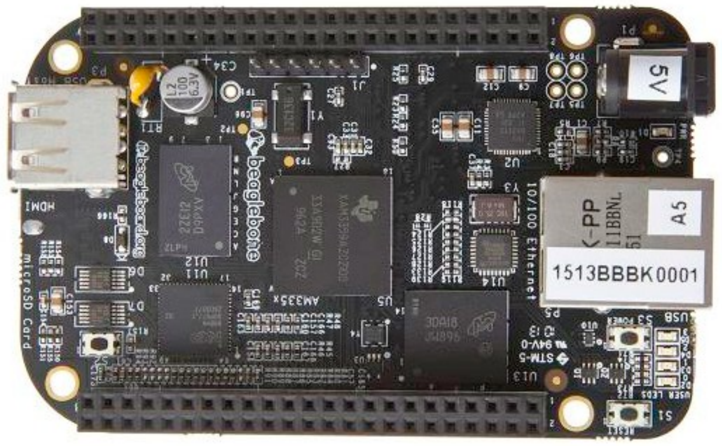


Figure 5 BeagleBone Black [13]

1.3.4. Why Raspberry Pi

At $35, the Raspberry Pi is at an affordable price for most learners and DIYers, which is a large part of its success. On the other hand, the Raspberry Pi is intended to be an educational tool rather than sold for its performance. Although it is also a much stripped down computer, the Raspberry Pi is still sufficient for many PC users who do not need a super-fast computer. The Raspberry Pi Model B generations are sufficiently capable of running a modern Graphical User Interface(GUI) based operating system(Raspbian) and performing general computing tasks.

The Raspberry Pi now has become the most famous single-board microcomputer, which is widely used in teaching situations and embedded projects. It has a professional group to maintain and keep updating the operating system to make sure Raspberry Pi can be used in a stable way for daily life and educational purpose.

1.4 Development Environment

Getting started with graphics programming, there are various applications for developing graphics, such as XNA with Visual Studio, Unity, SFML, Processing, OpenFrameworks, Quartz Composer and EMOTION. XNA is a wrapper around native DirectX graphics library, which is intended to allow developers to write 2D and 3D graphics programs with Visual Studio and run them on Windows platforms. In early 2013, Microsoft claimed that the XNA would be fully abandoned in 2014, and "the XNA Game Studio is not in active development, and DirectX is no longer evolving as a technology" [14]. Fortunately, other open source framework implementations of XNA, including MonoGame and SlimDX, continue to evolve. This project decided to base the Raspberry Pi development on XNA as it had been experienced and found it is suitable for teaching 2D and 3D graphics programs and understanding programmable graphics pipeline.

However, neither XNA Framework nor Visual Studio IDE can be directly used on the Linux-based Raspbian operating system. The MonoGame and its supported IDE "MonoDevelop"(now rebranded as "Xamarin") was first considered, because they are designed to be used for cross-platform, including Windows, OS X and most Linux distributions. Both MonoGame and MonoDevelop are built on top of "Mono" which is a cross-platform implementation of Microsoft .NET Framework providing C# compilers, Common Language Runtime and compatibility libraries for multiple operating systems. Because of Pi's limited performance, the fully featured MonoDevelop provides comprehensive facilities that cause it to load and run slowly on the Raspberry Pi. Additionally, other factors such as versions of Mono used or lack of related libraries lead it to be unstable on ARM-based Raspberry Pi and not capable of developing all C# programs. On the other hand, Wang(2014) stated in his research that MonoGame "currently does not support development on Raspberry Pi. As MonoGame is targeted to a number of different platforms, and the performance of Raspberry Pi is limited, even if MonoGame was available on Pi, it would be a little bit complicated for Pi to program with XNA. What is more, MonoGame strives for exact source code compatibility with

XNA. This is not desirable for shader coding.[15]" Therefore, in order to push the ease of graphics programming on Raspberry Pi forward, the project would be divided into two parts.

In the first part, when programming with XNA on Raspberry Pi, the ideal result is the code is similar to that on Windows system so that when people go back to Windows can easily duplicate or develop programs without extra learning costs. As the major low-level graphics functions provider, XNA requires DirectX to access graphics card(s). Figure 6 illustrates the architecture of an XNA program based on DirectX. However, DirectX is specific to the Windows system. An alternative to the DirectX API(Application Programming Interface) is needed, as well as a method of interaction between XNA-like programs and graphics system via the new alternative. This part has been completed by another student at the University of Waikato, Lichao Wang, who worked on rewriting some of the XNA classes and investigated the possibility of building and displaying graphics with a supported graphics API – OpenGL ES on the Raspberry Pi. The basic idea is taken from the JBBRXG11 project which is an open source extension of the Microsoft's XNA game development system to inherit the features that geometry shader from DirectX10 and both tessellation and compute shader from DirectX11.

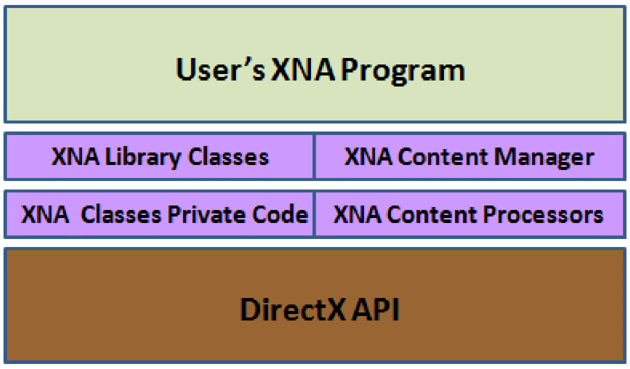


Figure 6 Architecture of an XNA Program [16]

In the other part, a modern IDE is required to ensure graphics programming on Raspberry Pi in an easy and flexible way. An integrated development environment(IDE) is a software suite provides program development facilities that typically involve a graphics user interface(GUI), source code editor, a compiler and a debugger. Some IDEs such as Microsoft's Visual Studio and Eclipse also support for multiple languages and code auto-completion(IntelliSense). Owing to lack of available IDE for XNA programs on Raspberry Pi, Lichao Wang had to start his part of the project by inputting all reference content and classes file names in the command terminal. Considering graphics libraries can be large and complex, having code auto-completion and build automation is worth the added learning effort of the IDE. So far, a number of software or systems support remote running and debugging of programs on Raspberry Pi, such as the Windows 10 IOT core OS and an extended package of Visual Studio – VisualGDB for deploying C/C++ programs from Windows to Linux platforms. However, it also means that users have to own both a Windows PC and a Raspberry Pi, which would increase the cost. Therefore, this part of the project attempted to enable users developing graphics programs on a single Raspberry Pi. Instead of looking for an existing IDE that can run with the XNA-like program on a Raspberry Pi, this project will focus on porting a lightweight IDE from Windows platform to the specified Linux distribution – Raspbian.

"Since XNA games are written for the(C#) runtime, they can run on any platform that supports the XNA Framework with minimal or no modification. Games that run on the framework can technically be written in any .NET-compliant language, but only C# in XNA Game Studio Express IDE and all versions of Visual Studio 2008 and 2010(as of XNA 4.0) are officially supported[17]." As Lichao Wang rewrote the XNA classes in the C# language, a working version of "XNA Framework" is available on the Raspberry Pi. Therefore, the target IDE must supply C# language supports, and initially include or is capable of importing a C# compiler and Common Language Runtime(CLR) to compile and run the XNA-like programs on the Raspberry Pi. As introduced above, Mono Framework provides C# compilers and the Common Language Runtime, which works on most Linux distributions. As a result, this project will port an existing lightweight open source IDE on Windows platform to Raspbian by Mono components.

A number of Windows programming environments for C# language support were considered for porting to Raspberry Pi, involving SharpDevelop, MonoDevelop, X-develop, Xacc and QuickSharp.

SharpDevelop is a fully featured open-source IDE mainly for C#, F# and VB.NET languages on the Microsoft's .NET platform. It provides debugging, code analysis, build automation, code completion, and Windows Forms designer just similar to Visual Studio. The MonoDevelop is a GTK#(Mono GUI toolkit) implementation of SharpDevelop which was ported to be cross-platform based on Mono platform.

xacc.ide is an open-sourced small IDE mainly targeted at .NET development. It supports multi-language syntax checking and compiling features.

X-develop is a commercial IDE for multiple languages and targets for cross-platform, including .NET platform, Mono platform and Java platform. It provides common modern IDE features such as code completion, syntax error checking, and so on.

QuickSharp is an open-sourced lightweight IDE targets to Microsoft's .NET platform and supports Mono platform. It provides syntax checking, error indicator, code completion for multiple languages. Additionally, the plugin-structured IDE allows users extend custom features. For example, a new program language support. The QuickSharp is completely written in C# language.

Table 2 compares above IDEs run on Windows in the aspect of whether they are capable of compiling and running with Mono, lightweight, open-sourced, error message display, code completion(IntelliSense) and the requirement of extensibility was included to allow the possible addition of compile time content management to the IDE.

	Lightweight	Mono	Open Source	Error Message	IntelliSense	Extensibility
SharpDevelop	✗	✓	✓	✓	✓	✓
MonoDevelop	✗	✓	✓	✓	✓	✓
Xdevelop	✓	✓	✗	✓	✓	✓
Xacc	✓	✓	✓	✓	✗	✗
QuickSharp	✓	✓	✓	✓	✓	✓

Table 2 comparison of C# support IDEs

Eventually, QuickSharp was chosen as the smallest, and most likely to work on Raspberry Pi. To sum up, the ultimate goal of this part of the project aims to port QuickSharp IDE from Windows platform to Raspberry Pi and extend the IDE to be able to process XNA-like program. The target IDE runs on Raspberry Pi is called “PiSharp”.

Chapter 2. Literature Review

The purpose of this project is to allow people to develop and learn XNA-like 3D graphic programming. This chapter introduces the previous research on implementing and extending XNA classes. It will first describe an open-sourced extended version of XNA on Windows platform, which has developed at the University of Waikato, and follows by describing the other part of this project that porting XNA classes to be compatible with Raspberry Pi, which developed by another student — Lichao Wang.

2.1 JBBRXG11

"XNA is a freeware set of tools with a managed runtime environment provided by Microsoft that facilitates video game development and management.[18]" It was first released in 2006 and the latest version of XNA(XNA 4.0) is built on top of Microsoft's native DirectX 9 graphics library. For this project, an XNA-like toolset built by Lichao Wang was used. It was based on an earlier project called "JBBRXG11". According to the project description of JBBRXG11 on the CodePlex, JBBRXG11 is a project developed at the University of Waikato, attempted to extend an open-sourced implementation of XNA(SlimDX) to access DirectX 10 and DirectX 11 libraries, and use the features in particular geometry shaders, hull shaders and domain shaders for tessellation, and compute shaders. In 2011, Bill Rogers started the project and built the first version by XNA 3.1 and DirectX 10, which was unofficially called "SlimDXna", to allow using geometry shaders in XNA program. A year later, James Boud ported the JBBRXG11 project to XNA 4.0 and also DirectX 11 for tessellation and compute shaders uses. [19]

JBBRXG11 project attempted to convert the all XNA 3.1 classes within the open-sourced SlimDX Framework to XNA 4.0, and enable user's graphics program access DirectX 11 methods and forwards to the graphics card(s). However, it is not completed. Although the JBBRXG11 toolset still partly relies on XNA 3.1, it demonstrated how an XNA framework interacting with low-level APIs and inspired the approach that porting XNA classes to another platform for Lichao Wang's project. Figure 7 shows the JBBRXG11 project architecture.

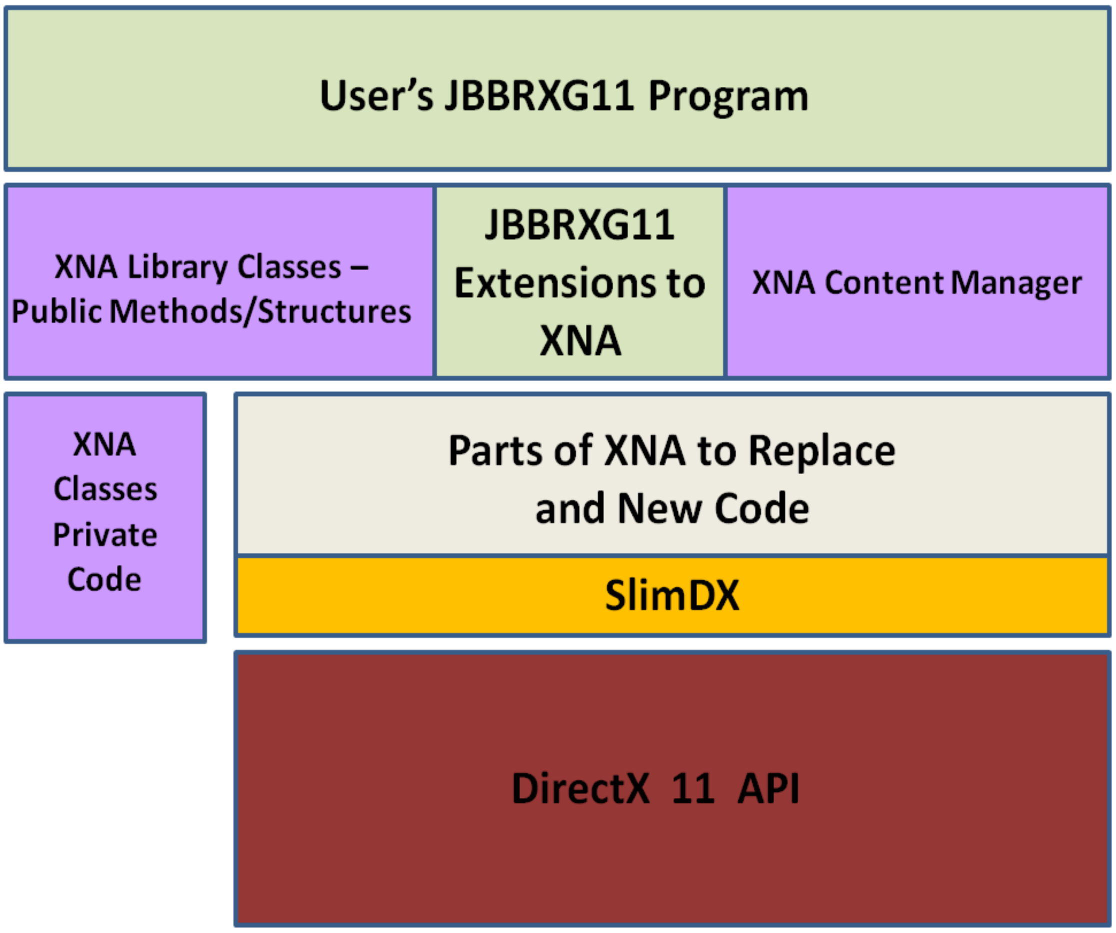

User's JBBRXG11 Program
XNA Library Classes – Public Methods/Structures
JBBRXG11 Extensions to XNA
XNA Content Manager
XNA Classes Private Code
Parts of XNA to Replace and New Code
SlimDX
DirectX 11 API

Figure 7 Architecture of a JBBRXG11 Program [16]

2.2 Pi-XNA

As explained in section 1.4, this project has been divided into two parts. This section introduces the basic idea of Lichao Wang's contribution to the entire project. His project(also called "Pi-XNA" project) focused on investigating whether developing XNA-like programs on the Raspberry Pi was feasible. Meanwhile, the project tried to rewrite some parts of the fundamental XNA classes in order to allow XNA-like program run on a Raspberry Pi. Because the DirectX is not available on Linux distributions, in particular, Raspbian, OpenGL ES 2.0 that supported by Raspberry Pi's GPU is used to replace the DirectX providing the low-level graphics APIs. Figure 8 illustrates the architecture of a typical graphics application running on Raspberry Pi. In the Pi-XNA project, OpenGL ES 2.0 and EGL APIs are both required in the modified XNA classes to get accessed to the graphics system. EGL(Embedded-System Graphics Library) "is an interface between rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system"[20]. In Pi-XNA project, the EGL library is mainly responsible for interacting with native Raspbian X Window system, initializing EGL data structure, handling graphics context management and drawing surface and binding frame buffer binding for rendering.

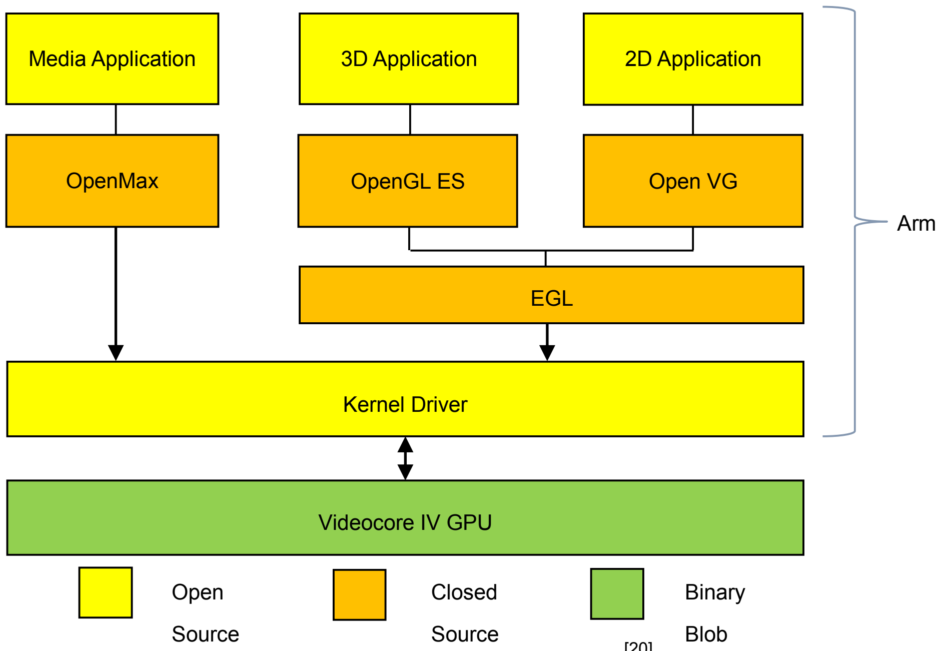


Figure 8 Raspberry Pi software Architecture[20]

Wang(2014) pointed out "When programming with the modified classes on Raspberry Pi, users' programs are similar to XNA programs on Windows, but the underlying classes interact with graphics systems through the OpenGL ES 2.0 library[15]". Figure 9 shows the Pi-XNA program architecture.

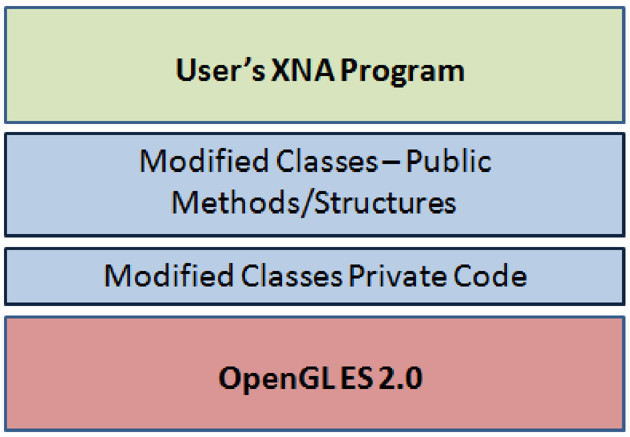

User's XNA Program
Modified Classes – Public Methods/Structures
Modified Classes Private Code
OpenGL ES 2.0

Figure 9 Architecture of a Pi-XNA Program [15]

OpenGL ES is a cut-down version of the widely used OpenGL graphics API because it was designed for rendering 2D and 3D graphics on the limited performance of embedded systems. For example, ARM-based Raspberry Pi. Unlike DirectX bindings to the Windows platform, OpenGL is an open source and cross-platform graphics API to interact with a GPU. OpenGL is widely used on Linux distributions, Windows, OS X, and other platforms.

The Pi-XNA project used Raspberry Pi supported OpenGL ES 2.0 library. The early versions OpenGL ES 1.x are implemented by OpenGL 1.5 specification provided Fixed Function Pipeline. The OpenGL ES 2.0 replaced the Fixed Function Pipeline with a programmable pipeline that enabled developers programming on embedded system can customize rendering effects by altering pixels, vertices and texture in shaders. Figure 10 and Figure 11 show the Fixed Function Pipeline and OpenGL ES 2.0 programmable graphics pipeline.

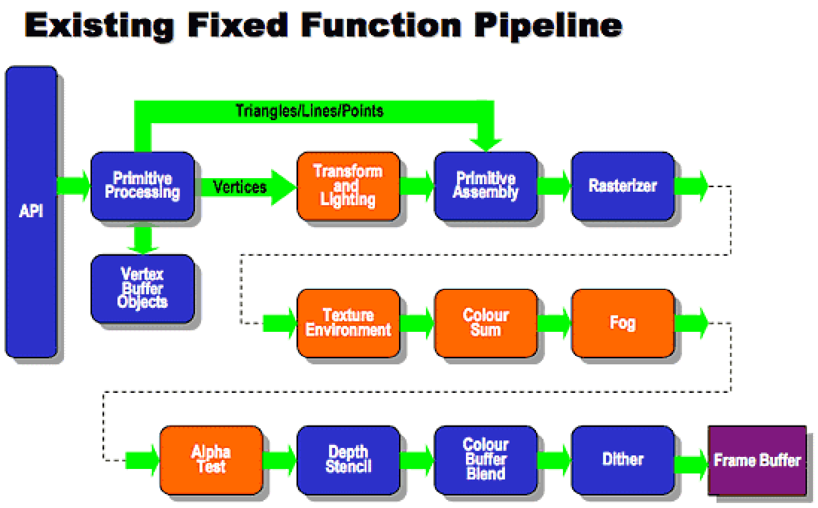


Figure 10 Fixed Function Pipeline [21]

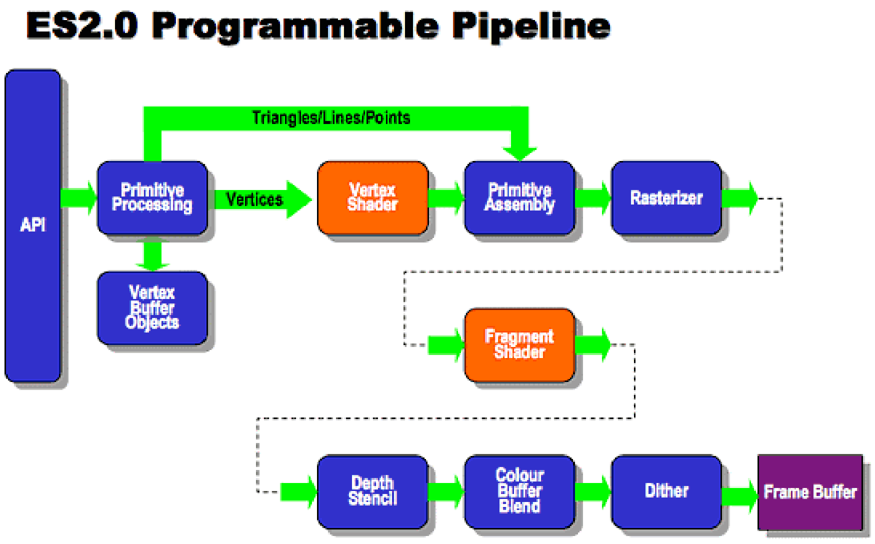


Figure 11 OpenGL ES 2.0 Programmable Pipeline [21]

Finally, the Pi-XNA project successfully built a set of graphics classes that allows an XNA-like program to compile and produce textured 3D models animation and displayed with lighting on screen. Although the Pi-XNA did not port the entire XNA classes to Raspberry Pi and tiny differences existed between the XNA and the Pi-XNA(e.g. the XNA uses HLSL as the shading language while the Pi-XNA uses GLSL), the project showed the possibility of writing XNA-like programs on the Raspberry Pi. Additionally, it implemented a number of shading effects and mathematical calculations.

Chapter 3. Underlying System

This chapter introduces the background of platforms and crucial components that support porting a PiSharp application, and then respectively explains how they are applied in this project. The subsections start with an introduction of the target Linux operating system running on Raspberry Pi – Raspbian. After that, the second section discusses the detail of QuickSharp application, which is ported to Raspberry Pi, in the aspects of QuickSharp's architecture on Windows and the features of this development environment. Finally, this chapter presents the primary parts of alternative components including Mono, GTK+ that used in the PiSharp project.

3.1 Raspbian

Raspbian is an official support operating system for Raspberry Pi devices. It is optimized and maintained by Raspberry Pi Foundation based on a Debian Wheezy hardware floating point version. The first version of Raspbian was accomplished by Mike Thompson and Peter Green in 2012 and now becomes to the most widely used operating system on Raspberry Pi. The latest version of Raspbian has been switched the desktop environment from LXDE(abbreviation for Lightweight X11 Desktop Environment) to a Pi-specific version – PIXEL(abbreviation for Pi Improved X-Window Environment, Lightweight). According to Long published on Raspberry Pi blog and responded in comments, the PIXEL desktop environment involves a modified LXDE desktop environment with the Openbox window manager configuration settings[22]. The new desktop environment made the most efforts on GUI-level appearance optimization and new applications addition. In PiSharp project, Raspbian is chosen as the operating system that supports low-level GUI environment for the ported IDE. Figure 12 shows the Raspbian Desktop based on LXDE.

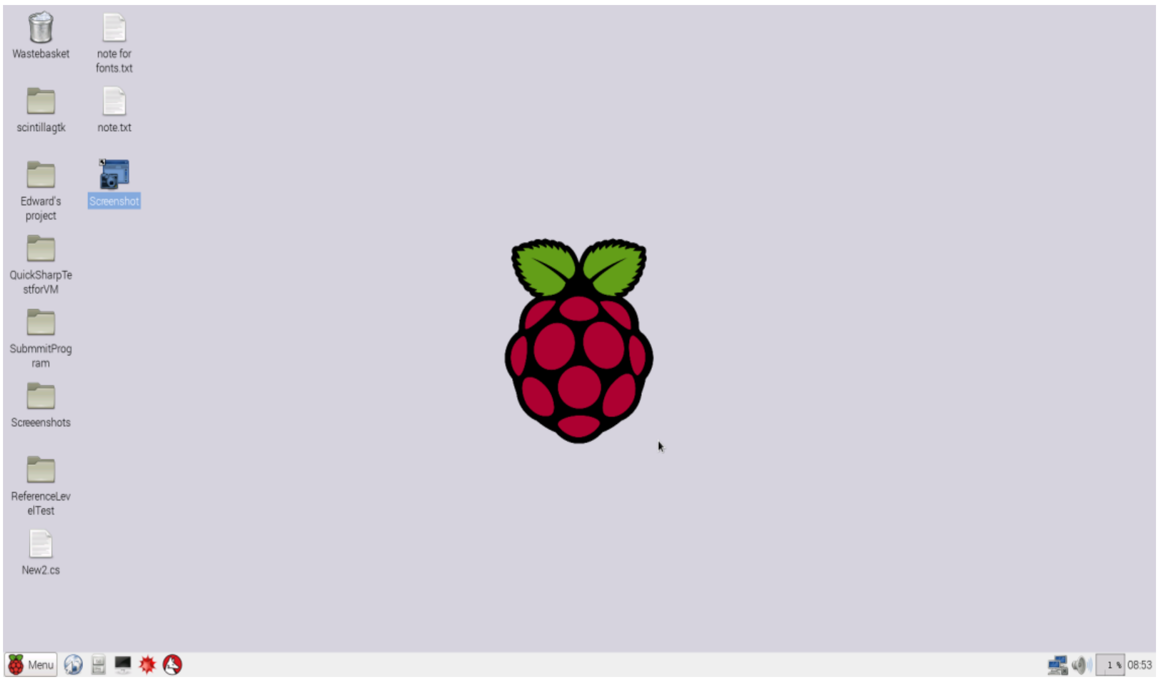


Figure 12 Raspbian Desktop(LXDE)

The X Window system is commonly used in UNIX-like system only provides low-level graphic user interface(GUI) framework such as graphics primitives input device interaction, drawing window on screen and that allows other desktop environment build on top. The LXDE is a desktop environment implementation based on X11 and uses GNOME/GTK+ GUI toolkit(GNOME is also a desktop environment built on top of the X11).

The X Window system consists of hardware-level components(X server), application-level components(X client), the communication protocol – X Protocol. The X server runs on local computer as a graphics resource provider and input devices event listener, which is a core of a GUI system. On the other side, X client applications can run on local or remote computer to communicate with X server for processing the user's events. For example, redraw and display on screen. A simple example of X Window system is shown in Figure 13.

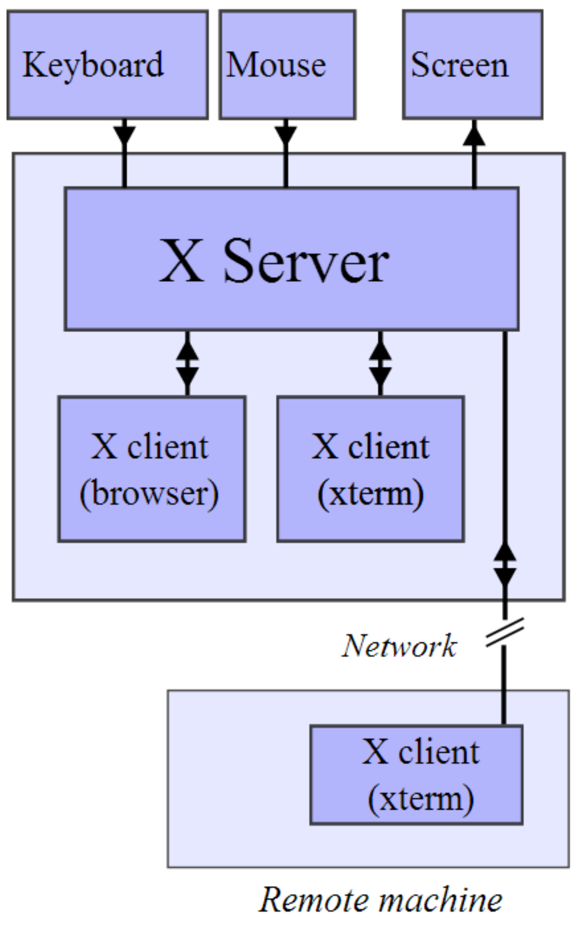


Figure 13 Simple example of X Window System[23]

3.2 QuickSharp

3.2.1. Basic Layout and Architecture of QuickSharp

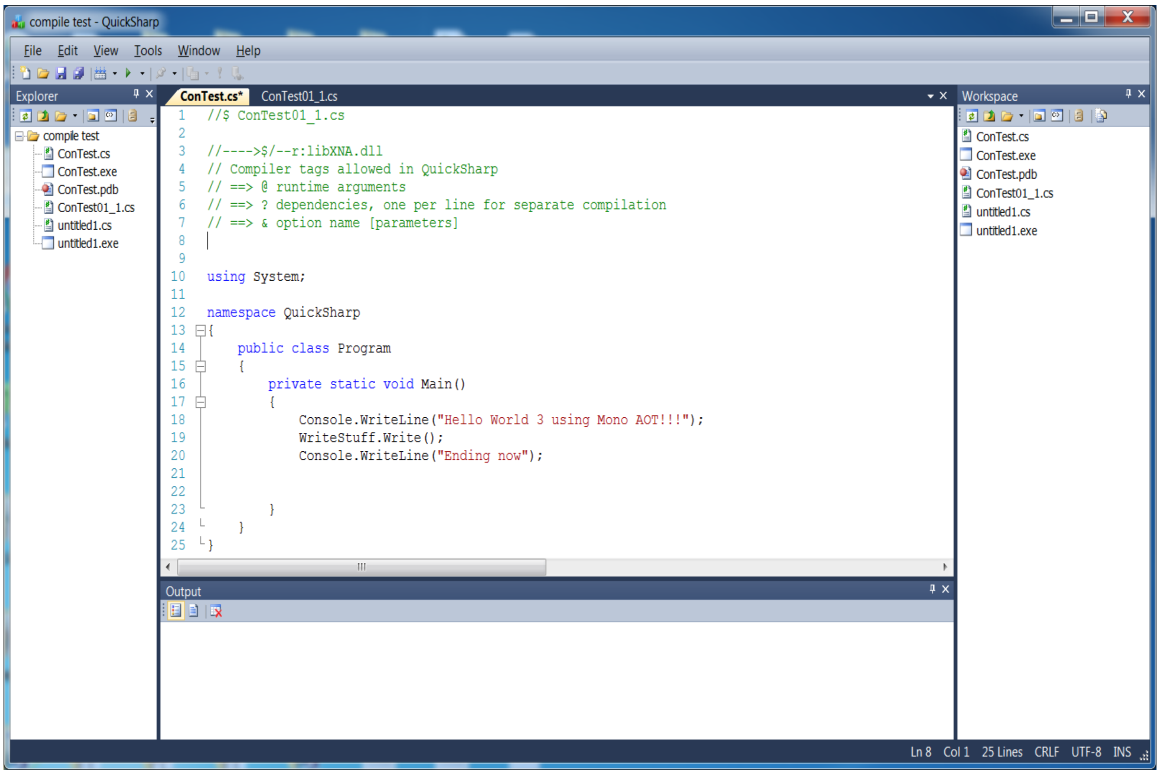
Owing to the restrictions of processor performance, memory capacity and data storage on the Raspberry Pi, a lightweight development environment with a simple development approach is required as the target IDE. The development environment needs to be open-sourced and compatible with Mono runtime so that it could be port to a LINUX distribution – Raspbian. Moreover, intending to easily write, compile and run XNA-like programs, features like build automation, error message display, debugging and code completion(IntelliSense) are expected to be involved in the development environment.

As introduced in Section.1.4, QuickSharp was chosen to be ported to Raspbian. QuickSharp is an open-sourced lightweight IDE targeted to Microsoft's .NET platform. More importantly, the QuickSharp and its some significant components are open source and built on top of Microsoft's .NET Framework 3.5 Windows Forms GUI toolkit. Despite the Window Forms bindings to native Win32 API in managed code on Windows platform, Mono components ported it to cross-platform and provides the Mono version of Windows Forms library. It means the QuickSharp IDE itself is possible to be compiled and test on the Mono platform.

According to the QuickSharp documentation, the open-source components are:

- DockPanelSuite project: .NET-based project provides a Visual Studio-style windowing framework;
- Scintilla: a Win32-based library provides the core of the text editor;
- ScintillaNET: a .NET based wrapper around Scintilla library allows Scintilla to be used within a .NET application as a common Windows Forms control;
- SharpZipLib Library: provides zip and general archive file management;
- SQLite and System.Data.SQLite Libraries: provide database management.[24]

Because QuickSharp uses the docking feature from DockPanelSuite's WinFormsUI project to manage windows, it presents a Visual Studio theme-like layout that two file manager windows are respectively docked to the sides; an output window is docked to the bottom; then opening files in text editors are filled in the rest of client window. The basic layout of the QuickSharp IDE is shown in Figure 14, and Figure 15 illustrates the architecture of QuickSharp.



```
//$ ConTest01_1.cs

//---->$/--r:libXNA.dll
// Compiler tags allowed in QuickSharp
// ==> @ runtime arguments
// ==> ? dependencies, one per line for separate compilation
// ==> & option name [parameters]


using System;

namespace QuickSharp
{
    public class Program
    {
        private static void Main()
        {
            Console.WriteLine("Hello World 3 using Mono AOT!!!");
            WriteStuff.Write();
            Console.WriteLine("Ending now");


        }
    }
}
```

Figure 14 QuickSharp Basic layout

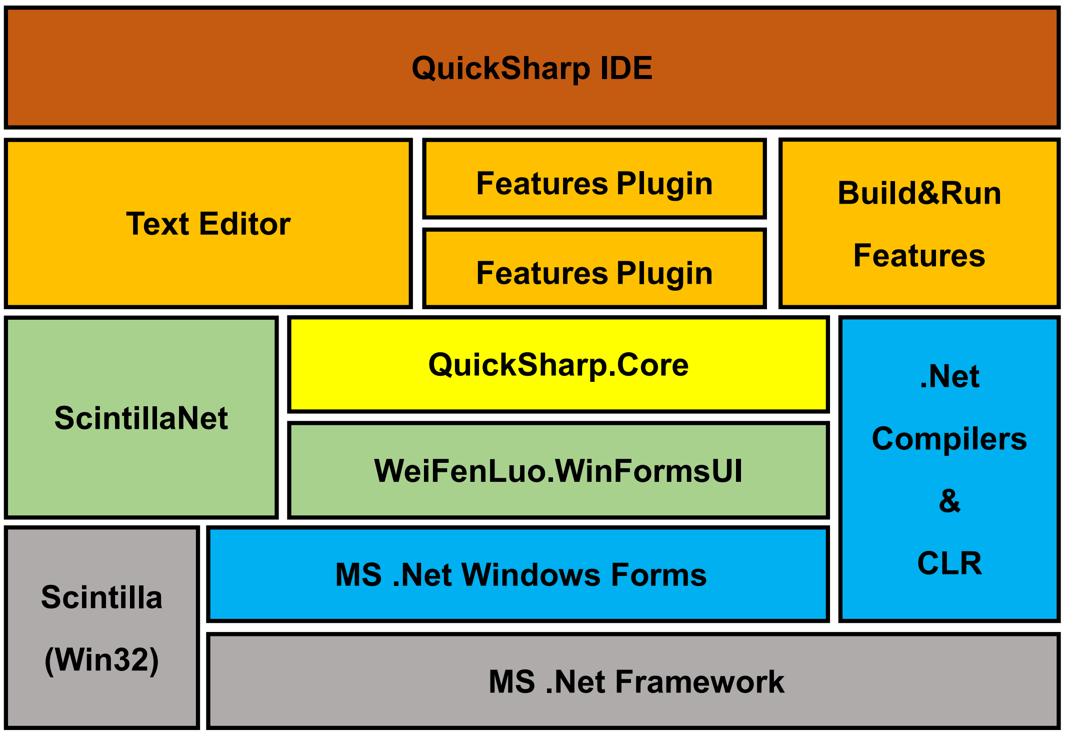


Figure 15 Architecture of QuickSharp IDE on Windows

Microsoft .NET Framework is one of Common Language Infrastructure(CLI) implementations together with libraries to access the windowing libraries. The CLI defines an international standard specification of execution and runtime environments in which “applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those applications to take into consideration the unique characteristics of those environments.[25]” QuickSharp IDE relies on the .NET compilers and Common Language Runtime(CLR) for its multiple programming languages support.

Unlike other modern IDEs, the QuickSharp IDE is designed to allow the rapid creation, compilation and execution of C# programs without using projects or solution files to manage all items, such as source files and references, and produce a single binary file. Instead, QuickSharp IDE prefers to develop each file individually “to give maximum flexibility in structuring projects and is designed to make the most of the component-based architecture of .NET[24]”. However, this method is neither convenient to beginners nor XNA-like programs which involve a large number of graphics classes and mathematical classes. Therefore, PiSharp project decided to reorganize the build process in QuickSharp.

As shown in Figure 15, the QuickSharp features are presented as a modular architecture that ensures a flexible and efficiently extensibility by adding or remove plugins. As all features plugins' carrier, "QuickSharp.Core" provides fundamental facilities for entire QuickSharp IDE. Literally, it is the core of the QuickSharp IDE, which provides the interaction and appearance management on main user interface and most pop-up window, file operations, user profile and plugins management.

The QuickSharp contains a great number of plugins mainly consisting of following several parts: QuickSharp.BuildTools, QuickSharp.Language, QuickSharp.Editor, QuickSharp.TextEditor, QuickSharp.OutPut, QuickSharp.Explorer, QuickSharp.WorkSpace, QuickSharp.CodeAssist.

"QuickSharp.BuildTool" provides the default Compile and Run tool menu and toolbar entries, but also provides its own extensible tools framework via a BuildToolManager plugin. This plugin provides the infrastructure for languages only that support compile and run actions. Specific tools and languages support require additional plugin – QuickSharp.Language.*.

"QuickSharp.Language" is a collection of language support plugins. It is catalogued by different type of programming languages. The QuickSharp IDE supports a number of programming languages including ASP.NET, C#, Dbml, Html, JavaScript, VB.NET, Wsdl and Xml. Each language component defines the document handling and at least one default build tool and build command configuration. It then associates with the "QuickSharp.BuildTool" component to accomplish the entire compile or run operation deployment.

"QuickSharp.CodeAssist" is collection of code completion(IntelliSense) for specific languages. It makes it easier for developers to use various program entities such as members, classes and namespaces, avoiding the need to remember detail and it avoiding spelling errors. This allows program entities to be listed for insertion into the editor by invoking the code assist pop-up window. The pop-up window presents a list of the class or members, as appropriate.

“QuickSharp.Editor” associates with “QuickSharp.TextEditor” to present a container required a Scintilla-based control. The complete text editor can do a number of text editing behaviors, such as “cut”, “copy”, “paste”, “undo”, “redo”, “editor find/replace”, “file save” and so on. It maintains the association between registered document types and the corresponding LEXER. A language represents a specific document type recognized by the editor and is mapped on to an internal LEXER. The LEXER(lexical analyzer) determines how the editor interprets the structure of a document, the language determines how the structure is presented(such as syntax coloring and fold points.).

“QuickSharp.Output” provides an output window that output the result from the programs that compile or run within QuickSharp IDE. The text content presents into two forms of view: a list view and a text view. The list view output is used to indicating and locating error messages in editors. Moreover, the actual procedure call of building and running is accomplished in the QuickSharp.Output plugin.

“QuickSharp.WorkSpace” is a plugin window that shows the current selected directory and provides file management services, such as file open, rename, copy, delete, compress, move to other folder and so forth.

“QuickSharp.Explorer” is a plugin window that shows a subset of the disk file system. The explorer provides file management services are very close to the workspace. Both explorer and workspace are very useful in a multi-file project development case.

3.2.2. Scintilla&ScintillaNET

Scintilla is an open source text editing component, written in C++, intended to be used in a part of a program. The Scintilla component supports for multiple platforms such as Windows, OS X with COCOA, QT and LINUX distributions with GTK+. The API provides advanced features for editing and debugging source code in multiple programming languages including syntax highlighting, searching, replacing, error indicators, and so on. There are a number of projects which use the Scintilla component. In particular, text editors and development environments, including Notepad++, SciTE, Wing IDE, LSW DotNet-Lab(LSW-DNL) and PythonWin[26].

As a derivative of Scintilla, ScintillaNET is a wrapper for Win32-based Scintilla component written in the C# language, which can be used as general controls in .NET Windows Forms applications. ScintillaNET allows the use of text editing features from Scintilla and provides some additional features, like multiple key-command bindings.

QuickSharp did not develop its own text editor. Instead, it uses "ScintillaNET" project. "ScintillaNET" is based on a famous source code editing component – Scintilla. ScintillaNET is the .NET implementation of wrapper around the Scintilla component. ScintillaNET supports a pre-built text editor with syntax highlighting and many other features like multiple key-command bindings to application or IDE. The LEXERs which used by QuickSharp's editor are the part of Scintilla that provides a means of distinguishing what a piece of text is in the context of a language.

3.2.3. WeiFenLuo.WinFormsUI

WeifenLuo.WinFormsUI library is an open-sourced alternative to Visual Studio themes which provides docking window layouts. It is built on top of the .NET Framework Windows Forms. The Dock panel WinFormsUI project was first released by Weifen Luo in 2006. In QuickSharp's GUI system, the docking library not only docks opening text documents to each other, but it is also responsible for docking all non-editor windows contained in the main window, including Explorer, WorkSpace and Output windows.

According to the DockPanelSuite documentation, WinFormsUI library support several dock style, such as DockLeft, DockRight, DockTop, DockBottom, Dock*AutoHide, Document, Float and Hidden.

- DockLeft, DockRight, DockTop, DockBottom: the on-screen window dock to the appropriate boundary of the multiple-document interface(abbr. MDI) container window;
- DockLeftAutoHide, DockRightAutoHide, DockTopAutoHide, DockBottom-AutoHide: similar to above style but hide just showing a tab
- Document: fix the window in the middle of the MDI container and showing with a little tab on the top
- Float: apart from the MDI parent, but can be docked into the container by users
- Hidden: make the dockable window is invisible[28]

An example of docking window system uses method is demonstrated in later Section 5.1 The DockPanelSuite project(docking window) primarily aims to provide a Visual Studio style of use for Windows applications. However, a number of docking functions or related methods requires to get access to native Win32 calls, such as DragDetect and SendMessage. Those invokes are not available on Mono/Linux platform. The DockPanelSuite maintainers efforted to allow the WinFormsUI to be compatible with Mono platform by sacrificing some Windows native features. For example, the drags and drops. More detail of altering WinFormsUI project to port to Mono/Linux platform will be described in Chapter 4 and Chapter 5.

3.2.4. Embedded options

Most commonly used IDEs, for example, Visual Studio, utilize text-based “project” files or “solution” files to describe and organize the information, relations and configurations that are required while building a project. In contrast, according to the QuickSharp documentation, QuickSharp's build module is based around each source file producing a corresponding output file. The QuickSharp documentation claims that "the IDE intends to bring maximum flexibility in structuring projects and is designed to make the most of the component-based architecture of .NET, dividing a project into multiple output files allowing the QuickSharp build system to be optimized enabling(only) changed files to be recompiled"[24]

QuickSharp has a build tool configuration system that can be used to set some compiler options. However, to specify which files and libraries should be used in a compilation QuickSharp requires the user to set "embedded options" which are formatted comment included in C# source files. The QuickSharp documentation includes the following sections describing the system.

> QuickSharp's build tool configuration allows compiler settings to be customized at a general level but occasionally it is necessary to provide options for a specific program. For example, there may be a need to reference a library or include a resource in a program. Embedded file options allow build processes to be configured for a program by 'embedding' the options directly in the program's source files. Embedded options are specially formatted comments that are ignored by compilers but are recognized by QuickSharp as additional build instructions. There are two types of embedded options: compiler options and runtime options.
>
> **Compiler options**
>
> Compiler options allow additional configuration information to be passed to the compiler(or any build tool) and are formatted as follows:
>
> ```
> //$ options here
> ```
>
> The '//$' indicates the start of a compiler option and must be included without any spaces between the opening comment and the '$'. The text following the '//$' will be passed to the compiler as part of a build tool format string in place of the macro ${EMBEDDED_OPT}. Compiler options can be included in a source file and will be passed in the order they appear.

Typical uses for compiler options are to reference libraries, embed resources or include additional source files in the compilation.

```
//$/r:Mono.Data.sqlte.dll
//$/res:myapp.resources
//$ file2.cs file3.cs
```

Runtime options

Runtime options allow runtime arguments to be passed to a program when run from within QuickSharp and follow a similar format to compiler options:

```
//@ arguments here
```

Unlike compiler options, only one option string will be passed to a program; the first one found will be used. The text following the '//@' will be included in the build tool format string in place of the macro ${RUNTIME_OPT}.

The QuickSharp IDE is intended to serve as a thin wrapper around the .NET Framework tools. By providing an editing environment and tool integration, it attempts to ease the process of working with these tools without attempting to hide them behind visual editors or wizards. Consequently, QuickSharp is not really intended as a beginner's tool, but for experienced developers. QuickSharp provides a lightweight environment for experimentation.

Therefore, in this project, porting the QuickSharp IDE to Raspberry Pi is not the only task, but there is also a need to simplify the users' processes to make sure beginners can easily start, such as optimize a project creation process; automatically add reference files or libraries to a project while compiling and running process, and so forth.

Overall, with this improvement the QuickSharp IDE seems satisfactory as the target IDE that can be used to develop and manage XNA programs on a Raspberry Pi.

3.3 Mono

On 12 November 2014, Microsoft announced that it was open sourcing the server-side .NET stack in the areas of ASP.NET, the .NET compiler, the .NET Core Runtime, Framework and libraries, and making it available to run Linux and Mac OS platforms[28]. Unfortunately, a significant infrastructure of the QuickSharp's GUI system, the Windows Forms class library, is contained in the client-side .NET stack for which Microsoft had no plan for open-sourcing and supporting different platforms. To port the .NET-based QuickSharp application to the Raspbian OS, an alternative to the .NET Framework work on Linux was urgently needed. A widely used Common Language Infrastructure(CLI) implementation on Linux systems is "Mono", which is usually considered to be a clone of the .NET Framework. According to Mono's documentation, Mono is an open-source implementation based on the .NET Framework, developed by Xamarin, enabling C# and other .NET language developers to create and run .NET applications cross-platform. It was first announced by Ximian in the middle of 2001. After several acquisitions, the primary maintainer and sponsor of the Mono project – Xamarin officially became a subsidiary of Microsoft Corporation in early 2016. Along with advances of the Mono project, Mono now supports running on multiple operating systems including Android, most Linux distributions, OS X, Microsoft Windows, Solaris, and a variety of CPU architectures including x86, x86-64, ARM and PowerPC[29].

3.3.1. Mono Components

As described in the Mono documentation, the Mono project provides several crucial components for C# developers:

- Mono's compilers for the C# programming language;
- Mono Runtime is designed to implement CLI specification and is compatible with the .NET Common Language Runtime;
- Microsoft .NET Framework compatibility Class Library which enables Microsoft .NET applications to port to Linux. This set of library comprises Base Class Library, ASP.NET, ADO.NET, Windows Forms and other class libraries that provide most .NET Framework functionality;

- Mono Class Library extends functionality outside of Microsoft .NET Framework stack and provides platform-specific class libraries such as GTK# for GUI development, Mono.Cairo for 2D graphics drawing, SharpZipLib library for zipping files, database libraries and Unix integration libraries(e.g. Mono.Posix) [28].

Among them, both Pi-XNA project and PiSharp project reference parts of the .NET Framework Compatibility Class Library, which allows the projects to be developed in a .NET-like programming environment and support .NET Framework functionality. For example, Pi-XNA and PiSharp both require the System.IO namespace for file stream manipulation. Pi-XNA can load texture information from an existing picture and PiSharp intends to read and write C# source file. In addition, PiSharp uses Mono's libraries, offering Windows Forms for GUI systems, file compression for the WorkSpace and data support for the SQLite database engine. The details of using these libraries in PiSharp project will be discussed in Chapter 4 of Issues porting QuickSharp to Raspbian.

The objective of the Pi-XNA program is to implement graphics and mathematics C# classes on the Raspberry Pi for XNA programming. It attempts to provide equivalent functionality to the XNA API from Windows, on the Raspberry Pi. Technically, the Pi-XNA project used for standalone Mono C# compiler and runtime. The C# compiler and runtime are in charge of building and testing XNA sample programs. Additionally, the Pi-XNA project takes advantages of Mono Runtime interoperability, C#'s capability to bind to native code on a platform, and access native OpenGL ES and EGL libraries, and low-level Linux windowing functionality in code.

In the PiSharp project, Mono's C# compiler and runtime are not only participating in building and running on PiSharp application itself, but also being embedded within the PiSharp IDE as third-party tools, to allow C# programmers to easily create and run their XNA programs. Whilst the primary target is Raspbian on the Raspberry Pi, it should be able to run on most Linux distributions and other Mono-supported platforms.

Historically, the Mono project has developed four C# compilers: gmcs, scms, dmcs and mcs. Each compiler references a set of the corresponding .NET libraries. Firstly, the "gmcs" compiler was implemented by C# 3.0 specification and references the libraries within .NET 2.0 and .NET 3.5. Secondly, the "smcs" compiler also aimed at the C# 3.0 specification, but only referencing a subset of .NET 3.5 assemblies, which are primary for creating Silverlight and Moonlight applications. After .NET Framework 4.0 was released by Microsoft, the "dmcs" compiler was built to support the C# 4.0 specification and reference the .NET 4.0 API. Last but not the least, the "mcs" compiler was first developed to reference .NET 1.0 and implement C# 1.0 and parts of the C# 2.0 and C# 3.0 specifications. Now evolved to be the unified Mono C# compiler and has replaced all of the above compilers since Mono 2.11.x.[28] In the PiSharp IDE, the "mcs" compiler is set as the default compiler for all C# program, although the IDE allows users to add and configure other Mono compilers.

The Mono Runtime involves a Just-in-Time(JIT) compiler which is used to read the bytecode produced by the .NET compiler and translate it into native code for use in the CLR process; By using System.Runtime.InteropServices assembly, managed code can directly use functions within from unmanaged code. Unmanaged code means source code written in C, C++, Visual Basic or other high-level programming languages compiled straight to machine code. In contrast, managed code(which generally refers to source code written in .NET languages) compiles to Common Intermediate Language(CIL bytecode) via a CLI compiler at compile-time and is converted by the JIT compiler into machine code at run-time.

3.3.2. Mono on Raspberry Pi

As has been described in the Raspbian section, Raspbian is an operating system which is a hardware floating point port of Debian wheezy for the Raspberry Pi and similar devices that use ARMv6 processors. However, before Mono version 3.2.7, which released in February 2014 and supported the hardware floating point Application Binary Interface(ABI) on ARM, the most common versions did not work with the hardware floating point version Raspbian. Therefore, .NET developers had to run software floating point Mono components on the older Debian Squeeze Linux distribution for Raspberry Pi giving computation speed far below expectations. This project looks forward to getting relatively fast compilation and running of XNA programs with the high quality of Raspberry Pi's capability. To solve this problem, Wang(2014) used a modified Mono components package in his thesis work, which worked on the original Raspberry Pi Models(Model-B and B+) based on ARMv6 Hard-Float[15]. As the Raspberry Pi 2, Raspberry Pi 3 and new Mono versions were launched halfway through this project, the project started with a Raspberry Pi Model B running Mono 2.11.4 runtime for ARMv6 Hard-Float then gradually moved to Mono 4.1.0 runtime platform within Raspberry Pi 2. Appendix A demonstrates the Mono components set up to Raspberry Pi Model B and Raspberry Pi 2.

3.3.3. Mono's GUI toolkits

The Mono project ships with three GUI toolkits, so as to provide support for different platforms: Mono WinForms, GTK# and MonoMac. MonoMac is dedicated to providing a GUI toolkit for .NET/Mono developers create and run windowing applications on OS X. The toolkit is only available on OS X or iOS because it binds to Apple's native COCOA API. In contrast, both GTK# and Mono WinForms are designed as cross-platform GUI toolkits.

GTK# is a managed wrapper around native GTK+ API written in the C# language. The GTK+ API provides a group of graphical widgets to create graphical user interfaces(GUIs) for applications. Using the GTK# toolkit enables native GTK+ GUI applications to run within the .NET/Mono Framework. The MonoDevelop IDE is a GTK# implementation of SharpDevelop IDE running on the Mono platform. Meanwhile, MonoDevelop supports the creation of GTK# applications with a visual designer called "Stetic" which looks like the Windows Forms designer in Visual Studio. Figure 16 shows the layout of the Stetic designer within MonoDevelop. The PiSharp project does not use

GTK# as it is not necessary. Instead, PiSharp implements its own wrapper to access Scintilla directly.

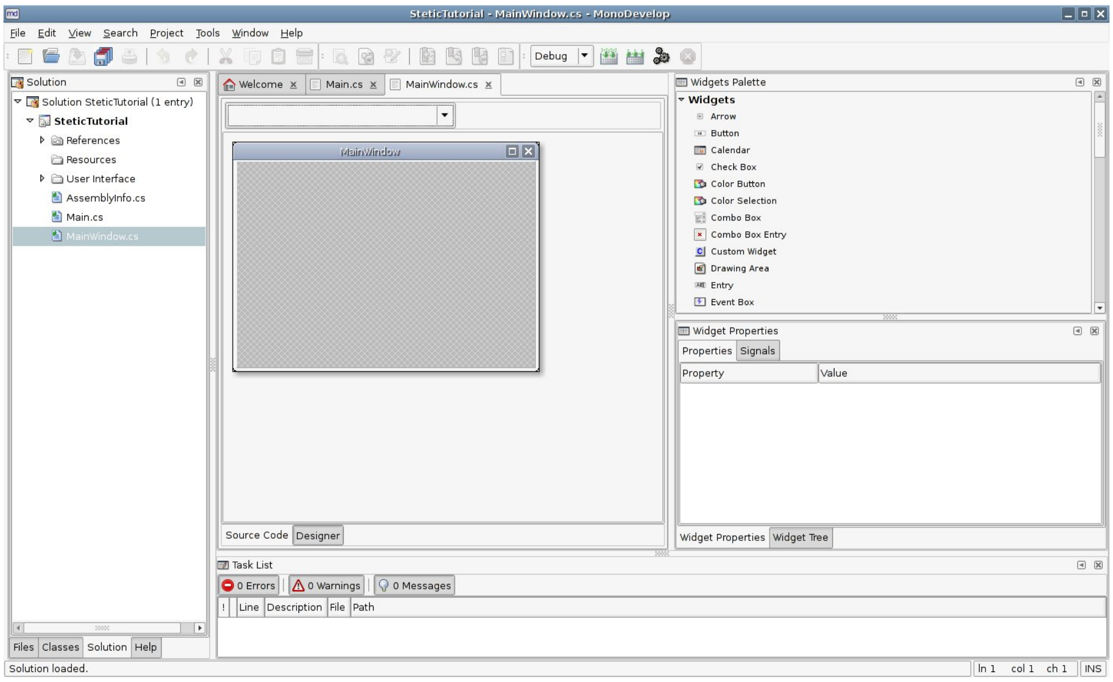


Figure 16 Stetic designer layout[30]

The Mono project provides a set of cross-platform .NET Framework compatibility Class Libraries that allows the .NET wrapped native Windows API to be ported to Mono platforms. One of the significant implementations is “Mono WinForms” providing a GUI toolkit for Mono desktop applications as an alternative to the Microsoft’s .NET Windows Forms API. With the .NET Framework compatibility stack, most .NET Windows Forms applications can directly work on the Mono platform.

To verify Mono’s compatibility, this project started with a sample program built by Visual Studio. Figure 17 shows the layout of a simple .NET-based Windows Forms application which simulates a multi-file text editor running on Windows. In this sample application, a new Tab page with a RichTextBox control can be added to the window by clicking “New” within the “File” drop-down menu. Meanwhile, an item with the same names as the Tabs is listed in the TreeView control docked on the right of the window. Tab pages can be selected by simply clicking the corresponding item in the TreeView control. The sample program was built using Microsoft’s Visual Studio IDE on the Microsoft .NET

Framework. Figure 18 and Figure 19 illustrate the graphical user interfaces directly running the executable file from Visual Studio with Mono on Windows and Raspbian OS respectively.

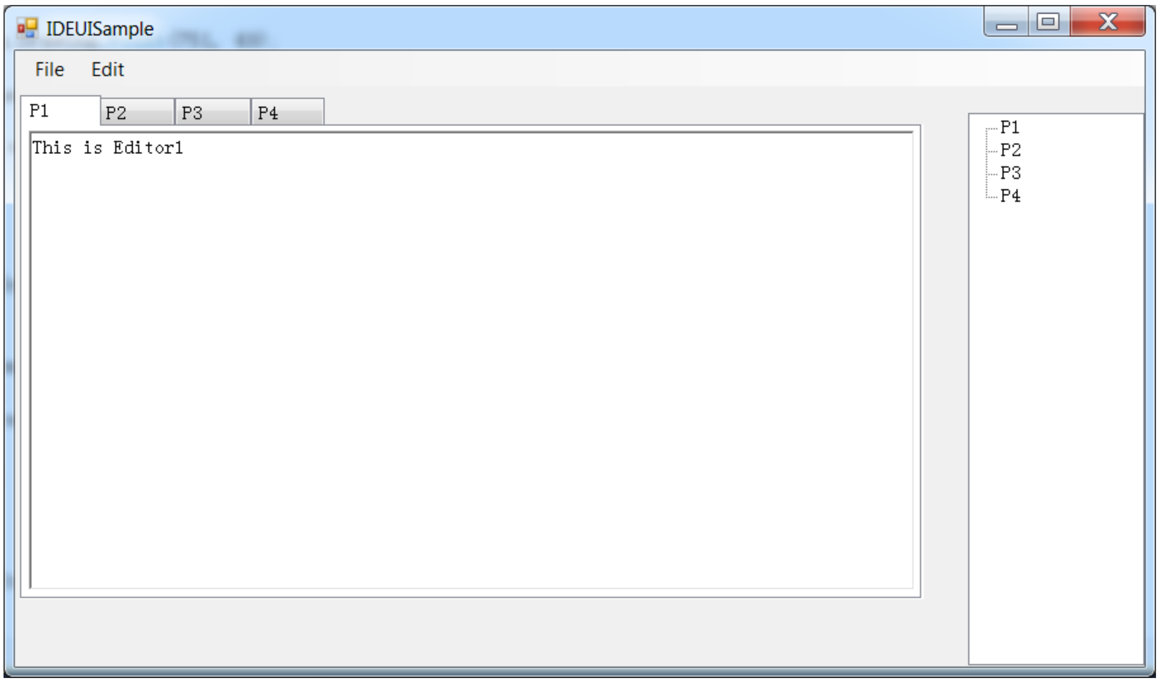


Figure 17 Windows Forms Program run with .NET on Windows

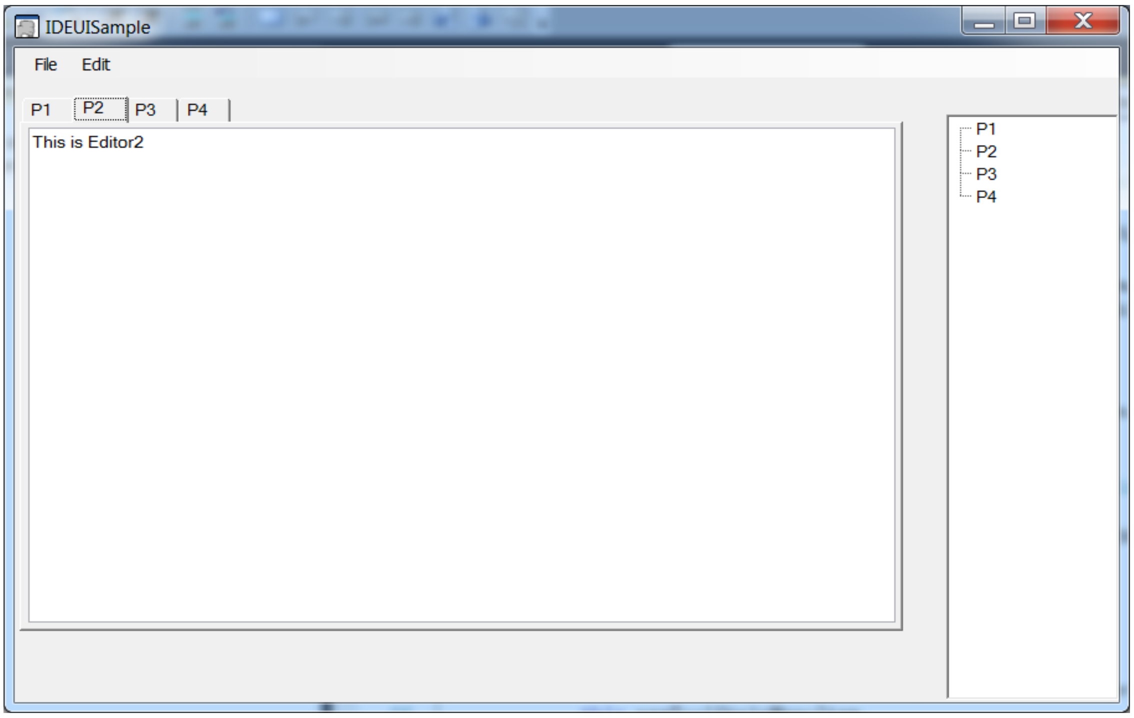


Figure 18 Windows Forms Program run with Mono on Windows

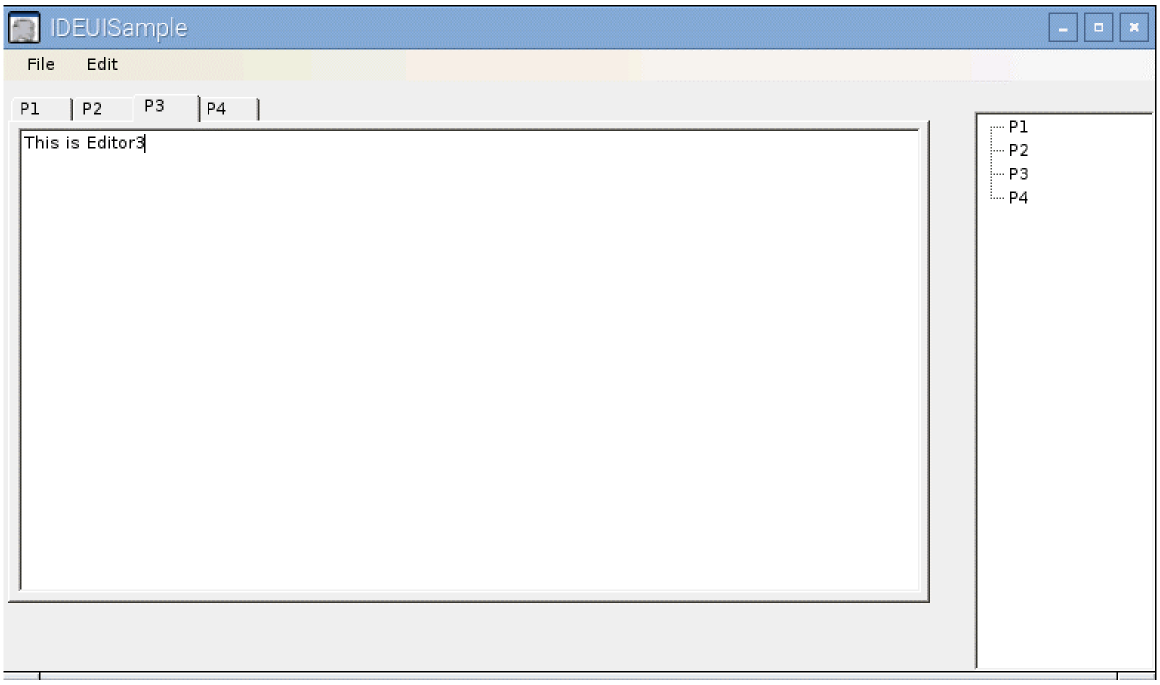


Figure 19 Windows Forms Program run with Mono on Raspbian

Obviously, the layouts of each interface are very similar but they have slightly different appearances since the Mono stack uses native drivers for each of the different operating systems supported, including X11, Win32 and OS X[28]. Aside from this, all features are achieved on both Windows and Raspbian OS.

Ideally, a .NET Framework application can be ported to a Mono platform without modification. However, as Mono states in the documentation about WinForms, “it is very unlikely that the implementation will ever implement everything needed for full compatibility with Windows.Forms. The reason is that Windows.Forms is not a complete toolkit, and to work around this problem some of the underlying Win32 foundation is exposed to the programmer in the form of exposing the Windows message handler(WndProc). Any control can override this method. Also developers often P/Invoke into Win32 to get to functionality that was not wrapped.”[28] In other words, the .NET Windows Forms is a binding over the Windows Win32 API, and most practical applications use methods that directly invoke Win32 callbacks. This links these applications to the Windows platform and makes it infeasible for them to operate on other platforms.

As shown in the architecture of QuickSharp IDE in Figure 15, QuickSharp and its WeiFenLuo.WinFormsUI library are built on top of Microsoft’s .NET Windows Forms. However, if the PiSharp uses another GUI toolkit provided by Mono, like GTK#, it likely to be necessary to rewrite the entire IDE structure and have to abandon the WinFormsUI assembly, particularly for the window docking control. Therefore, PiSharp chose Mono’s WinForms GUI toolkit for the main part of IDE so as to require with less changes to the structure.

3.4 GTK+

As a range of platforms text editing supplier, Scintilla supports editing functions on GTK platform. GTK+ is one of the most popular GUI widget toolkits for the X Windowing System(also called as X11 or X). It was initially developed for the X Windowing System widely used on UNIX-like operating systems, and is now implemented on multiple platforms including Windows and OS X. X11 does not forms a user interfaces, but offers basic protocols and graphical primitives for desktop environments to design and display graphical user interfaces for applications in their own style, and allows users to interact with the applications using input devices. It means every desktop environment can have a similar window appearance.

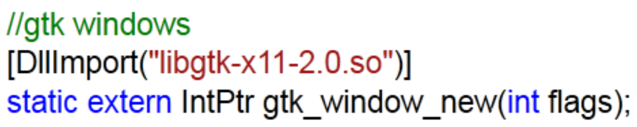
For both the PiSharp and Pi-XNA project, the underlying operating system – Raspbian uses LXDE(Lightweight X11 Desktop Environment) or LXDE-based PIXEL(Pi Improved X-Windows Environment, Lightweight) as its major desktop environment. LXDE is a desktop environment implementation based on the GTK+ 2 toolkit primarily aimed at the X Windowing System.

During PiSharp IDE development, the biggest barrier was the ScintillaNET-based text editor which cannot be displayed in PiSharp's docking window system. Instead, the decision was made in the PiSharp project to produce the text editor for each opened document within an individual GTK+ top level window using a modified version of ScintillaNET API.

To create a GTK+ application on the Raspberry Pi, the first step is to set up a GTK+ development environment with the command:

```
apt-get install libgtk2.0-dev
OR apt-get install libgtk-3-dev
```

In a managed program development, the System.Runtime.InteropServices assembly allows developers to create "DllImport" function declarations for accessing an unmanaged library with a valid and explicit path, and invoke code inside the library. In other words, it is possible for PiSharp to produce a GTK-based text editor by directly importing every required procedure call from a GTK+ library. Figure 20 shows the declaration of GTK Window creation function as the main container for other widgets. The reason “DllImport” entries identifies a GTK+ library by a name rather than an explicit path is that the directory where the GTK+ shared library is located, “/usr/lib/arm-linux-gnueabihf/”, is one of the paths that the dynamic linker searches by default.



```
//gtk windows
[DllImport("libgtk-x11-2.0.so")]
static extern IntPtr gtk_window_new(int flags);
```

Figure 20 Import GTK Window creation function into C# code

The GTK+ environment should be initialized before use to establish connection to the operating system windowing facilities. Once it is initialized, GTK+ operates with an event driven model. Usually, GTK+ waits for input in its “main loop”. When a user action, like key press, occurs the main loop will activate and call the appropriate callback routine. Although, this is the same mechanism as the Mono WinForms uses, the GTK+ and Mono system will each have their own main loop. This causes difficulties that will be discussed later.

ScintillaGTK is a custom GTK Widget, and therefore need to run with GTK+. In particular, GTK+ must be initialized and a GTK+ main loop must be running. Furthermore, the approach of associating with GTK+ API to produce ScintillaNET-based text editor for PiSharp IDE is describing in detail in Chapter 5.

The complete architecture of PiSharp IDE is presented in Figure 21. To sum up, Mono is an ideal tool for porting .NET applications to Linux distributions. In this case, Raspbian.

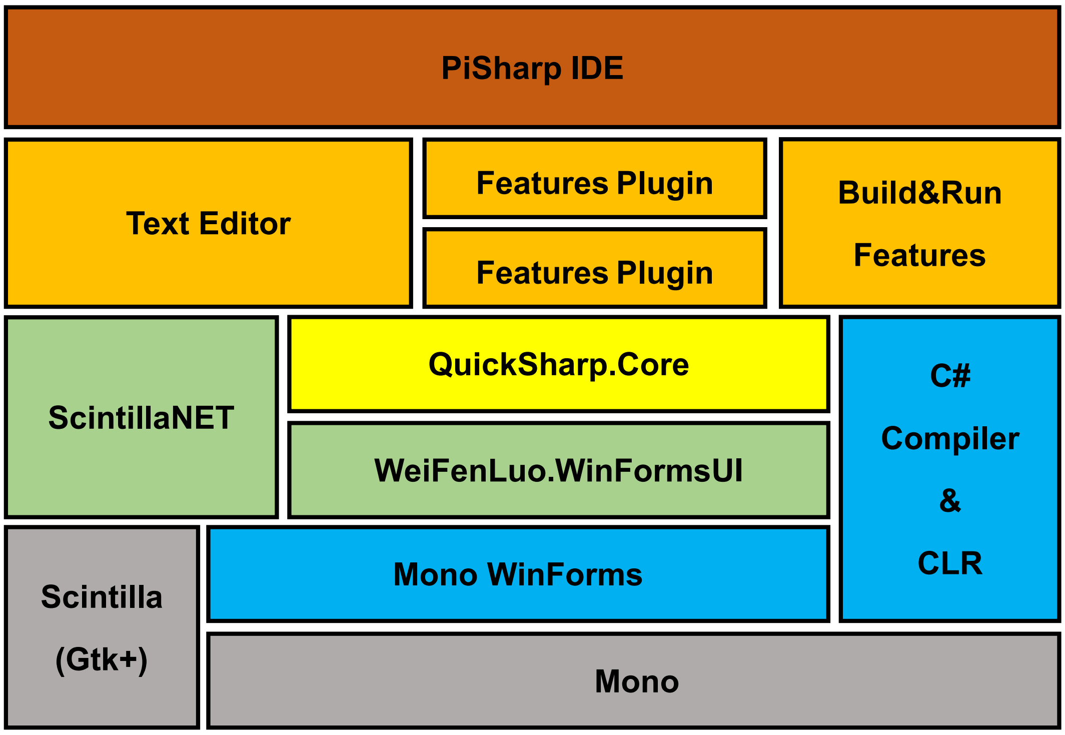


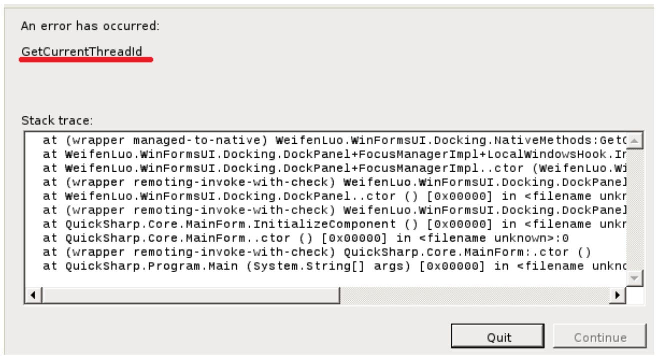
Figure 21 Architecture of PiSharp IDE on Linux

Chapter 4. Issues porting QuickSharp to Raspbian

This chapter describes the general issues occurred during porting QuickSharp to Raspbian platform and exposes corresponding solutions for each of them. The approaches of implementations and improvement in the areas of IDE user interface, text editor, programs building process and IntelliSense. Clarify the issues and figure out the feasible solution.

As described in Chapter3 – "Underlying System", one of the most significant Mono components is the Mono runtime. The original QuickSharp project is built with the .NET Framework on the Windows operating system using Microsoft implementation of Winforms.

In simple cases, Mono could have been compatible with the executed program and run with Mono. However, the QuickSharp program or its underlying components such as WinFormsUI and ScintillaNET require some native procedure calls. Figure 22 shows the error message of an application invoking the "GetCurrentThreadId" Win32 call, when directly running QuickSharp on Raspbian. Therefore, the portation should have commented out those unexpected P/Invoke calls or find alternatives.



An error has occurred:

GetCurrentThreadId

Stack trace:

```
  at (wrapper managed-to-native) WeifenLuo.WinFormsUI.Docking.NativeMethods:GetC
  at WeifenLuo.WinFormsUI.Docking.DockPanel+FocusManagerImpl+LocalWindowsHook.Ir
  at WeifenLuo.WinFormsUI.Docking.DockPanel+FocusManagerImpl..ctor (WeifenLuo.Wi
  at (wrapper remoting-invoke-with-check) WeifenLuo.WinFormsUI.Docking.DockPanel
  at WeifenLuo.WinFormsUI.Docking.DockPanel..ctor () [0x00000] in <filename unkr
  at (wrapper remoting-invoke-with-check) WeifenLuo.WinFormsUI.Docking.DockPanel
  at QuickSharp.Core.MainForm.InitializeComponent () [0x00000] in <filename unkr
  at QuickSharp.Core.MainForm..ctor () [0x00000] in <filename unknown>:0
  at (wrapper remoting-invoke-with-check) QuickSharp.Core.MainForm:.ctor ()
  at QuickSharp.Program.Main (System.String[] args) [0x00000] in <filename unknc
```

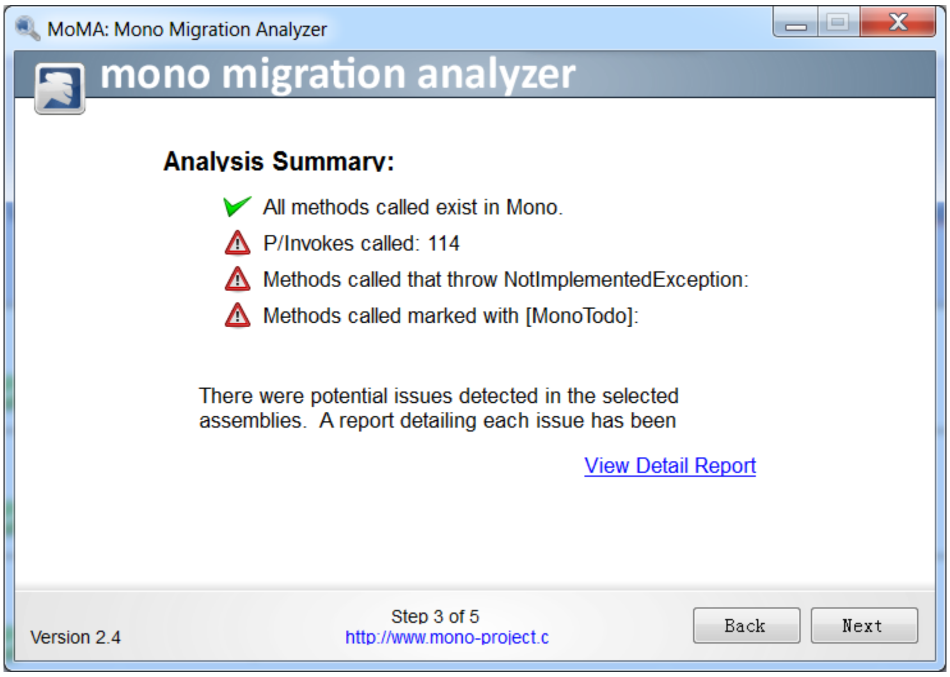
Figure 22 error message of P/Invoke

In this chapter, the discussion focus on porting the original open source QuickSharp IDE on Windows to a LINUX distribution Raspbian. The basic method of porting an existing Winforms Application is taken from an article which is published in the documentation of the official Mono project website by Pobst[31] as a guide of porting methodology.

The process starts with a tool called Mono Migration Analyzer(MoMA) which can help developers identify potential issues that may have when porting a .NET application to Mono. It aids the developer in pinpointing platform specific calls(P/Invoke) and areas in their code that are not yet supported by the Mono project and therefore will not work in Mono on Linux distributions. Although many complex factors could not be covered by the simple tool, the results still can be regarded as a guide to get started on porting an application to Mono.

4.1 MoMA analyzer report

This project starts with analyzing the potential issues of porting the C# QuickSharp application. Mono allows to develop and run .NET applications cross-platform on Windows, Linux, Mac OS X, and Unix. MoMA demonstrates an overview of issues while porting a .NET application to Mono platform, which helps port a .NET Windows application to Linux.



Analysis Summary:

✓ All methods called exist in Mono.
⚠ P/Invokes called: 114
⚠ Methods called that throw NotImplementedException:
⚠ Methods called marked with [MonoTodo]:

There were potential issues detected in the selected assemblies. A report detailing each issue has been

View Detail Report

Figure 23 using MoMA for QuickSharp

The MoMA detail report of the QuickSharp SDK package has been illustrated in Appendix B. MoMA reports four types of potential issues. The detected issues descriptions from MoMA documentation are included in this section.

Missing Methods

"This is the most severe type of issue. These methods represents the methods that have not been implemented yet in Mono. If you try to compile your application that uses these methods with Mono, you will get an error like:

myfile.cs(22,16): error CS0117: 'xxxxxxxxxxxxxxxxx' does not contain a definition for 'xxxxxxxxxxxxxxx'

If you compile your application with Microsoft's compiler, your application will run on Mono until it tries to use the missing method. It will then exit the whole application with an error like:

System.MissingMethodException: Method not found:xxxxxxxxxxxxxxxxxxx

These method calls must be worked around and removed from the application before you can compile or run on Mono. Alternatively, you can implement the function yourself in Mono and submit it for inclusion in future version of Mono."[37] According to the report from MoMA, the QuickSharp source code does not contain [Missing Methods] problem.

MonoTodo

"Methods marked with [MonoTodo] may or may not cause problems for your application. Sometimes a method may be marked with this to remind a developer that some small part is not implemented or to clean it up later. Other times, the method may not be implemented at all and simply will not perform any function. This is generally done to make an application compile and run, even if it missing some functionality. The detail report may list a specific reason why the method is marked with [MonoTodo]. Going forward, it has been requested that any developer who uses [MonoTodo] provide a reason that can be used for this report. However, numerous pre-existing tags do not have this reason.

These issues can probably be ignored in your initial porting. The application should still run without crashing, however there may be missing functionality. Missing functionality can be fixed by working around Mono’s unfinished method, implementing the method yourself, or waiting until the method is completed in Mono.”[37]

NotImplementedException

“In many cases the methods are not implemented at all, and simply throw a NotImplementedException as soon as they are called. In other cases, the method may only throw the exception under certain circumstances, while most calls work as expected.

These issues are similar to MonoTodo’s. It is a gamble as to whether they will cause problems or not. The application will compile just fine under Mono with these issues, and you will need to test the application to see if further work are required around these calls.”[37]

P/Invokes

“P/Invokes(Platform Invokes) are used to call functions that are written in unmanaged languages, often times provided by the current platform itself(user32.dll, shell32.dll, kernal32.dll on Windows). However, these can also be calls into your own unmanaged libraries. Mono can handle these calls when the unmanaged library is available for the platform you are using, however many times the whole purpose of using Mono is to run on many platforms.

--All methods called exist in Mono, which means you’re not calling methods that the Mono project hasn't implemented.

--No P/Invokes are called, which means you’re not calling directly into the operating system.

--No methods that throw NotImplementedException are called, which means you're not calling methods that technically exist as a stub in Mono but haven't yet been coded.(Remember, Mono is an ongoing project.)

--No methods marked with [MonoTodo] are called, which is similar in nature to the previous category.”[37]

4.2 Modification for running PiSharp on Linux

Case Sensitivity:

Pobst pointed out “difference between Windows and many other operating systems such as Linux is that the file system is case sensitive. That is, in Windows the files ‘readme.txt’ and ‘README.TXT’ are the same, but in Linux those are distinct files.”[31] “Although this will work on Windows, it will generate a FileNotFoundException on Linux.”[31]

The Path Separator:

“Another issue may run across is the path separator(“\”) used in file paths. In many other operating systems, such as Linux, the path separator is a forward slash(“/”) instead of a backwards slash like Windows. In our example, we have hard coded a backwards slash that will cause our file to not be found.”[31] One thing that could be found is that when programmers are developing code, they might replace a backslash “\” with double backwards slash(“\\”) or(@“\”) in file paths to prevent the regular expression of compilers a backslash escapes the following character to a special character, which can make a specific path invalid.

Windows ADO.NET:

On a Windows platform System.Data.SQLite is an ADO.NET(database interface) provider for SQLite Data which is an embedded and serverless SQL database engine. The original QuickSharp IDE provides a feature work with an ADO.NET compatible database making it possible to develop queries in QuickSharp and get intelligent code completion(called “Code Assist” in the QuickSharp IDE) support for SQLite. To keep this feature in Raspbian, a Mono.Data.Sqlite library is required to substitute the Windows’s System.Data.SQLite. This was built by the mono-project team in an effort to allow SQLite features over multiple platforms.

ICSharpCode.SharpZLib:

"ICSharpCode.SharpZLib is a Zip, GZip, Tar and BZip2 library on Windows platform written entirely in C# for the .NET platform."[36] In the QuickSharp IDE, it is mainly used to compress multiple files or a folder under a file management scene, such as put into an archive in WorkSpace or Explorer window and create a new file or project from templates. The Mono project supports ICSharpCode.SharpZipLib to replace the Windows version ICSharpCode.SharpZLib.

P/Invoke calls Issues:

Because the original QuickSharp IDE is for Microsoft's .NET platform running on a Windows System, there are numerous basic features rely on Windows libraries support. For example, drag and drop file operation, send and post messages, show scrollbar, and so forth. These features will be lost while running on Linux distribution because the target libraries do not exist. At this stage, all Win32 assemblies(user32.dll, shell32.dll, kernel32.dll) specific Windows system but not implemented in Linux and all referred function invokes will be temporarily disabled in order to compile the QuickSharp development environment on Raspbian without harm.

WeifenLuo.WinFormsUI user32.dll, kernel32.dll, ScintillaNET user32.dll, kernel32.dll shell32.dll and SciLexer.dll(unmanaged library requires Win32 assemblies). The ScintillaNET replacement will be explained with the text editor. QuickSharp is based on WinFormsUI for Visual Studio like theme and docking windows. Like drag and drops, however, some of WinFormsUI features use Win32 native methods which do not exist in Linux distributions. For example, SetFocus() and GetFocus() are both user32.dll functions relying on Win32 that are not implemented in Linux system itself or Mono libraries. They set or retrieve the keyboard focus to a specified window respectively.

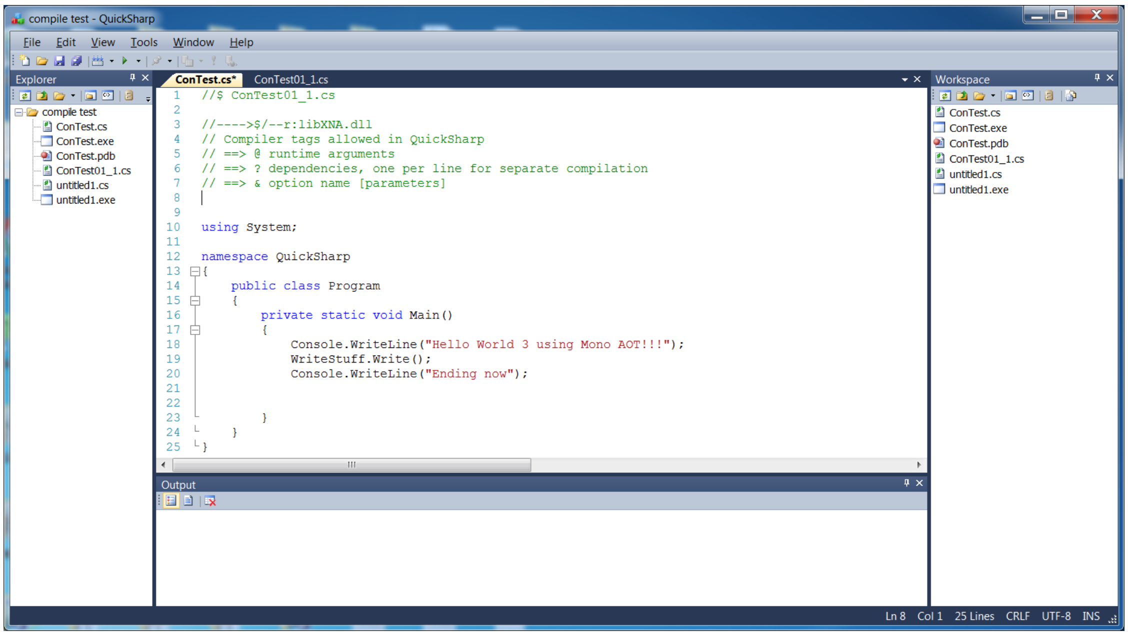
After abandoning the Win32 native methods and fixing some other errors, the IDE displays properly.

Chapter 5. Text Editor

The text editor is one of the most significant parts of an IDE. It is where a developer can write and view source code, supported with syntax highlighting, indicators, code folding, snippet management and other editing features. In this chapter, the Section 5.1 and Section 5.2 respectively introduce how the QuickSharp IDE utilizes a .NET implementation of Scintilla(ScintillaNET) and a docking window component(WeifenLuo.WinFormsUI) to display multi-document text editors; and the ScintillaNET-based text editor's dependencies and usage on Raspbian/Linux. The third section then explains issues of the text editor in PiSharp after porting from the Windows platform to Raspbian, and describes the re-formed new text editor within the PiSharp environment in detail.

5.1 Text Editor in QuickSharp on Windows

Figure 24 shows the basic layout of the QuickSharp IDE on the Windows platform. Two documents: "ConTest.cs" and "ConTest01_1.cs" are sharing the same area, located and displayed in the middle of the QuickSharp IDE's main window with tabs. In the QuickSharp environment on Windows, the multiple document text-editing window is supplied by two major components: ScintillaNET and DockPanelSuite's WeifenLuo.WinFormsUI.

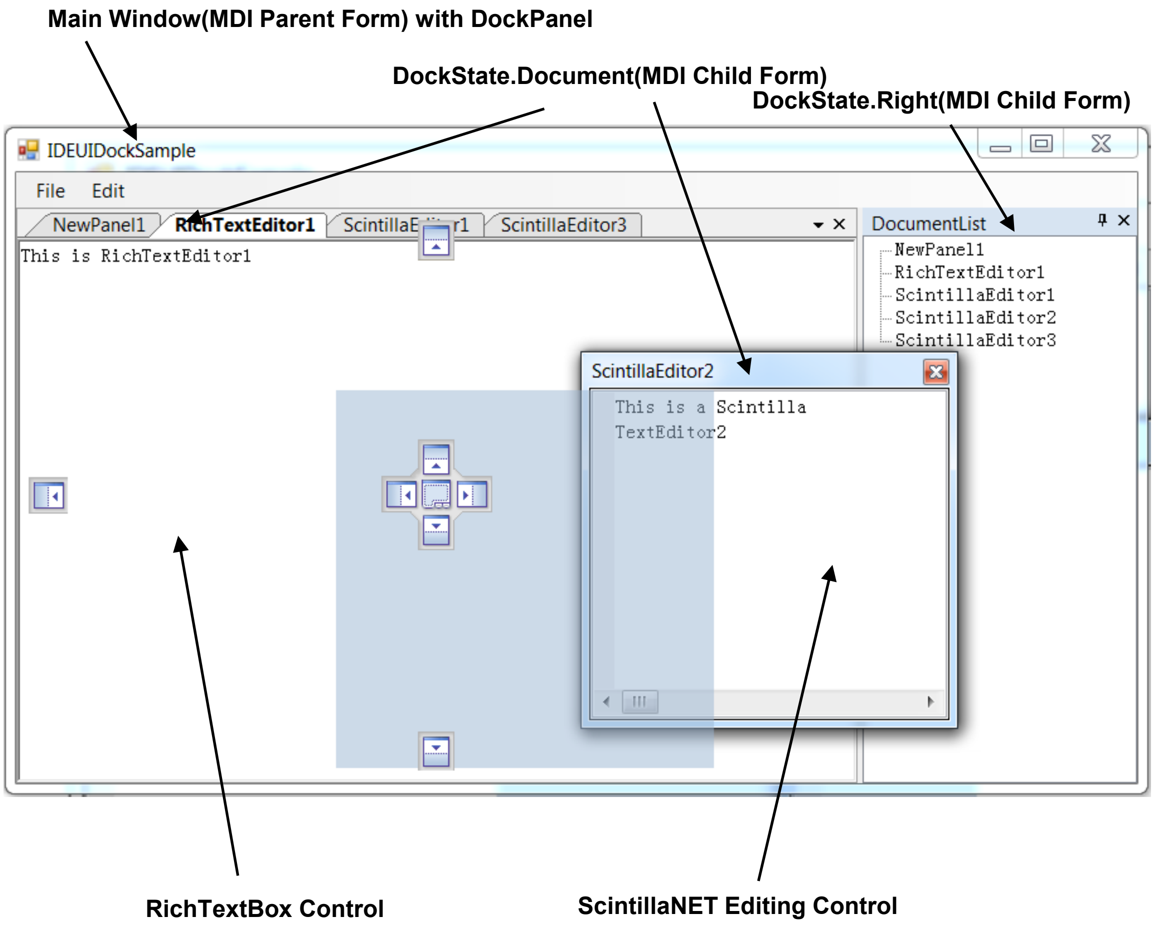


```
//$ ConTest01_1.cs

//---->$/--r:libXNA.dll
// Compiler tags allowed in QuickSharp
// ==> @ runtime arguments
// ==> ? dependencies, one per line for separate compilation
// ==> & option name [parameters]


using System;

namespace QuickSharp
{
    public class Program
    {
        private static void Main()
        {
            Console.WriteLine("Hello World 3 using Mono AOT!!!");
            WriteStuff.Write();
            Console.WriteLine("Ending now");


        }
    }
}
```

Figure 24 QuickSharp layout with Scintilla-based Text Editor

As mentioned in the Underlying System chapter, Scintilla is a text editing component, written in C++, intended to be a part of a program and is compatible with most common platforms such as Windows, OS X with COCOA, QT and LINUX distributions with GTK+. The API provides advanced features for editing and debugging source code in multiple programming languages including syntax highlighting, searching, replacing, error indicators, and so on. As a derivative of Scintilla, ScintillaNET is a wrapper for Win32-based Scintilla component written in the C# language, which can be used as general controls in .NET Windows Forms applications. ScintillaNET allows the use of text editing features from Scintilla and provides some additional features, like multiple key-command bindings. The QuickSharp IDE takes advantage of ScintillaNET for most program editing behaviors. The WeifenLuo.WinFormsUI library provides docking window layouts. It is an open-sourced alternative to Visual Studio themes, built on top of the .NET Framework Windows Forms.

To expose the QuickSharp text editor structure and further explore porting issues, this project starts with building a sample application called "IDEUIDockSample". The layout of this application is demonstrated in Figure 25. Like QuickSharp, "IDEUIDockSample" is a main parent window, which is assigned as a Multiple-Document Interface(abbr. MDI) container for WinFormsUI dockable windows. A DockPanel(a WinFormsUI version of the Windows Forms panel control) is created to fill the client area of the MDI Parent-Form except for the Menu bar. All MDI Child-Forms will be docked to specified position on the DockPanel. This application allows users to create document DockState Child-Forms with a total of three different type of controls: a simple blank Panel colored red for clarity, a RichTextBox control and a ScintillaNET control. Once a document is created by clicking one of "NewPanel", "NewRichText" or "NewScintilla" on the "File" drop-down Menu, a tab will be shown at the top of the document to enable users to switch between and manage the multiple documents. In the meantime, the child-window "DocumentList", which is docked to the left, simply presents the Workspace/Explorer window of QuickSharp with a normal TreeView control listing all current open documents within the sample application. The document DockState Child-Forms are attached to share the rest of window. As shown in Figure 25, each Child-Form may be “torn” off and dragged to a new position including top, bottom, left and right of the DockPanel or even as a floating window above the whole MDI Form by indicators.



This is RichTextEditor1

This is a Scintilla TextEditor2

Figure 25 IDEUIDockSample layout

In the QuickSharp IDE, once a document is created or loaded, the IDE sets up a Child-Form on the IDE’s main user interface containing a Scintilla-based text editor as a control object. The WinFormsUI-based EditForm class provides the IDE’s user interface level interaction. The Child-Form inherits from a QuickSharp self-contained “ScintillaEditForm” abstract class which provides QuickSharp-specific functionality for the Scintilla-based text editor so that QuickSharp can react appropriately while a user is editing documents. For example, QuickSharp will update the status bar at the bottom of the main window to display the current cursor position when a user clicks in the document shown in a text editor.

According to the Scintilla documentation, when using the Scintilla API on the Windows platform, the first step is to load an unmanaged Scintilla library: “SciLexer.dll”. The SciLexer.dll is a Win32 version of a Scintilla library that contains a range of programming language LEXERs(lexical analyzers) and lexing support features. The ScintillaNET API loads this library for each Scintilla-based text editor creation. Once the library is loaded successfully, it will register a new window class “Scintilla” as a new Scintilla text editing control. This Scintilla control object can be handled like other Windows controls[26]. The QuickSharp section in the Underlying System chapter introduces the architecture of the entire QuickSharp IDE in Figure 15. Figure 26 shows the architecture of the Text Editor part extracted from the QuickSharp IDE architecture diagram. To be compatible with .NET WinForms applications, ScintillaNET maintains this registered “Scintilla” class with its own .NET wrapper to communicate with the Scintilla library at run-time, to bring in Scintilla native features and to improve the capability of the text editor control. Figure 26 shows the layout of a QuickSharp text editor running on Windows when a C# document is opened. Both Microsoft .NET Windows Forms and Win32-based Scintilla use the underlying windowing features of the Microsoft platform, and the Windows can cooperate because C# code can access a window by its Win32 handle, even if it was created by C++ code.

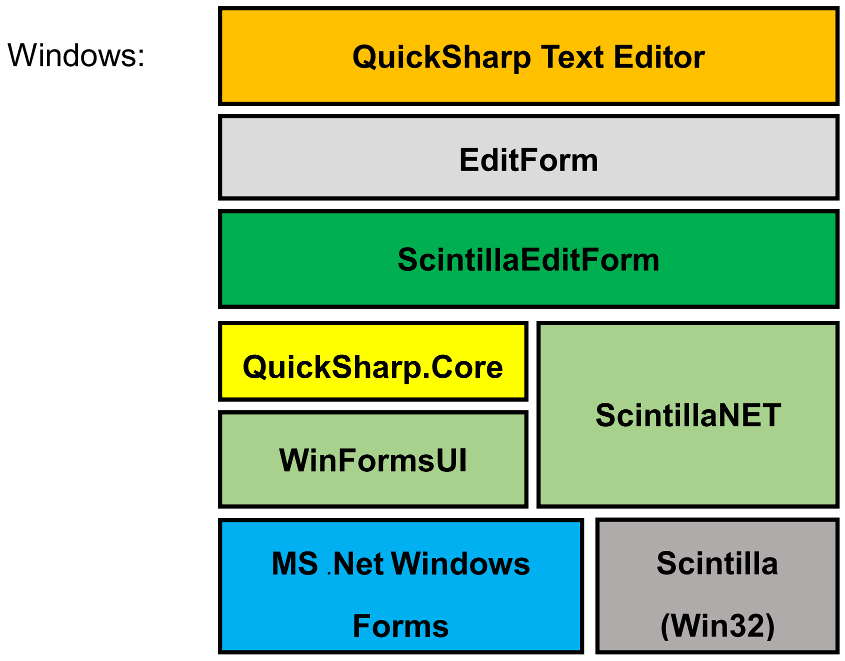


Figure 26 Architecture of a QuickSharp Text Editor on Windows

5.2 Create and Use Scintilla Shared Library on Raspbian

Both the .NET Framework and Mono implementations of the CLI specification support interoperability features allowing managed code to invoke unmanaged methods that are implemented in dynamic-link libraries. As introduced in the underlying system chapter and the Section 5.1, the QuickSharp project and its fundamental dependencies are: DockPanelSuite API(WeifenLuo.WinFormsUI.Docking.dll) and ScintillaNET API are written in the C# language. The core code of QuickSharp is completely built on top of the .NET Framework while the DockPanelSuite API and ScintillaNET API also require use the interoperability feature to directly access native Windows methods. Although Mono has the mechanism to access native method, some required Windows native methods are not available on Linux system or not implemented in Mono Framework.

To port the QuickSharp application to run on Mono, all native usage of Microsoft Win32 API from DockPanelSuite and ScintillaNET has been sacrificed or substituted by skipping or replacing the Windows P/Invoke call. Among the P/Invoke calls and native dependencies, the unmanaged SciLexer library which supplies core Scintilla functionality for ScintillaNET API relies on Windows Win32 native methods. In contrast to QuickSharp's architecture, PiSharp is a port of the Windows version of QuickSharp using the WinForms system in Mono. There is no doubt that the Win32-based Scintilla library(SciLexer.dll) cannot be directly used by the PiSharp project on Linux distributions, in this case, Raspbian OS. However, Scintilla source code supports a number of versions for multiple platforms including Windows Win32, COCOA, QT and GTK. Unlike the dynamic-linking SciLexer library offered for the Windows Win32 platform, the official Scintilla source code only supports production of a static library(*.a) for the GTK platform. In general, unmanaged functions implemented in a static library are linked at compile-time. For each function call, a linker takes a copy of the machine code of the function from the static library and copies it into the final binary file. In contrast, a function in a dynamic or shared library, referenced by a program in source code, is allocated addresses in memory space and loaded at load-time or later at run-time.

In order to use native Scintilla functions and variables in ScintillaNET, managed ScintillaNET must embed the unmanaged Scintilla component as a shared library. This has an added advantage in that using a shared Scintilla library will improve the maintainability and extensibility of PiSharp, because any new features developed for the text editor can be supplied in an update version of the Scintilla library without recompiling the entire program. Therefore, the PiSharp project improves the Scintilla library to create a shared library instead of a static library so that the functions implemented in the unmanaged Scintilla library can be easily bound into managed C# code using the interoperability capability of the .NET/Mono Framework. In the PiSharp project, the Scintilla core API is no longer the Win32-based SciLexer library but the GTK-based Scintilla library.

Basically, a Scintilla static library is built from a set of object files(.o) compiled from corresponding source files. The tool used for producing the Scintilla library is called ‘make’, which reads a specification written in a file called ‘Makefile’. The Scintilla Makefile compiles each source file into an individual object file and generates a single archive file(the Scintilla static library) from the object files. To create a Scintilla shared library, the first step is to create object files that will be gathered into the target shared library by using the “-fPIC”/“-fpic” flag(may require adding the “-fPIC” flag manually in early versions of the Scintilla GTK Makefile). The “f” stands for a set of “gcc” command’s options that “control the interface conventions used in code generation”[32]. “PIC” is an abbreviation of “Position Independent Code” and means that the generated machine code can be located at any memory address, which is commonly used by shared object generation. Typically, shared libraries are named by prefixing with “lib” and suffixing with “.so”(some shared libraries may be specified with an additional version number, mirror number or release number). Therefore, the new Scintilla shared library associated with GTK+ GUI toolkit is named with “libscintilla.so” and the Scintilla shared library generation command which replaced the static library can be written as:

```
g++ -shared -o libscintilla.so ScintillaGTK.o PlatGTK.o LexerBase.o LineMarker.o ……
```

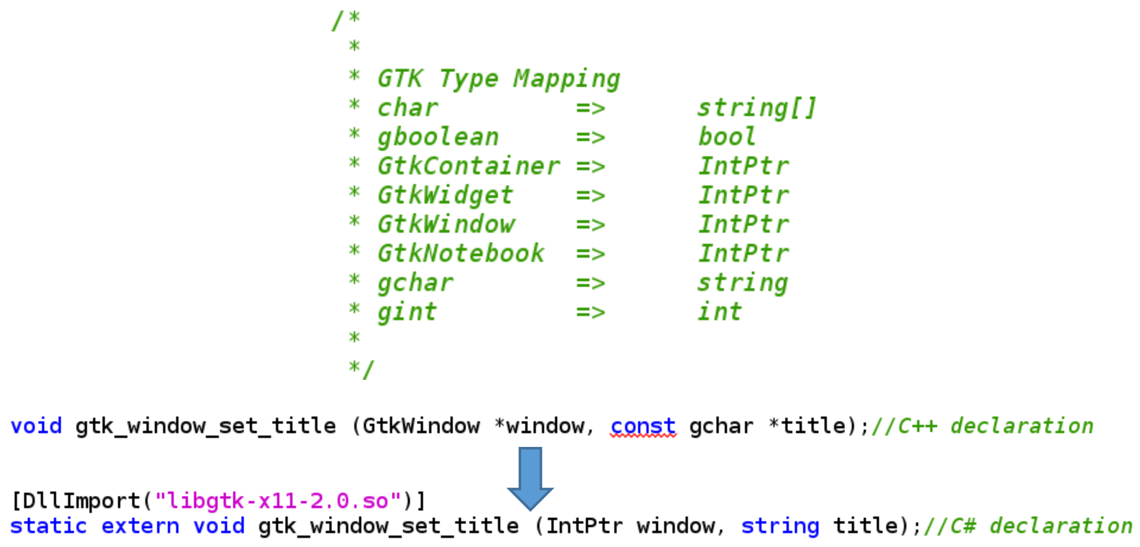
The “-shared” command option stands for producing a shared object that can be linked with other objects to form an executable or other libraries. The “-o” option is following with a specified output library name. In this case, the output shared library is “libscintilla.so”.

On Linux distributions, according to dlopen(3) man page, the shared libraries required by a program are retrieved by searching in the following sequence:

1. A colon-separated list of directories that in the user’s LD_LIBRARY_PATH environment variable. This requires users manually export the explicit library path.
2. The list of libraries cached in /etc/ld.so.cache. /etc/ld.so.cache is created by editing /etc/ld.so.conf and running ldconfig(8).
3. /usr/local/lib(some Linux Distributions may not include this directory in /etc/ld.so.conf file for default directories searched), /lib and /usr/lib in that order[33].

In the PiSharp project, the “libscintilla.so” library has been placed in /usr/lib directory so that PiSharp users can run this IDE application without further required Scintilla library exporting commands.

To consume functions in an unmanaged library from managed code, CLI Framework Platform Invoke(P/Invoke) features must be used. Specifically, the “System.Runtime.InteropServices” namespace enables a CLI program to use the “DllImport” function declaration to specify explicit functions and the dynamic library(*.dll) or shared library(*.so) that contains them so that they can be used just like a static entry point. In C#, the “DllImportAttribute” identifies the library and functions, and defines the functions with the “static” and “extern” keywords. Figure 28 shows a sample program written in C#, which invokes functions implemented in libscintilla.so and libgtk-x11-2.0.so GTK+ 2.0 library, builds a Scintilla editor embedded in a GTK top-level window and defines its output layout. “scintilla_new()” and “scintilla_send_message()” functions are both implemented in the libscintilla.so library. “scintilla_new()” builds on top of the GTK widget to create a Scintilla widget that can be added to a GTK container and displayed. “scintilla_send_message()” is the main entry point that allows setting of parameters and supplying data to the Scintilla API to accomplish the configuration of a Scintilla widget. Additionally, as both libscintilla.so and libgtk-x11-2.0.so are written the in the C++ language, and the GTK+ API defines its own data type, the GTK data types are necessary to convert to C# language defined data types. For instance, “GtkWindow” is used to define a GTK type window container. The GtkWindow and its parent type such as GtkContainer, GtkWidget are defined by a numerical value which represents the registered type ID pointer. In a C# program, “IntPtr” can be used to represent a pointer or a handle accessing unmanaged code. Figure 27 shows the “GtkWindow” type conversion from C++ to C#, and lists all GTK-type mappings used in the PiSharp program and further sample programs. In this sample program, the Scintilla widget is simply configured by adding some text, allocating a LEXER, setting keywords and coloring them. More usage of the Scintilla and GTK shared libraries in PiSharp will be provided in following sections. Figure 29 shows the resulting GTK window.



```
/*
 *
 * GTK Type Mapping
 * char          =>      string[]
 * gboolean      =>      bool
 * GtkContainer  =>      IntPtr
 * GtkWidget     =>      IntPtr
 * GtkWindow     =>      IntPtr
 * GtkNotebook   =>      IntPtr
 * gchar         =>      string
 * gint          =>      int
 *
 */
void gtk_window_set_title (GtkWindow *window, const gchar *title);//C++ declaration

[DllImport("libgtk-x11-2.0.so")]
static extern void gtk_window_set_title (IntPtr window, string title);//C# declaration
```

Figure 27 Gtype mapping

```
[DllImport("libgtk-x11-2.0.so")]
static extern void gtk_init(int argc, string[] argv);

[DllImport("libgtk-x11-2.0.so")]
static extern IntPtr gtk_window_new(int flags);

[DllImport("libgtk-x11-2.0.so")]
static extern void gtk_widget_show_all(IntPtr w);

[DllImport("libgtk-x11-2.0.so")]
static extern void gtk_main();

[DllImport("libgtk-x11-2.0.so")]
static extern void gtk_main_quit();

[DllImport("libgtk-x11-2.0.so")]
static extern void gtk_container_add(IntPtr container, IntPtr contained);
```

Scintilla functions declaration

```
[DllImport("libgtk-x11-2.0.so")]
static extern void gtk_widget_set_usize(IntPtr w, int wide, int high);

[DllImport("libgtk-x11-2.0.so")]
static extern void gtk_signal_connect_full(IntPtr obj, string name, exit_handler func, int zero1, int data, int zero2, int zero3);
// Note:  gtk_signal_connect is a macro, leaving out some parameters
[DllImport("libscintilla.so")]
static extern IntPtr scintilla_new();

[DllImport("libscintilla.so")]
static extern void scintilla_send_message(IntPtr s, int m, int w, int l);

[DllImport("libscintilla.so")]
static extern void scintilla_send_message(IntPtr s, int m, int w, string l);

public static int exit_app(IntPtr w, IntPtr e, IntPtr p) {
    gtk_main_quit();
    return 0;
}

public static void Main(string[] args)
{
```

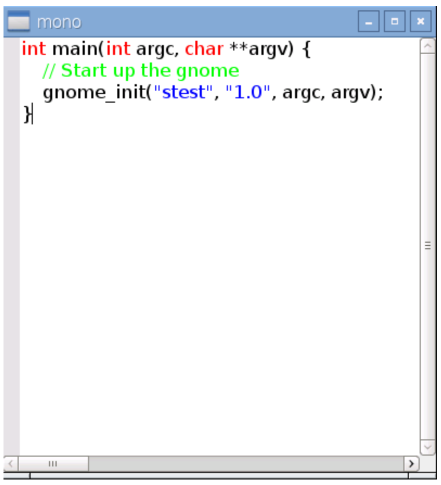
Scintilla widget creation

```
    gtk_init(0, null);

    IntPtr app = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    IntPtr editor = scintilla_new();
```

text, key words, colorization… attributes settings

```
    gtk_container_add(app, editor);
    gtk_signal_connect_full(app, "delete_event", new exit_handler(exit_app), 0, 0, 0, 0);
    gtk_widget_set_usize(editor, 400, 400);

    scintilla_send_message(editor, SCI_SETLEXER, SCLEX_CPP, 0);
    scintilla_send_message(editor, SCI_SETKEYWORDS, 0, "int char");
    scintilla_send_message(editor, SCI_STYLESETFORE, SCE_C_COMMENT, 0xff00ff);
    scintilla_send_message(editor, SCI_STYLESETFORE, SCE_C_COMMENTLINE, 0x00ff00);
    scintilla_send_message(editor, SCI_STYLESETFORE, SCE_C_NUMBER, 0xffff00);
    scintilla_send_message(editor, SCI_STYLESETFORE, SCE_C_WORD, 0x0000ff);
    scintilla_send_message(editor, SCI_STYLESETFORE, SCE_C_STRING, 0xff0000);
    scintilla_send_message(editor, SCI_STYLESETBOLD, SCE_C_OPERATOR, 1);

    scintilla_send_message(editor, SCI_INSERTTEXT, 0,
                     "int main(int argc, char **argv) {\n"
                   + "    // Start up the gnome\n"
                   + "    gnome_init(\"stest\", \"1.0\", argc, argv);\n}");

    gtk_widget_show_all(app);
    gtk_main();
```

Figure 28 a sample C# program calls methods in unmanaged libraries



```
int main(int argc, char **argv) {
   // Start up the gnome
   gnome_init("stest", "1.0", argc, argv);
}
```

Figure 29 the output of the GTK+ Sample program

5.3 Text Editor in PiSharp

5.3.1. Porting issues and possible solutions for the Text Editor on PiSharp

Since the GTK version of the Scintilla shared library can be successfully built on Raspbian and the new Scintilla API has been shown to work as described in Section 5.2, the next step was to use it to implement the text editor in PiSharp. The PiSharp project started by attempting to directly replace the Win32 version of Scintilla library(SciLexer.dll) with the new ScintillaGTK library "libscintilla.so". As explained in Section 5.1, ScintillaNET wraps the original Scintilla API. On the Windows system, ScintillaNET needs to import the SciLexer library, so that a Scintilla class must be registered and a window handle(HWND) of the Scintilla control created before its constructor can proceed. Therefore, a Win32 dynamic library loading function "LoadLibrary()" is used in an overridden Windows Forms Control's properties and appearance initialization: "Control.CreateParams". In the PiSharp project, dlopen() function can be used to import the Scintilla shared library instead of the Windows P/Invoke call.

However, at this development stage, the PiSharp program is terminated when attempting to get the handle of the native Scintilla control when the PiSharp IDE is loading a document into its ScintillaNET text editor. The reason is that all native features of ScintillaNET require a Scintilla handle when making calls to the native Scintilla control, and GTK+ version of the Scintilla library does not auto-register a Scintilla window class when is loaded. Accordingly, a handle of the Scintilla control fails to be created when initializing the control, because the expected Scintilla window class does not exist. Before addressing this problem, the following paragraph will introduce how a window handle is used in a GUI application.

On Windows, a GUI application running and interacting relies on window procedures receiving and processing window messages which are produced by both system and the application. The messages cover a wide range of the communications between system and applications or just between applications, such as input events, window changes, task dispatching, and so on. When the application is started, the operating system creates a GUI thread(which could be the application's main execution thread) for it. The thread attaches a message queue and runs a message loop to handle all events and user requests. Generally, the events convert into corresponding messages, and the messages get posted to the thread's message queue with four parameters: window handle(HWND), message ID(Msg) and two additional system maintained values(wParam and lParam) which are described in Table 3.

Property	Description
HWND	Window handle of the message.
Msg	Message identifier constant of the message.
wParam	Additional information of the message.
lParam	Additional information of the message.
Result	Specifies the value that is returned to Windows in response to handling the message.

Table 3 Message structure[34]

As an application may contain multiple windows, Window handles(HWND) are used to identify and access the appropriate windows that should receive the messages. For each window, the system uses a window procedure callback(WndProc) for message processing. The Message ID is used to determine how a window procedure will respond to a message. For example, A WM_CLEAR message tells the window procedure to delete selected text in the edit control. The message loop is responsible for listening the messages in turn from the thread's message queue and dispatching the messages to appropriate windows. Figure 30 illustrates a simple Windows messaging system. When the program starts running, it sets up the user interface and then the application's GUI thread goes into the message loop and gets messages by continually checking the queue. When the thread finds messages, it takes them and dispatches them to the WndProc of the appropriate windows or sometimes just processes them directly. However, a system presents a GUI application with a large number of events. An application only processes some of the messages. To guarantee all messages will be processed, a default window procedure callback function – "DefWndProc" is used to take over the rest of window messages that are not processed by the application's WndProc and provides default processing.

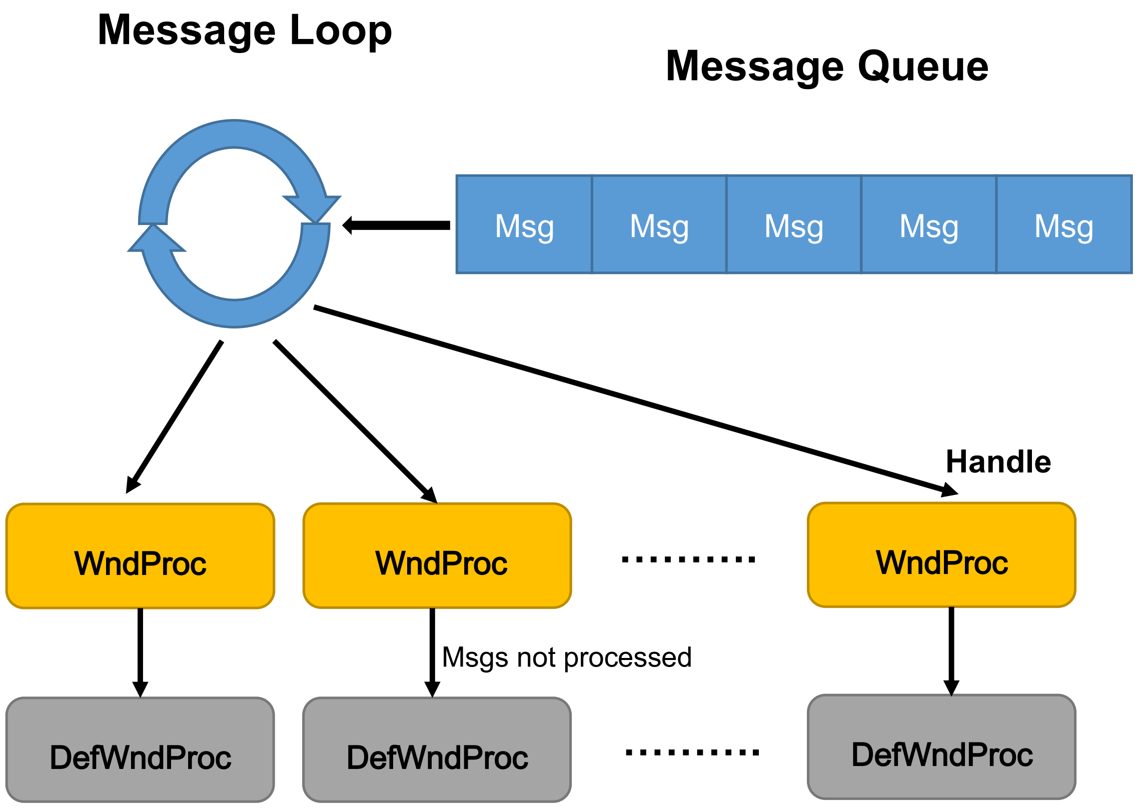


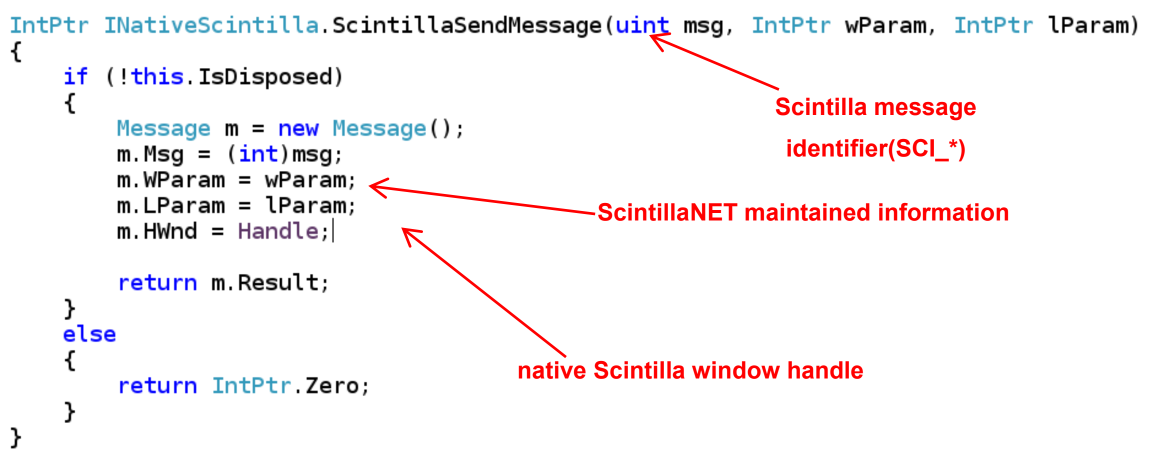
Figure 30 Windows Messaging System

Like other controls, a Scintilla control can communicate with applications by messages. Typically, the Scintilla component defines and processes two major types of message: Scintilla messages(SCI_*) and Scintilla notification messages(SCN_*). Scintilla messages(SCI_*) are a type of Scintilla-specific commands, and they are mainly responsible for completing editing activities within a Scintilla object, such as text modification and retrieval, coloring, zooming, and so forth. The Scintilla notification messages(SCN_*) are for handling events and states, including key pressing, mouse clicking, and UI updating in a Scintilla control. Scintilla messages(SCI_*) are usually used by other programs to tell the Scintilla control what to do or converted from Windows message in response to operating system messages, while the Scintilla notification messages(SCN_*) are sent from the Scintilla control to its container to accomplish appropriate behaviors. Some of Windows editing messages(WM_* or EM_*) can be forwarded to a Scintilla control. These messages will be converted into Scintilla messages(SCI_*) so that the control can execute system editing requests with Scintilla-specific methods. For example, if a window procedure of a native Scintilla control receives a "WM_CUT" message, it will translate the message identifier to "SCI_CUT", and then process this Scintilla message, removing currently selected text in the Scintilla control, and copying the content to the clipboard. The complete message mapping is illustrated in Appendix C.

The Scintilla documentation introduces two approaches to operate a Scintilla control[26]. The first one bypasses the thread's message queue and directly sends each Scintilla message via a Win32 "SendMessage()" function to the Scintilla control's window procedure, and returns the processed message result. As for normal message dispatching, SendMessage requires a window handle to determine which window the message will be passed to. In this case, the identifier is the handle of Scintilla control. Because the sending thread of SendMessage() will be blocked and cannot keep performing other commands until WndProc finishes processing the unqueued Scintilla message, this method could slow performance down, especially in an intensive command-calling case. In order to provide better performance, the other option is to maintain a function pointer to retrieve the address of the Scintilla control's message handling function(WndProc), and directly invoke the window procedure callback function, as a normal procedure call, which avoids the Windows message system. This method only uses the SendMessage() function twice for each Scintilla control instance, to get a pointer to the Scintilla object and a function pointer to the control's WndProc, rather than repeated SendMessage() function calls for the Scintilla commands. A note is declared in Scintilla documentation: "From version 1.47 on Windows, Scintilla exports a function called Scintilla_DirectFunction that can be used the same as the function returned by SCI_GETDIRECTFUNCTION. This saves you the call to SCI_GETDIRECTFUNCTION and the need to call Scintilla indirectly via the function pointer."[26]

However, the version of the ScintillaNET component in QuickSharp uses neither method. As the DefWndProc function can provide a default processing for any window message, ScintillaNET executes each native command by directly calling the Scintilla control's default window procedure. Figure 31 shows that ScintillaNET constructs Scintilla messages associating with the native Scintilla handle and ScintillaNET maintained related information of the message(if necessary), then directly passes the messages to its DefWndProc callback function. This function wraps all native Scintilla calls implemented in ScintillaNET for .NET compatibility. A simple case is to get the current LEXER of the control. A wrapped "GetLexer()" method to directly call the function in Figure 31, would be like:

```
return INativeScintilla.SendMessageDirect(SCI_GETLEXER, IntPtr.Zero, IntPtr.Zero);
```



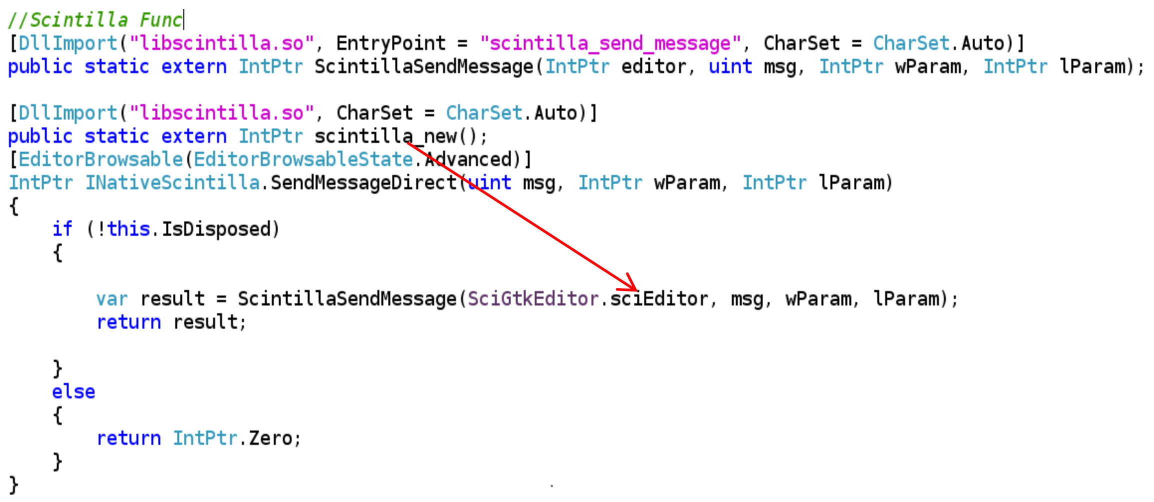
```
IntPtr INativeScintilla.ScintillaSendMessage(uint msg, IntPtr wParam, IntPtr lParam)
{
    if (!this.IsDisposed)
    {
        Message m = new Message();
        m.Msg = (int)msg;
        m.WParam = wParam;
        m.LParam = lParam;
        m.HWnd = Handle;

        return m.Result;
    }
    else
    {
        return IntPtr.Zero;
    }
}
```

Scintilla message identifier(SCI_*)
ScintillaNET maintained information
native Scintilla window handle

Figure 31 ScintillaNET communication method in QuickSharp

All three methods require a native Scintilla handle to determine where the messages will be sent. However, the Scintilla-GTK shared library cannot provide a pre-registered window class for the Scintilla handle creation. If PiSharp keeps using the ScintillaNET communication method or the other that runs without a valid Scintilla window handle, all ScintillaNET functions support for control window creation and appended features will fail to access their corresponding native Scintilla methods once a Scintilla control has been loaded into the PiSharp's main window.

According to the use of libscintilla.so shared library mentioned in Section 5.2, Windows runtime interoperability allows the .NET-based PiSharp program to invoke an unmanaged function from the “libscintilla.so” shared library through “DllImport” command. The “scintilla_new()” and “scintilla_send_message(ScintillaObject *sci, unsigned int iMessage, uptr_t wParam, sptr_t lParam)” functions are used in Section 5.2 to create a GTK-type widget and emulate features of a normal text editor. Using “scintilla_new()” creates a Scintilla object and “scintilla_send_message()” provides an entry point that allows a program to send Scintilla-specific messages to the Scintilla object. As a result, the PiSharp project solved the native Scintilla handle missing issue by creating a GTK-based Scintilla object and using the “scintilla_send_message()” function to directly send each called command to this new Scintilla widget. As the altered native communication method shows in Figure 32, the “SciGtkEditor.sciEditor” is an instance native Scintilla-GTK widget while “ScintillaSendMessage()” is the Scintilla-specific messages sending function. In this context, when a document is opened into the PiSharp IDE, the program no longer crashes by continuous requiring a handle for a Scintilla control. Figure 33 illustrates a state of the PiSharp window opens a “Color.cs” document after using this altered method. A tabbed document Child-Form is set up on the PiSharp’s main window, but no Scintilla object displayed on the child form.



```
//Scintilla Func
[DllImport("libscintilla.so", EntryPoint = "scintilla_send_message", CharSet = CharSet.Auto)]
public static extern IntPtr ScintillaSendMessage(IntPtr editor, uint msg, IntPtr wParam, IntPtr lParam);

[DllImport("libscintilla.so", CharSet = CharSet.Auto)]
public static extern IntPtr scintilla_new();
[EditorBrowsable(EditorBrowsableState.Advanced)]
IntPtr INativeScintilla.SendMessageDirect(uint msg, IntPtr wParam, IntPtr lParam)
{
    if (!this.IsDisposed)
    {

        var result = ScintillaSendMessage(SciGtkEditor.sciEditor, msg, wParam, lParam);
        return result;

    }
    else
    {
        return IntPtr.Zero;
    }
}
```

Figure 32 native communication method in PiSharp

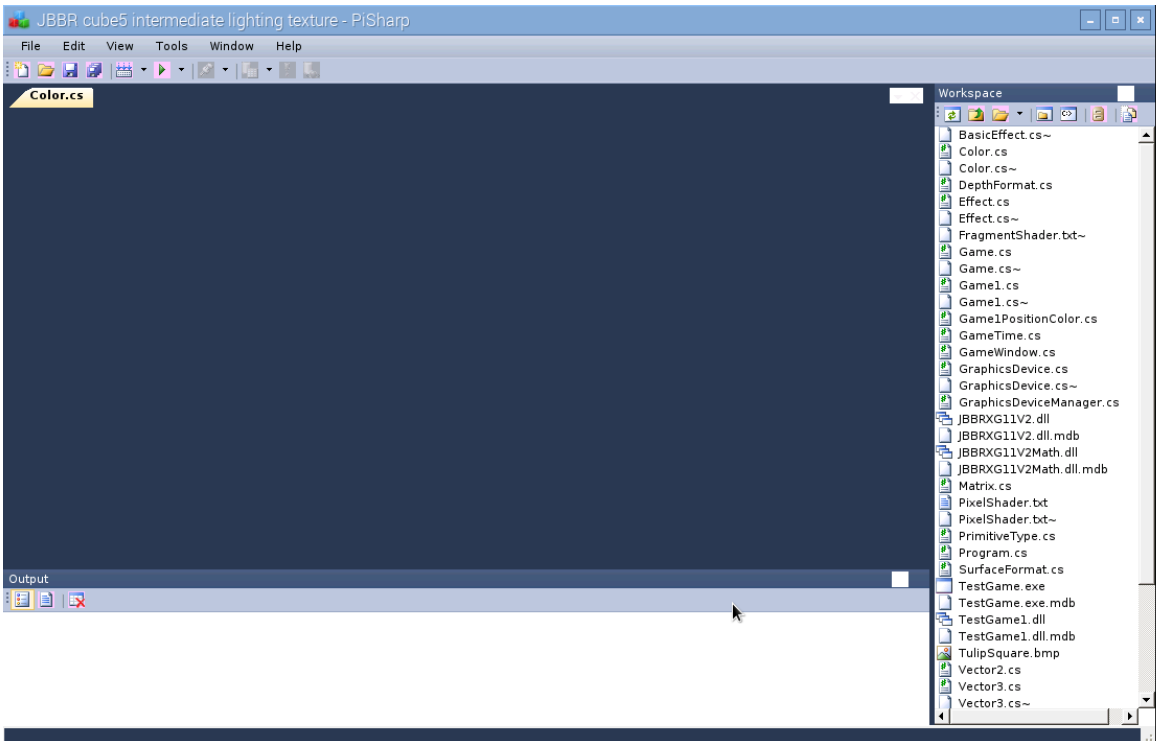


Figure 33 Resulting Window without Scintilla control displayed

Obviously, this method is not enough to show the Scintilla control properly on the PiSharp IDE, because at this stage, PiSharp still adds the ScintillaNET wrapped control to the dockable container despite the fact that Scintilla-specific messages that serve to create and configure the Scintilla control have been sent to the Scintilla-GTK widget. The messages will be processed and the message results are just stored in the widget. Ideally, if all Scintilla-specific messages can be passed to the Scintilla-GTK widget and return the process result back to ScintillaNET. The ScintillaNET control could be possible to be shown. It means all properties and notifications will be assigned for both ScintillaNET and Scintilla-GTK. However, an initialization procedure may just assign data rather than "set" and "get". Moreover, some Scintilla function could just be called by event. Figure 34 demonstrates a simple process of setting text to the Scintilla control when a PiSharp text editor is created. The parent window tries to attach the ScintillaNET wrapped control with default text as "Scintilla Editor". ScintillaNET uses the native Scintilla method of text modification by overriding the Windows Forms control "Text" property. As a result, when the "Scintilla Editor" text property is assigned, the SetText(string text) function gets called, it now wraps the PiSharp-specific native communication method –

"scintilla_send_message()" using the "SCI_SETTEXT" command. Therefore, this "SCI_SETTEXT" message carries the text and directly calls the Scintilla-GTK widget's WndProc. The WndProc processes the message to set the widget text as "Scintilla Editor", even though the widget is not displayed yet. A GetText() function does very similar process. It sends a message with "SCI_GETTEXT" and waits until this message is processed. Neither an effective message callbacks scheme nor a straight forward way to add a GTK-type object to a Windows Forms application as common controls existed at this stage, so it was no wonder that the PiSharp could not present an available text editor.

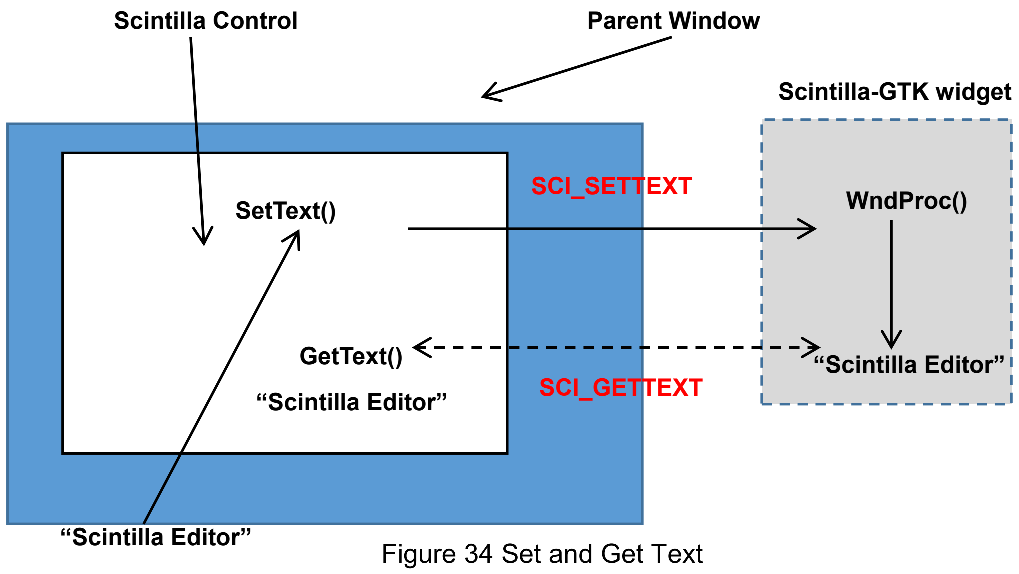
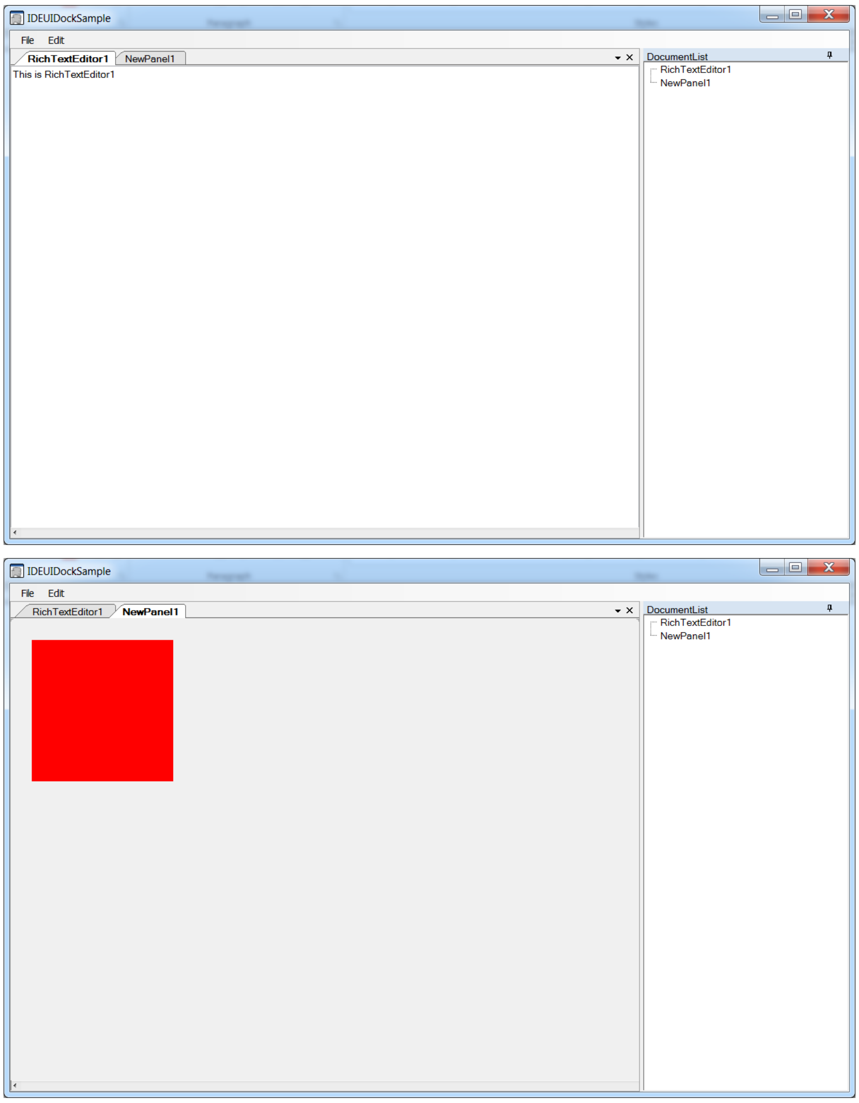


Figure 34 Set and Get Text

During PiSharp project development, this was not the only barrier found that hinders a Scintilla control properly displaying on the docked document MDI Child-Form. Additionally, when a Windows Forms application runs on Linux system with Mono, a control cannot be shown on the MDI child window unless it is a float window or other boundary-state window. To investigate the issue, the WeifenLuo.WinFormsUI-based "IDEUIDockSample" program was ported to run with Mono on both Windows system and Raspbian system. The results are shown in Figure 35 and Figure 36 respectively. As described in Section 5.1, the IDEUIDockSample application allows users to create docked document MDI child windows with three different controls: a blank Panel colored red for clarity, a RichTextBox and a ScintillaNET control. The application run with the Mono-runtime on Windows can do most of the work, except for the Win32 native features invoked in the docking library, for example, drags and drops of the docked windows. In contrast, the application run on Raspbian with Mono cannot show any control on the docked document MDI child windows.



This is RichTextEditor1

Figure 35 IDEUIDockSample on Windows with Mono

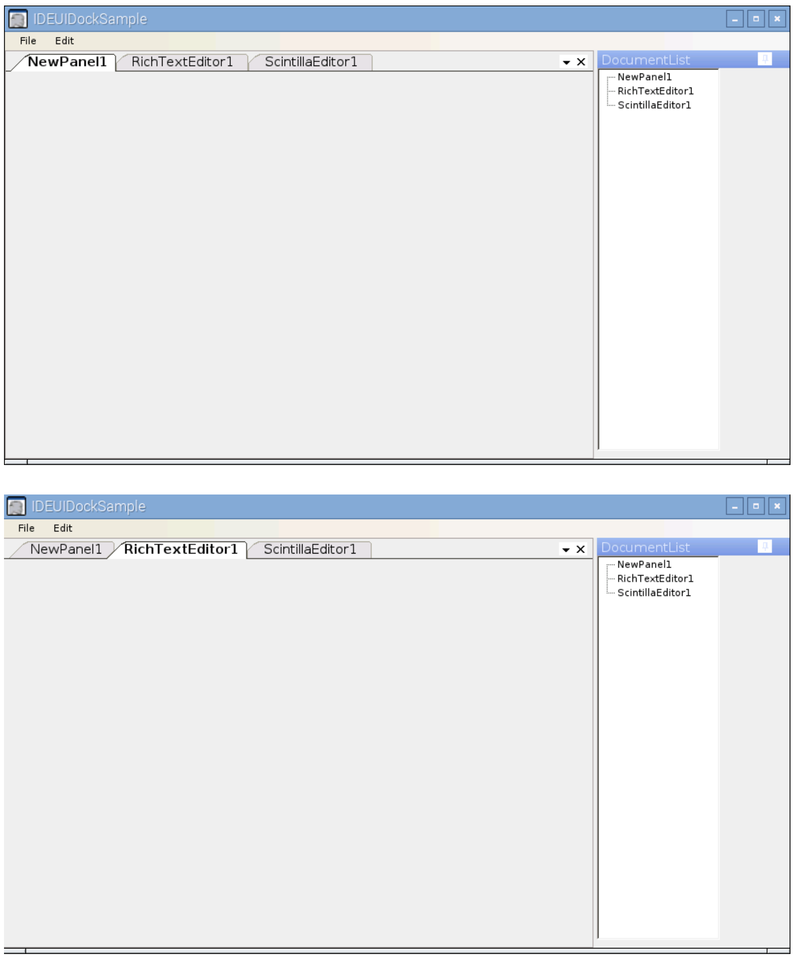


Figure 36 IDEUIDockSample on Raspbian with Mono

Aside from this, an interesting discovery is that if the “dockpanel”, which is the area all MDI child windows dock onto, is set at a fixed size smaller than the client window size, with the MDI Child-Forms dock-style set as “Fill”. Then the control is shown, but at the back of other child windows. The resulting window is displayed in Figure 37.

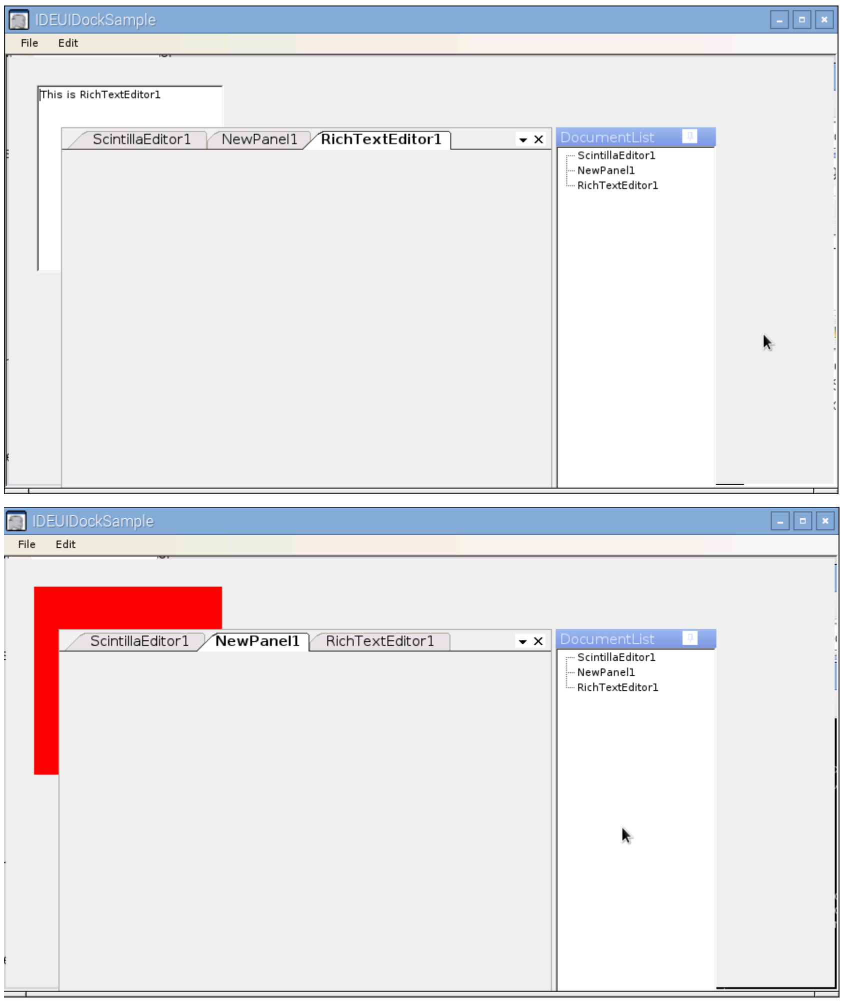


Figure 37 IDEUIDockSample on Raspbian with Mono

To sum up all experiments, because DockPanelSuite's WinFormsUI component primarily aims to provide Visual Studio-like themes for .NET Windows Forms applications, it provides relatively high compatibility for Mono WinForms on the Windows platform. Most times, a Windows Forms application can be supported by the Mono runtime. Mono WinForms applications could perform and run very close to Windows Forms applications. However, some of the .NET Windows Forms functions directly interop Win32 native methods which are not available on Linux. Therefore, the Mono WinForms implementation have to implement this functionality by calling X11 functions or via other existing technologies. This may cause conflicts with other code, because the Mono WinForms may present GUI application features in different ways, even though the eventual outcomes look the same. The example in this case was that the WinFormsUI component presented a document MDI child window with common control commands in the wrong sequence.

As explained, to present a proper Scintilla text editor both Scintilla messages communication in both directions and WinFormsUI compatibility for Mono WinForms on Linux need to be solved at the same time. Otherwise, although an effective connection is built between ScintillaNET and Scintilla-GTK widget allowing the Scintilla-GTK widget send all messages result back to ScintillaNET, the Scintilla control still cannot be displayed in the docked document window, and vice versa.

There were several possibilities considered or attempted to solve the text editor issues in PiSharp:

1. Rewrite all of PiSharp window handling to use GTK. Mono provides a GTK# GUI toolkit which is built on top of the GTK+. The MonoDevelop IDE uses GTK# as its UI framework. Then, the new GtkWigdet-type Scintilla object could be wrapped by C# code and directly embedded into a GTK-based UI container once the PiSharp UI framework was completely rewritten with GTK#. However, the PiSharp project is too big. The entire PiSharp interface including all top-level windows and pop-up windows uses the capabilities of Mono WinForms. Meanwhile, as the container functionality supplier of the Scintilla-GTK widgets, the WeifenLuo.WinFormsUI project would also need to be rewritten to be compatible with GTK#.

2. Maintain two widgets for each editor instance, one a docked ScintillaNET object and the other a separate Scintilla-GTK widget, which would not be visible. Use the Scintilla-GTK widget as a message processor to receive, process and store all Scintilla Notifications sent from ScintillaNET, and build callback functions for each of the message allowing them to send back to ScintillaNET and replicate the appearance of the real Scintilla-GTK widget. To complete this option, it is necessary to understand how Mono WinForms presents a window layout. This

method could have a heavy overhead because it processes all Scintilla messages twice(ScintillaNET-ScintillaGTK-ScintillaNET) and might give a bad performance. Considering the relatively low-performance of the Raspberry Pi toolkits in CPU and memory, this method was implemented in the PiSharp on Raspbian.

3. Create an individual top-level GTK Window to hold a Scintilla-GTK object without window docking and keep it separate from the PiSharp window. As described in Section 5.1, the text editor on QuickSharp allows users to "tear off" dockable windows(text editor) and run with this style, so the resulting system of PiSharp is not completely unlike QuickSharp. Compared with solution 2, this method sacrifices some appearance consistency but only requires normal the notifications sending back to the ScintillaNET and forward to the parent window, to allow PiSharp to respond to the Scintilla-GTK text editor behaviours. Eventually, the PiSharp project decided to utilize to use this method. Figure 38 illustrates the architecture of PiSharp Text Editor on Raspbian.

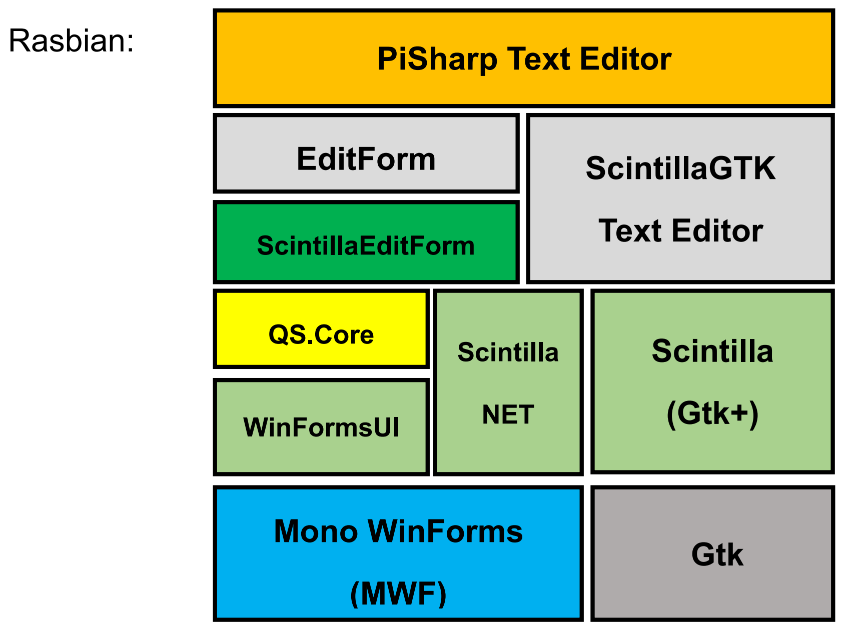
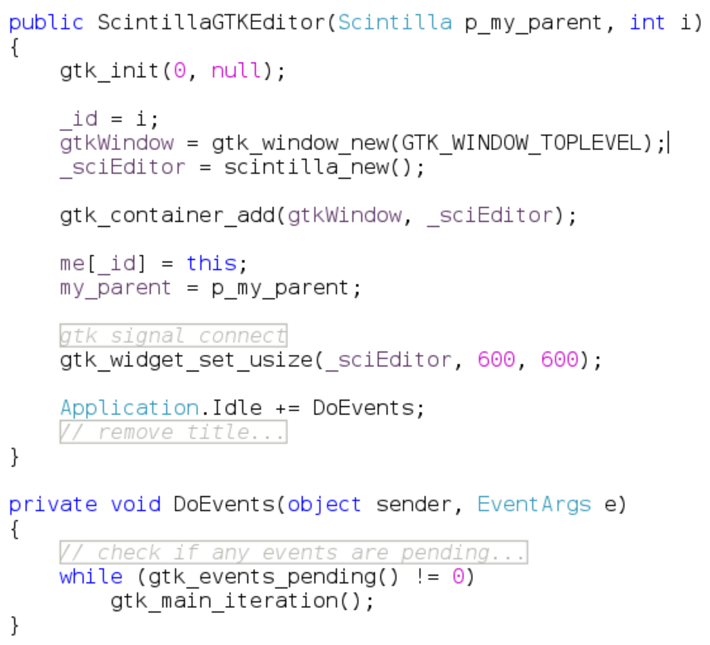


Figure 38 Architecture of PiSharp Text Editor on Raspbian

5.3.2. Scintilla-GTK Text Editor Creation

In Section 5.2, a sample program demonstrates how C# code invokes unmanaged functions from both GTK+ library and Scintilla-GTK library via the Mono implementation of interoperability to present a top-level GTK window with attached a Scintilla widget. The sample program manages and customizes some Scintilla editing features such as inserting additional text, assigning a lexical analyzer and highlight specific characters and strings. Similarly, PiSharp project was extended to define a new class "ScintillaGTKEditor" which contains the same methods as the sample Scintilla text editor created within the GTK environment. Some unmanaged functions are wrapped by C# methods as class properties so that a Scintilla object can respond to requests from other classes. For example, ShowGtk() wraps the unmanaged "gtk_widget_show_all()" function which shows a GTK window and its child widget – Scintilla widget. The EditForm class calls ShowGTK() after all Scintilla configuration is completed, to ensure that a Scintilla text editor will be displayed as EditForm defines rather than shown with default settings. The system shows an individual Scintilla-GTK window when PiSharp attempts to create a Scintilla control from ScintillaNET for a document or a plain text editor. It also tries to generate an instance of ScintillaGTKEditor to initialize the GTK set up for both a GTK window and a Scintilla widget. During setup "scintilla_send_message()" is never used. Internal ScintillaGTKEditor class methods are used to initialize the attributes and features of Scintilla. After initialization, as described in subsection 5.3.1, the all the wrapped "scintilla_send_message()" functions in ScintillaNET can be sent to the Scintilla-GTK widget.

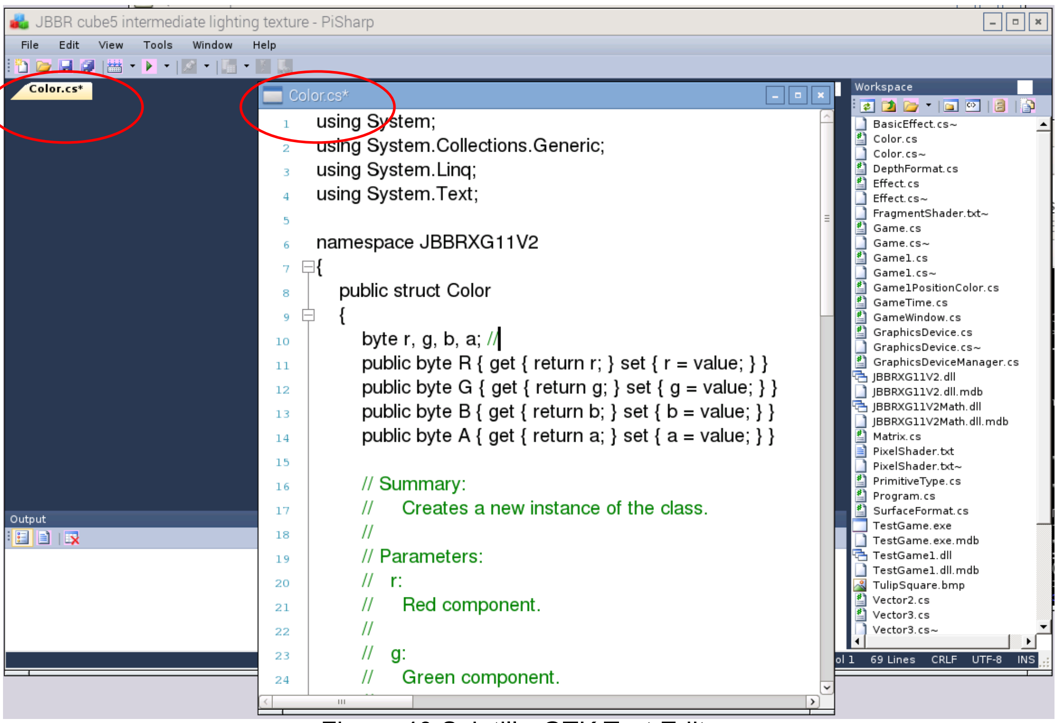
Running two kinds of window causes a problem with message handling. For every GTK application, a "gtk_main()" is used to run a UI main loop. gtk_main() listens and passes events to GTK widgets, e.g. a button pressed. Similarly, a Mono WinForms application runs in its own message loop. The PiSharp system associates with both the Mono WinForms-based main window and the GTK window. These separate windows each have their own message loops for handling events. Only one message loop can run at a time, while the other events just wait in their queue and their windows do not respond. For this problem, the first solution was to use a WinForms method – Application.DoEvent() and a GTK method – gtk_main_iteration() together in a single infinite loop, which could process all pending events and force UI update for the two windows. The loop contained a short delay to avoid excessive processor load. This method worked until the Mono WinForms window display a dialogbox. For example, when an auto-completion window(IntelliSense) of PiSharp was popped up, the GTK window was interrupted and became invisible(screenshot). The reason was that the Mono WinForms dialogue processor uses a different main loop, which pushed the PiSharp system back to the previous situation. The final solution was to use an Application.Idle() event to handle the gtk_main_iteration() as a callback function. Application.Idle() is triggered when the main window finishes processing thus GTK widget processes all its pending events when the main window is in the idle state. Then gtk_main_iteration() is blocked and waits for upcoming events, and so repeatedly. Figure 39 demonstrates the method that maintains both WinForms and GTK UI main loop.



```
public ScintillaGTKEditor(Scintilla p_my_parent, int i)
{
    gtk_init(0, null);

    _id = i;
    gtkWindow = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    _sciEditor = scintilla_new();

    gtk_container_add(gtkWindow, _sciEditor);

    me[_id] = this;
    my_parent = p_my_parent;

    gtk signal connect
    gtk_widget_set_usize(_sciEditor, 600, 600);

    Application.Idle += DoEvents;
    // remove title...
}

private void DoEvents(object sender, EventArgs e)
{
    // check if any events are pending...
    while (gtk_events_pending() != 0)
        gtk_main_iteration();
}
```

Figure 39 WinForms and GTK UI main loop maintain

After this, a GTK Window contained Scintilla-GTK widget can be successfully shown separate from the PiSharp window. Figure 40 shows the layout of the PiSharp opening Scintilla-GTK Text Editor. Additionally, a “gtk_window_set_title(IntPtr w, string s)” is used to assign the title bar text of a GTK window with current active document name, otherwise it defaults as “mono”. Even more, when a user changes document content, the title bar text adds a “*” until the document is saved, which synchronize with the tab text at the top of client window.

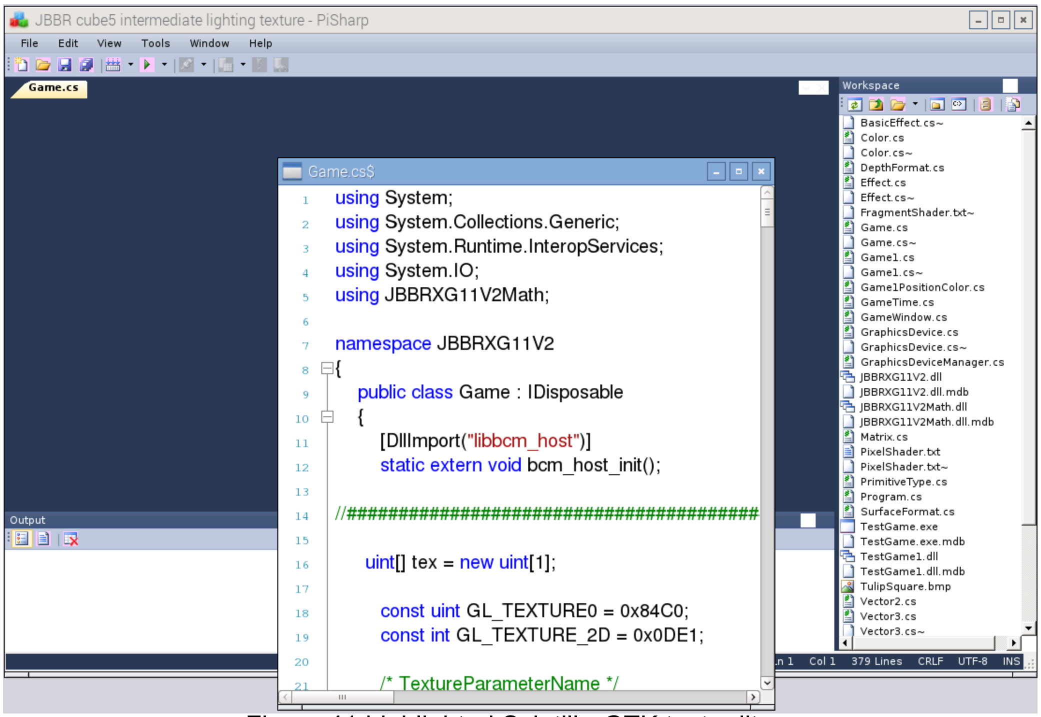


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace JBBRXG11V2
{
    public struct Color
    {
        byte r, g, b, a; //
        public byte R { get { return r; } set { r = value; } }
        public byte G { get { return g; } set { g = value; } }
        public byte B { get { return b; } set { b = value; } }
        public byte A { get { return a; } set { a = value; } }

        // Summary:
        //     Creates a new instance of the class.
        //
        // Parameters:
        //   r:
        //     Red component.
        //
        //   g:
        //     Green component.
```

```
BasicEffect.cs~
Color.cs
Color.cs~
DepthFormat.cs
Effect.cs
Effect.cs~
FragmentShader.txt~
Game.cs
Game.cs~
Game1.cs
Game1.cs~
Game1PositionColor.cs
GameTime.cs
GameWindow.cs
GraphicsDevice.cs
GraphicsDevice.cs~
GraphicsDeviceManager.cs
JBBRXG11V2.dll
JBBRXG11V2.dll.mdb
JBBRXG11V2Math.dll
JBBRXG11V2Math.dll.mdb
Matrix.cs
PixelShader.txt
PixelShader.txt~
PrimitiveType.cs
Program.cs
SurfaceFormat.cs
TestGame.exe
TestGame.exe.mdb
TestGame1.dll
TestGame1.dll.mdb
TulipSquare.bmp
Vector2.cs
Vector3.cs
Vector3.cs~
```

Figure 40 Scintilla-GTK Text Editor

At this time, Scintilla-GTK widget cannot fully highlight for the C# keywords. A straight forward way is to set the Scintilla LEXER as “cpp” type which contains C# language analyzer, and then the document properly highlights the C# keywords for the text in Scintilla-GTK widget. The Figure 41 shows a highlighted C# program in Scintilla-GTK text editor.



```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.IO;
using JBBRXG11V2Math;

namespace JBBRXG11V2
{
    public class Game : IDisposable
    {
        [DllImport("libbcm_host")]
        static extern void bcm_host_init();

//##########################################

      uint[] tex = new uint[1];

        const uint GL_TEXTURE0 = 0x84C0;
        const int GL_TEXTURE_2D = 0x0DE1;

        /* TextureParameterName */
```

Figure 41 highlighted Scintilla-GTK text editor

5.3.3. Notification system in PiSharp

As a control is a child window, the events occur in the control will be delivered to its container(parent window). Because there is no Scintilla control on the Editform(tabbed container), no notification messages from the ScintillaNET-based control proceed. In contrast, the Scintilla-GTK widget uses signals and callback functions to emit events that handle the native messages(SCI_*) and notification(SCN_*) by itself. Above all, ScintillaNET will never receive any Scintilla-GTK notify message.

Originally, ScintillaNET self-handled some event activities to accomplish editing features, including indicator clicking, text adding and so forth, by send WM_NOTIFY to the parent window and receiving reflection. To keep those features working properly, the PiSharp project intends to make a channel(signal emit) for the notifications so that ScintillaNET can respond to the notifications from a Scintilla-GTK widget and work as usual.

On the Windows platform, notifications are passed by using the WM_NOTIFY or WM_COMMAND messages with the window message mechanism. The GTK environment uses signal and callbacks system, in which notifications are sent with “sci-notify” signals to a GTK widget. It then connects the callback to process as defined in the callback function. The gtk_signal_connect_full(g_signal_connect) function enables sending Scintilla notifications from ScintillaGTK to ScintillaNET. The PiSharp project utilizes this method to emit all notifications triggered in the Scintilla-GTK widget.

5.3.4. GDK Keys Mapping.

Another problem occurring in the Scintilla-GTK widget is the key control in which only characters can be typed in the text editor. In other words, the Scintilla-GTK editor does not respond to any key bindings or key commands, such as “delete”, “enter”, “ctrl+c”, “escape”, and so on. The native Scintilla API provides key control of a text editing application in both text input and commands binding on all Scintilla supported platforms. As a wrapper around Scintilla API, ScintillaNET intends to take over all the command bindings from the Scintilla component. In this case, ScintillaNET removes all keyboard command mapping by setting an empty mapping table so that it can rebuild a new key mapping table to handle ScintillaNET specific commands for conflicts with the native Scintilla commands.

The problem is that the PiSharp text editor is built on top of GTK+, which uses the GDK key map. Additionally, key binding interactivity between ScintillaNET and Scintilla-GTK did not exist. More specifically, ScintillaNET does not know what keys a Scintilla-GTK widget has inputted; likewise Scintilla-GTK does not know how a key binding is defined in ScintillaNET.

To solve this issue, PiSharp constructs a key mapping between the Windows Forms(Mono WinForms) key codes and the GDK key codes. Furthermore, Scintilla-GTK sends each key-press event back to ScintillaNET when Scintilla-GTK receives a “key_press_event” signal, and tells ScintillaNET whether the key input is paired with “SHIFT”, “CTRL” or “ALT”. So that ScintillaNET is possible to respond with the corresponding command.

5.3.5. Scintilla-GTK Window Close

For the reason that PiSharp presents a text editor as a separate window, closing a Scintilla-GTK text editor can be done in multiple ways, because the PiSharp text editor requires the independent Scintilla GTK window, the tabbed container window and an invisible ScintillaNET object simultaneously exist. Therefore, PiSharp needs a bi-directional closure from the both the tabbed container window and the GTK window and shut down all the three object in one user closing event.

In QuickSharp, when a user closes a text editor, the container sends a "close" message to notify the child window – Scintilla control to dispose itself. According to this, the PiSharp forwards a procedure call which wraps the GTK+ closing function "gtk_widget_destroy(gtkWindow)" to close the GTK window and release all objects it holds. Conversely, when a close event triggered from a GTK window, the GTK window receives a "delete-event" signal. Then GTK window sets a handler for callback function to trigger a custom event called "SciWindow_CloseEvent". When the container receives this event, it processes to close itself and release resource of ScintillaNET. In either case PiSharp successfully completes the closing of a text editor window.

Chapter 6. “Build” System

The primary capability of an IDE that differs from a standard text editor is that the IDE is capable of compiling, running and debugging programs written in various programming languages. This section will describe the original QuickSharp compile and run configurations and how PiSharp manages the build process. It also covers altering the build system from the Microsoft .NET Framework on Windows to be compatible with the Mono platform on Raspbian and introduces an approach to enhancing the existing build process for a complex project architecture.

In a general compilation process, a compiler compiles programmer-readable code(also called source code)written in C, VB or other high-level programming languages directly into machine code(ignoring the assembly stage). In contrast, Common Language Infrastructure(CLI) standard languages, e.g. C#, can be compiled from valid source code to a second, platform-neutral language called Common Intermediate Language(CIL).

Figure 42 uses C# language source as an instance to illustrate a simplified overview of a .NET compilation and execution process. Firstly, a C# program is loaded by a corresponding compiler provided in a specific system environment. Secondly, the compilation process starts and compiles the source code into CIL code(called “bytecode”). Afterwards, a platform-specific Common Language Runtime(CLR) is responsible for executing. The JIT(Just-In-Time) compiler of the platform CLR reads CIL code and compiles it into machine code(also called native code) that can be executed. Finally, the machine code compiled by the JIT compiler is executed by operating system services and outputs the result that the developer requires.

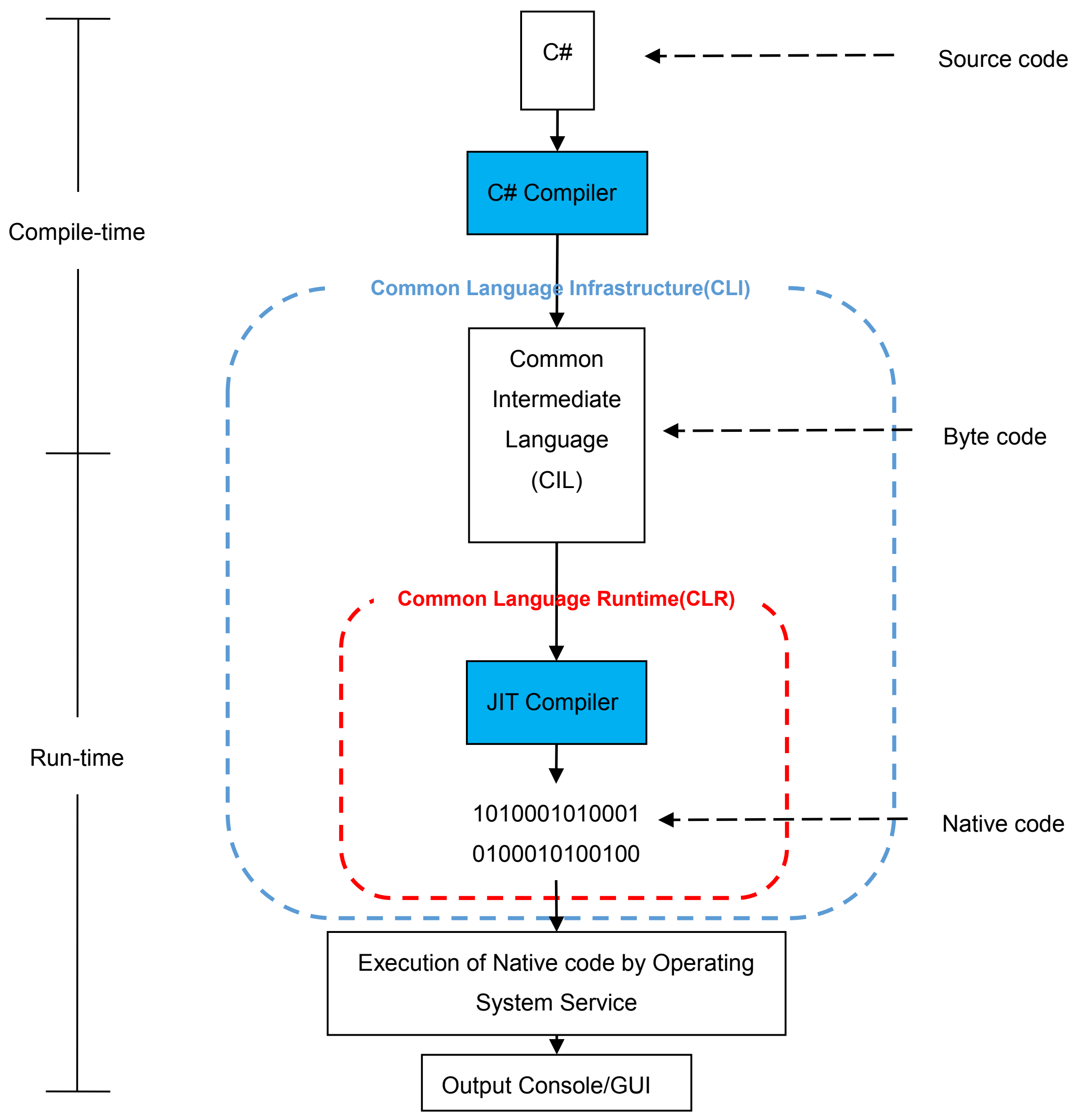

C#
Source code
C# Compiler
Compile-time
Common Language Infrastructure(CLI)
Common Intermediate Language (CIL)
Byte code
Common Language Runtime(CLR)
JIT Compiler
Run-time
1010001010001
0100010100100
Native code
Execution of Native code by Operating System Service
Output Console/GUI

Figure 42 .NET compilation and execution process

The compile time and runtime operations shown in Figure 42 can be performed on different CPU's and/or different operating systems. The CIL is system-independent, in particular, it is possible to compile on x86/x64 Windows platform and run on ARM-based Raspberry Pi with Linux distributions, or vice versa.

In QuickSharp, the entire build system is constructed in three parts: a BuildTool module, Language Support modules and an Output module. In fact, both BuildTool module and Language Support modules are responsible for determining and completing the members of the target build-command. The concrete build process is integrated into the Output module. The following Table 4 lists the generic members of a BuildCommand class.[24]

Member	Description
BuildTool	The build tool(compiler or runtime) used in the build command
SourceInfo	Information about the input source file
SourceText	The source code supplied to the build tool
TargetInfo	Information about the output file
TargetType	The document type of the output file
Path	The expanded file path of the build tool
Args	The expanded arguments pass to the build tool
StartText	Text displayed before the tool runs
FinishText	Text displayed after the tool runs
Cancel	Flag used to allow the command to be cancelled
CancelResult	Result to be returned by the command in the event of the cancellation
SuccessCode	Return code to determine successful completion of a build command

Table 4 BuildCommand Class Member

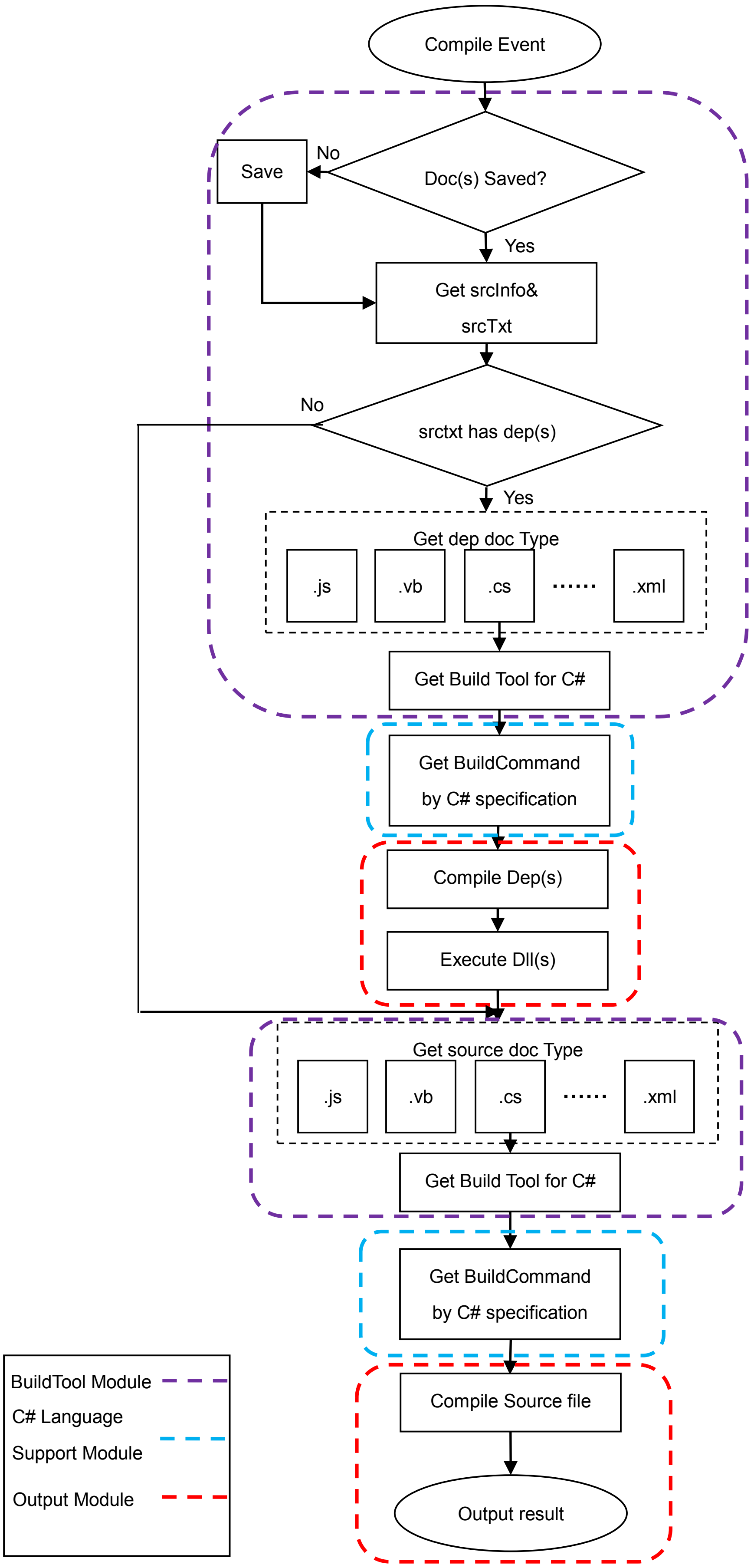


Figure 43 QuickSharp build process

The Figure 43 demonstrates a QuickSharp build process workflow with C# source code. Similar to other mainstream IDEs, while one or multiple documents are active, a "Compile" event in QuickSharp can be triggered by one of following the main window provided interactions: simply pressing keys with "Shift+F5"; clicking "Compile" under a drop-down menu or clicking the corresponding button on the toolbar. As a preparation for the actual build process, the BuildTool module then obtains a list of active documents current open in the Scintilla editor. The information on currently active documents will be saved in the BuildCommand class. This includes file path, file name, file type, document time stamp and the file content. In addition, the BuildTool module can detect whether there is any other dependent source file mentioned in the source text by the QuickSharp build management mechanism(embedded options). If any dependencies exist in the same directory as the input source file, the build process may compile the each dependency conditionally if that file is out of date. Otherwise, the input file will be compiled directly. The input file and dependencies use the same procedure calls to complete the build process.

In QuickSharp, the IDE presents a multi-language support build environment. The support for each available language is developed as a plugin that registers a corresponding build command provider to the modular architecture IDE when it starts up. In addition, some build tool configurations for a language could be created separately by different build tool versions or both "Compile" and "Run" actions. A build tool is defined by a specific configuration associated with the document type and current action. The build command delegate representing an abstract tool retrieves a build command from a provider supplied by a language support plugin which provides a method to create an actual build-command. When the delegate is invoked, a specific build tool configuration with its corresponding build command method allows it to be converted into a concrete build-command for execution in the output console window. Figure 44 describes the build-command invocation process for the C# language in brief.

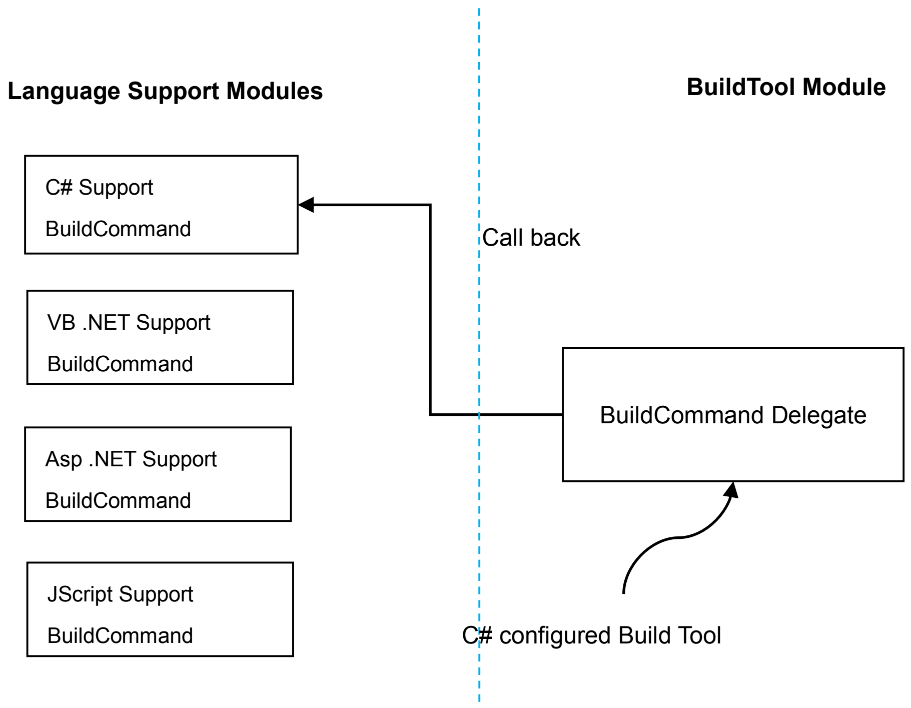


Figure 44 Concrete BuildCommand invocation process

Eventually, the Output module takes over the compilation process and executes the completed build command. The compiler will run and present the results as well as the error messages for syntax errors and warnings of the program if necessary in the output window.

6.1 Build-tool Replacement

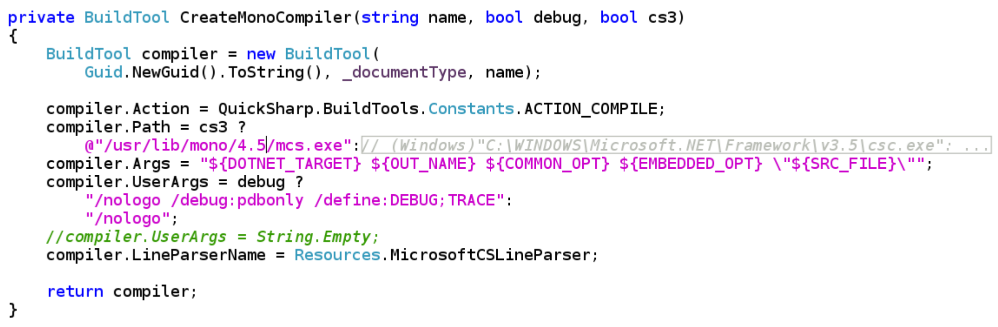
The official QuickSharp IDE builds a C# program using the Microsoft .NET Framework C# compiler and runtime on the Windows platform. PiSharp intends to make the QuickSharp IDE run on Raspberry Pi(R-Pi) to compile and run XNA-like programs written in the C# language. In the new system, the PiSharp will take advantage of Mono Project components. Mono provides Mono version C# compilers and an implementation of the CLI standard runtime. The intention of the Mono project is to allow developers to build cross-platform C# applications(on Windows, Linux or other OS).

The Mono project also provides a Windows version of the Mono platform and development tools. QuickSharp made an effort to be compatible with Mono. Once the Mono components(in particular the compilers and run tool) are installed on a Windows system, the Mono Language support module in QuickSharp will provide a default configuration for them. As the Microsoft .NET Framework C# compilers and runtime do not exist on any Linux distributions, in this case Raspbian, the Mono C# compilers and run tool directly replace the default configurations of Microsoft C# build tools in PiSharp. As described in the Mono background section above, “mcs.exe” represents the most commonly used and functional compiler in current Mono versions so far. The instruction expression of C# program compilation based on Mono compiler is:

```
mcs [option] [source files]
```

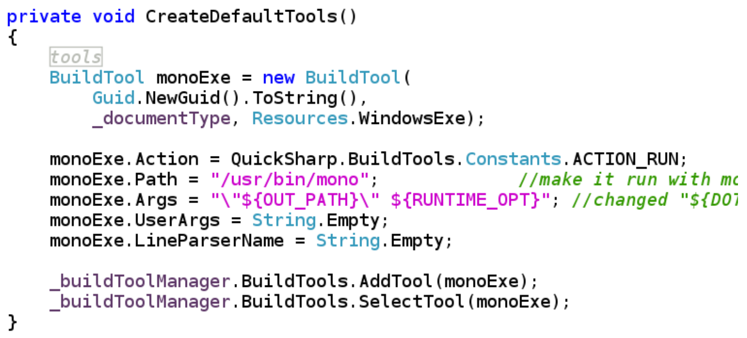
C# source files ends with a “.cs” extension.

Developers can pass one or more options to drive the compiler and it is possible to compile multiple source files into one output file. Figure 45 and Figure 46 shows the code for the default compiler and run tool configuration replacement as implemented in PiSharp.



```
private BuildTool CreateMonoCompiler(string name, bool debug, bool cs3)
{
    BuildTool compiler = new BuildTool(
        Guid.NewGuid().ToString(), _documentType, name);

    compiler.Action = QuickSharp.BuildTools.Constants.ACTION_COMPILE;
    compiler.Path = cs3 ?
        @"/usr/lib/mono/4.5/mcs.exe":// (Windows)"C:\WINDOWS\Microsoft.NET\Framework\v3.5\csc.exe": ...
    compiler.Args = "${DOTNET_TARGET} ${OUT_NAME} ${COMMON_OPT} ${EMBEDDED_OPT} \"${SRC_FILE}\"";
    compiler.UserArgs = debug ?
        "/nologo /debug:pdbonly /define:DEBUG;TRACE":
        "/nologo";
    //compiler.UserArgs = String.Empty;
    compiler.LineParserName = Resources.MicrosoftCSLineParser;

    return compiler;
}
```

Figure 45 configuration of default compiler tool in PiSharp

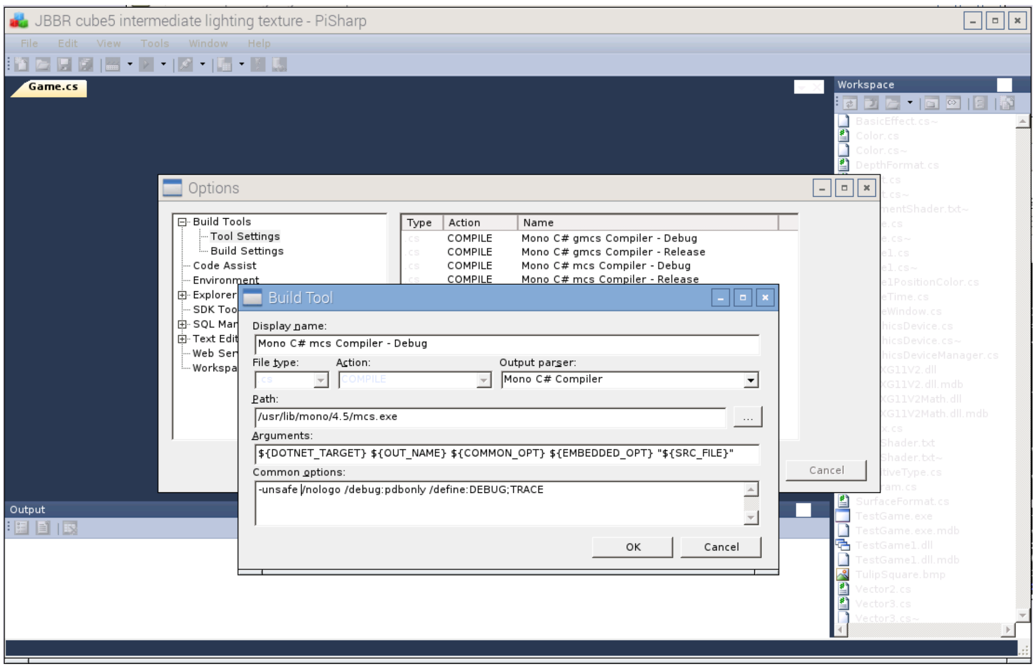


```
private void CreateDefaultTools()
{
    tools
    BuildTool monoExe = new BuildTool(
        Guid.NewGuid().ToString(),
        _documentType, Resources.WindowsExe);

    monoExe.Action = QuickSharp.BuildTools.Constants.ACTION_RUN;
    monoExe.Path = "/usr/bin/mono";             //make it run with mo
    monoExe.Args = "\"${OUT_PATH}\" ${RUNTIME_OPT}"; //changed "${DOT
    monoExe.UserArgs = String.Empty;
    monoExe.LineParserName = String.Empty;

    _buildToolManager.BuildTools.AddTool(monoExe);
    _buildToolManager.BuildTools.SelectTool(monoExe);
}
```

Figure 46 configuration of default run tool in PiSharp

Windows Form Applications cannot be compiled directly by early versions of the Mono C# compiler. This is because the Mono C# compiler only references three assemblies: mscorlib.dll, System.dll and System.Xml.dll by default. Developers who want to refer to more comprehensive libraries must manually specify them using the "-pkg" flag or the "-r" flag, or a straightforward way is to use the "-pkg:dotnet" command line option to get all the .NET base libraries immediately. This gives access to a set of libraries similar to those commonly available on a Windows system for writing GUI programs. In particular, it gives access to System.Windows.Form.dll. The current Hard-Float Mono version(4.5) "mcs" compiler on Raspbian can refer System.Windows.Form library by default.

Quicksharp provides a flexible and convent system allowing developers customize build tool configurations: such as a change to a new build tool or specifying an argument for a build tool. The build tool system expands the configuration for build tool paths and arguments by replacing template macro texts with instance actual values. Figure 47 presents the BuildTool option window layout and a Mono C# build tool configuration.



Type	Action	Name
.cs	COMPILE	Mono C# gmcs Compiler - Debug
.cs	COMPILE	Mono C# gmcs Compiler - Release
.cs	COMPILE	Mono C# mcs Compiler - Debug
.cs	COMPILE	Mono C# mcs Compiler - Release

Display name: Mono C# mcs Compiler - Debug
File type: .cs
Action: COMPILE
Output parser: Mono C# Compiler
Path: /usr/lib/mono/4.5/mcs.exe
Arguments: ${DOTNET_TARGET} ${OUT_NAME} ${COMMON_OPT} ${EMBEDDED_OPT} "${SRC_FILE}"
Common options: -unsafe /nologo /debug:pdbonly /define:DEBUG;TRACE

Figure 47 BuildTool Setting Window

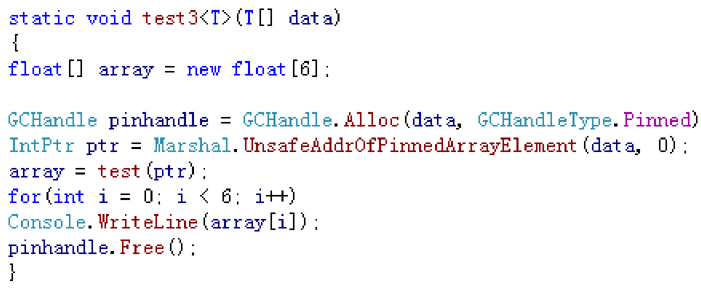
As shown above, a C# build tool argument described with macros is:

${DOTNET_TARGET} ${OUT_NAME} ${COMMON_OPT} ${EMBEDDED_OPT} “${SRC_FILE}”

- ${DOTNET_TARGET}: a tool specific macro for .NET compilation determines a target output file type “.dll” or “.exe” by setting option flags: “/t:library” or “/t:exe” respectively;
- ${OUT_NAME}: a target output file name;
- ${COMMON_OPT}: option flags for building;
- ${EMBEDDED_OPT}: options obtained from QuickSharp’s native features to manage the build process embedded in source code;
- ${SRC_FILE}: source file name

The full collection of generic macros is listed in Appendix D. Using this system, add a “-pkg:dotnet” flag will pass to the build tool configuration in place of the ${COMMON_OPT} macro in arguments.

Another command option used in this project is “-unsafe”. This command option is required by the Pi-XNA project. As he mentioned, his test programs are trying to access data in an array of a Vertex structures from the Main function. To achieve this goal, pointers are needed. An example is shown in Figure 48. Because pointers in C# programs are considered to be unsafe code, the class also has to be declared as unsafe[15].So, developers who are programming Pi-XNA code must add the "-unsafe" flag as a common option manually to avoid failures in build processes(See in Figure 47).



```
static void test3<T>(T[] data)
{
float[] array = new float[6];

GCHandle pinhandle = GCHandle.Alloc(data, GCHandleType.Pinned);
IntPtr ptr = Marshal.UnsafeAddrOfPinnedArrayElement(data, 0);
array = test(ptr);
for(int i = 0; i < 6; i++)
Console.WriteLine(array[i]);
pinhandle.Free();
}
```

Figure 48 C# code using unsafe pointer[15]

6.2 PiSharp Build Process Enhancements

As this thesis explained in Section 6.1, the QuickSharp IDE manages the build process and develops programs by embedded options rather than using configuration files like project files or solution files. This method provides a flexible modular structure. It is helpful that a compiled program can be organized into a main program and a number of libraries. This ensures that components in the modular structure project can be modified or added without affecting the rest. However, QuickSharp does not provide a way of assembling sets of source files into libraries. Instead, if a library structure is needed, it will make a separate library for each source file. For graphic programming, this is not ideal, because the number of source files can be large and loading many libraries leads to slow program start-up. A goal of the PiSharp is to provide a better library system. This section shows the development of the PiSharp build system.

Whist, it might be appropriate for professional developers to maintain a program architecture by manually adding embedded options to source code, this could be complicated and challenging work for beginners. For example, if a user builds from Wang's(2014) Pi-XNA sample program[38] with QuickSharp on a Raspberry Pi, all the mathematics classes, graphic effects providers(developed in Wang's project), and the current program references have to be embedded into the source text by hand. Figure 49 demonstrates output of a 3D test program from Wang's project: a rotating textured box under a lighting system running Raspberry Pi. Figure 50 shows all relevant source files of the rotating textured box program referenced in a source file to compile with the original QuickSharp build system, and generate a "Program.exe" execuTable The compile options can be assigned to the full path of C# file.

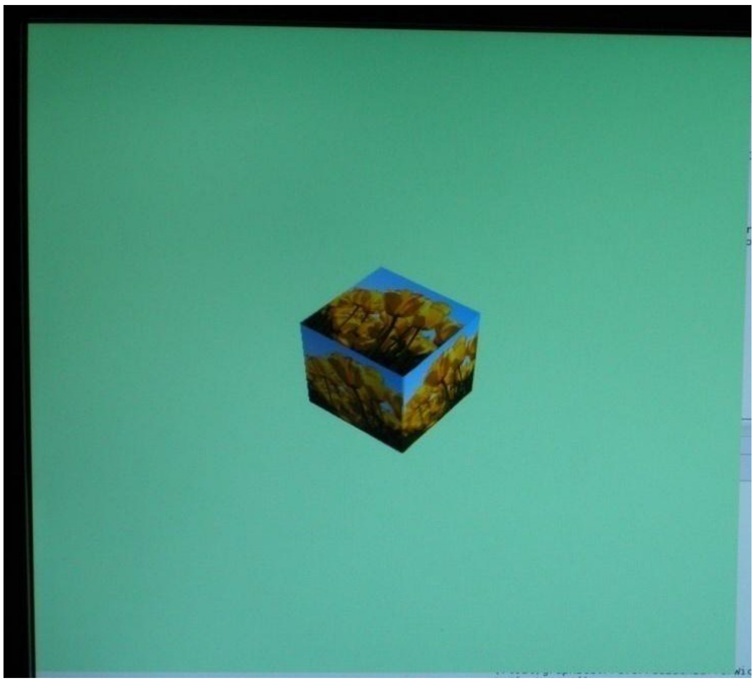
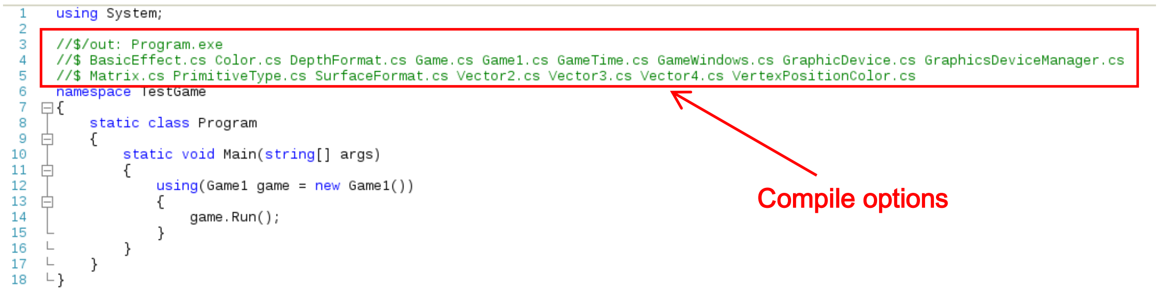


Figure 49 Rotating Textured Box under a Lighting System[38]



```
using System;

//$/out: Program.exe
//$ BasicEffect.cs Color.cs DepthFormat.cs Game.cs Game1.cs GameTime.cs GameWindows.cs GraphicDevice.cs GraphicsDeviceManager.cs
//$ Matrix.cs PrimitiveType.cs SurfaceFormat.cs Vector2.cs Vector3.cs Vector4.cs VertexPositionColor.cs
namespace TestGame
{
    static class Program
    {
        static void Main(string[] args)
        {
            using(Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
}
```

Figure 50 embedded compile options for “Rotating Texture Box” Program

Without an IDE, users have to build a program in a command prompt manually allocating a proper compiler, source files and all required references. The Build command auto-completion(source files and libraries aspect) of QuickSharp requires them in compile options. However, if the project contains a number of source files, manually defining each source file is risky and inefficient.

In fact, an Pi-XNA C# program structure contains three main parts: A mathematical component(XNA Mathematics Classes) consisting of Matrix, Vector2, Vector3, Vector4 classes to represent the fundamental needs of an XNA Framework, which supports mathematical calculations for coordinate system transformation from a 3D virtual environment to a 2D screen; Modules like GraphicsDevice, VertexPositionColor,

GameWindow, PrimitiveType represents graphics components(XNA graphics Classes). They may require mathematical calculations to handle primitives(points, lines or triangles) and provide 3D functionality; The custom program(XNA Program) developed by application programmer creates concrete 3D objects and animations on the screen.
A Pi-XNA program class hierarchy is illustrated in Figure 51.

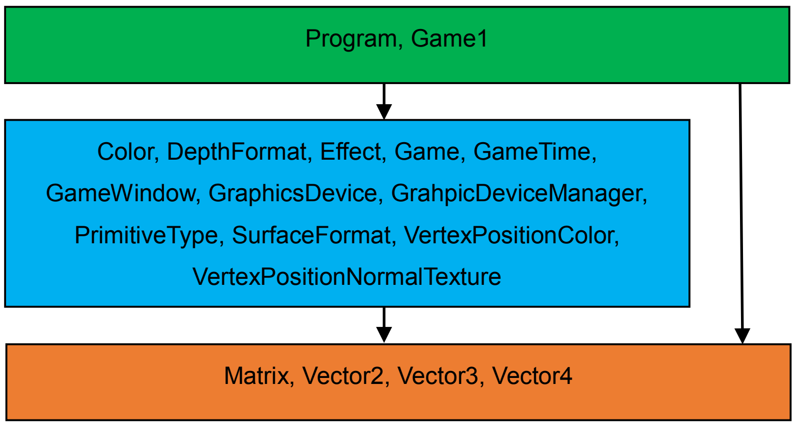

Program, Game1
Color, DepthFormat, Effect, Game, GameTime, GameWindow, GraphicsDevice, GrahpicDeviceManager, PrimitiveType, SurfaceFormat, VertexPositionColor, VertexPositionNormalTexture
Matrix, Vector2, Vector3, Vector4

Figure 51 Pi-XNA sample program class hierarchy

Apparently, QuickSharp's build process is only suited to experienced developers who want to maintain their project by themselves. Besides the difficulty of computer graphic development, solving program architecture in a complex case could bring more problems for beginners. In a complex program, compiling many of source files into multiple library files is easy to maintain, and it may also reduce compile time during development. In particular, the Pi-XNA program contains a number of math components and graphic components which developer will not modify very often(do not have to recompile all files every time, drawbacks: could be slow when program is running).

More importantly, in QuickSharp, dependent files can only be extracted from the current input source file. Any dependent files which contain their own dependencies are ignored. Obviously, specifying all the embedded options is not convenient to new programmers. In particular, a large number of source files give a complex program. Managing a program by the IDE itself could be a good idea for beginners, who can learn their program architecture gradually.

Therefore, this part is going to concentrate on build command completion for one or multiple source documents situations. Considering this ported IDE on Raspberry Pi toolkits is primarily for beginners, the PiSharp project intends to make the build process easier for beginners. In a multiple source file project scenario, a compilation process starts with an active source file. In the PiSharp, the build process is reorganized. The new method traverses every C# source file in the current directory and detects which one contains a "Main" program. The source file of compilation will be reset from the currently active file to the Main program file. The process then adds appropriate embedded options with referenced files' names at the top of the Main program file copy in the BuildCommand class. Therefore, the user will never see the embedded options.

As a result, PiSharp requires that developers create and place all their program source files in the same directory. The build system combines file names with the current directory to the valid full path of source files for the build tool configuration. Developers can use the "WorkSpace" or "Explorer" window to manage which the current directory is the build process working in.

In the compilation examples shown in this chapter, all source files for an XNA-like program are in a single directory. This allows experimentation with Math and Graphic source files. The system does allow a "closed" build of the "XNA libraries, by keeping their source code in a separate directory, and building one or more libraries that could be referenced with the "-pkg" flag from a single program. As development of the graphics library is ongoing, the option to efficiently modify the library in PiSharp is very convenient, and serves here to illustrate the flexibility of the build system.

The goal of PiSharp build system is to make the entire build processes into one instruction from the user. The optimization considered four different situations:

1. If the current input source file's content has any embedded compile option or dependency option, PiSharp assumes that the developer expects to maintain the program structure by him/her-self. This feature is as established by the

QuickSharp group, thus keeps it alive for initial purposes;

2. If the program is light-weight(less than 10 source files in total) or read source file uses the same namespace, PiSharp will auto complete the compile options and compile all the source files in the current working directory into a single executable file(assuming that all C# files in the working directory are part of the program);
3. If the program has relatively complicated structure(more than or equal to 10 source file in total, and has any underlying relationship in the project), PiSharp will reorganize the build processes order for each dependency and the program’s main body compilations. The program will generate all corresponding libraries and an executable file at once. The PiSharp build process regards the source files in the current working directory that have the same namespace as an individual library for the part of the condition definition;
4. If a “*.csproj” project file exists in the current opening folder, PiSharp prefers to build with the project file.

As part of this process, PiSharp adds an extra flag to set “compileProcessStatus” for BuildCommand functions(in all build command delegate, invocations, and all language providers methods) as a condition to identify the status of the current build process: the process is using the original build process; the process is using the PiSharp build process trying to generate an executable file; the process is using the PiSharp build process but trying to build a library. In the situation of the program having relatively complicated structure, the build process could use both “build exe” and “build dll” flags if the project does have a Main function for the entry point.

6.2.1. Build Process of Light-Weight Programs

A user may work on any source file while developing a program, not necessarily the one that contains the Main method. However, when the user starts to build and run the output executable program(.exe file), a library result might instead because the QuickSharp build process determines to compile a source file into an executable program or a library according to the input file content. If the source code contains a valid Main procedure, then an executable program will be created. Otherwise, a library will be the result. The QuickSharp solution is to fix a source file that the build command starts with. Figure 52 illustrates QuickSharp "pins" a source file for building. To do this, the user has to specify the source file with a Main function on the toolbar, with the two restrictions that the Main source file was already active in the QuickSharp editor window before compilation was requested, and that the other relevant files are specified in the Main source code file.

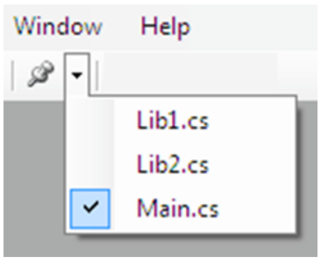


Figure 52 QuickSharp "pins" a source file for building

To simplify this, when the total number of source files is fewer than 10 or all source files are under a same namespace, the new PiSharp build process can start with any active source file in the IDE editor. It finds the source file with a Main function in the working directory and copies the Main program file's information and content(if the file exists) to the build command. Then it extracts all the other source file names in the same directory and embeds references to them as a series of compile options into the Main program's source text. The "source text" is a temporary string stored in memory as a part of the build command structure, which the C# language build tool can compile with, so that PiSharp build process will never change the actual source file content. Figure 53 demonstrates an example that a build process starting with a file "Class2.cs" currently active in the text editor. The build process compiles the "Main.cs" as the source file, associated with "Class1.cs", "Class2.cs" and "Class3.cs" files to generate a executable "Main.exe". A "/out:" option is used to specify the target file name. If there is no file with a Main program, PiSharp will build a library for instead, using the namespace as its name or "Program" if no namespace is used.

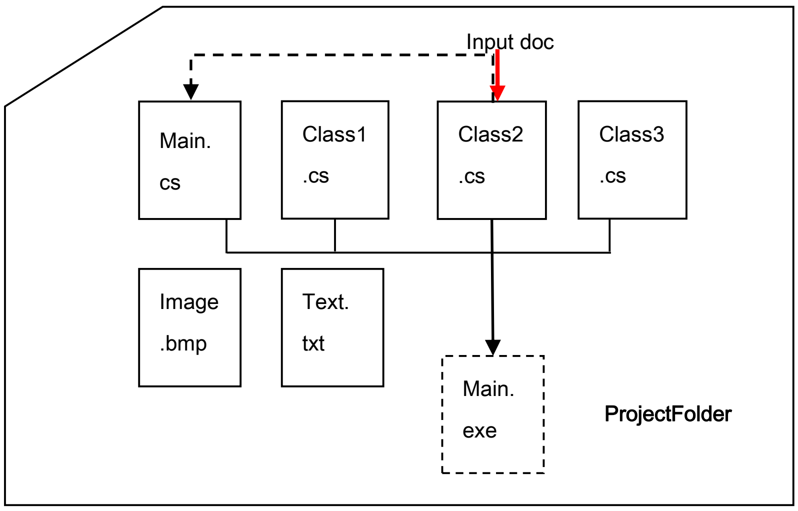


Figure 53 PiSharp build process in a working directory

Text embedded into Main.cs file is:

```
//$/out:Main.exe
//$ Class1.cs Class2.cs Class3.cs
```

The resulting build command will be:

```
mcs.exe /out:Main.exe Class1.cs Class2.cs Class3.cs “Main.cs”
```

This method will execute a single output file but recompile every source file. The other features such as runtime options, build task, resource, icon and “runinownwindow” as embedded options are retained for customization by the users because those options may still be of value.

6.2.2. Build Library for Complex Program Structure

QuickSharp prefers to compile each single source file to an individual library. However, that brings difficulty not only to programmers but also to the system, when the number of source files is large. It slows the program launch-in that the program has to retrieve and load all the libraries separately. Therefore, PiSharp modified this build system to make a proper library facility.

In some case, especially graphic programming, the project may consist of a large number of source files or be based on functional underlying assemblies(like mathematical or graphical providers). To avoid a long compilation time for each build action, the optimization plan is to compile those source files into a set of library files, which programmers will probably not change frequently. If total number of source files is more than 10 and the total files' namespaces number is greater than or equals 2(source files which have no namespace defined will have "Program" set as their namespace by the build process), the PiSharp compilation process will generate a library file for each sub-project based on their own namespace except for sub-project involved the Main program file. In brief, except for the project contains Main program file, the rest of the source files will be compiled into library files based on namespaces.

Figure 54 describes the structure of a simulated XNA-like program that satisfies the build libraries condition, which is complex enough to explain the entire PiSharp build process. From Figure 54 and Table 5, the total 11 source files are divided into six parts: Matrix.cs,Vector2.cs Vector3.cs and Vector4.cs constitute the fundamental mathematics part; Graphic.cs references the mathematics assembly; Line.cs and Point.cs requires the Graphic component. Triangle.cs references both mathematics and elements; Cube.cs is based on Triangle.cs; Program.cs and Game1.cs program are built on top of mathematics and Cube object class.

Source files	NameSpace	Assemblies Referenced
Matrix.cs,Vector2.cs Vector3.cs, Vector4.cs	Math	
Graphic.cs	GraphicComponents	Math.dll
Line.cs, Point.cs	GraphicElement	GraphicComponents.dll
Triangle.cs	ObjectBase	GraphicElement.dll, Math.dll
Cube.cs	Object	ObjectBase.dll
Program.cs, Game1.cs	Game	Object, Math.dll

Table 5 Sample Program Relationships

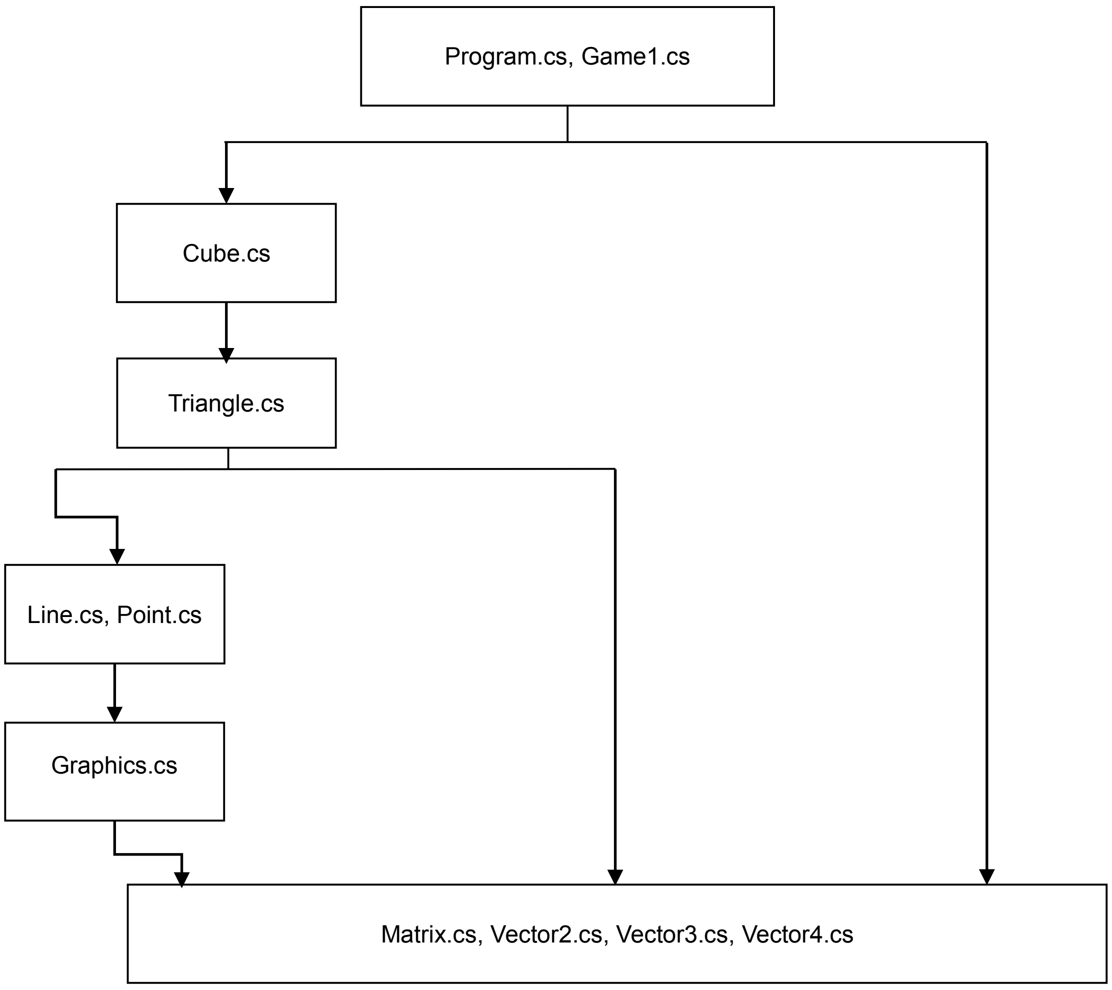


Figure 54 a sample program for testing PiSharp build process

A target file only can be executed if its required references exist. In this sample program the expected build process sequence is:

1. Compile: Matrix.cs, Vector2.cs, Vector3.cs, Vector4.cs into “Math.dll”;
2. Compile: Graphic.cs associated with “Math.dll” into “GraphicComponents.dll”;
3. Compile: Line.cs, Point.cs associated with “GraphicComponents.dll” into “GraphicElement.dll”;
4. Compile: Triangle.cs associated with “GraphicElement.dll” and “Math.dll” into “ObjectBase.dll”;
5. Compile: Cube.cs associated with “ObjectBase.dll” into “Object.dll”;
6. Compile: Program.cs Game1.cs associated with “Object.dll” and “Math.dll” into “Game.exe”.

Therefore, the subproject involves Main program does not have to be organized because it always build in the last.

To implement the goal of the PiSharp build process reorganization, the first step is to classify all the source files based on their namespaces. In order to group source files, a more detailed file information class is required than that used in QuickSharp. In PiSharp, the class is called "FileDetail" which involves the entire content of "FileInfo", in addition to FileText, Namespace and used Assemblies for the further group and sort method, and requiring all external libraries to be mentioned. Figure 55 shows the method of extracting the namespace from the source code, a parser was written by using clauses. It uses regular expression to match text formed as "namespace *". The "\s" and "\w" respectively represent whitespace and word character.

```
private string GetNameSpace (string srcText)
{
    srcText = RemoveComments (srcText);

    Regex re = new Regex (@"namespace\s+\w+([^?/:&""\*#;%|<>]\w+)*");
    Match m = re.Match (srcText);

    string nsp = m.Value.Trim ();
    try
    {
        int postion = nsp.IndexOf (" ");
        nsp = nsp.Substring (postion).Trim ();
    }
    catch
    {
        nsp = "Program";
    }
    return nsp;
}
```

Figure 55 Get Namespace method

The PiSharp uses a Language-Integrated Query(LINQ) "GroupBy" operator to take a collection of all dependencies with detail information as inputs, and group dependencies based on the namespaces. LINQ is part of the Microsoft .NET Framework 3.5 library system to make SQL-like facilities available in source code.

The "GroupBy" expression in this case is:

```
var groupedDeps= depfilesList.GroupBy(
        fDetail => fDetail.NameSpace).Select(group =>group.ToList)
```

Then the process has a set of grouped but unordered dependency lists based on namespaces. After that, add assembly library file information(if it exists) at the end of each list following the detail from Figure 54 and Table 5 to form the compilation requirement lists.

Generally, a build process determinates whether the list of source files needs to be compiled depended on two conditions:

1. "all the referenced libraries are up to date":

   No any referenced library/Each referenced library exists and its last modified time is greater than all its source files last modified time;
2. "the target file execution is required": the target file does not exist or its last modified time less than its source files last modified time.

The PiSharp build process sorts the order by comparing all lists from the groups with a List<T>.Sort(IComparer<T>) Method but customize the comparison rule with:
If the first input list satisfies both conditions but the other does not, the first one will stay in the front of the second one and vice versa, else they both keep their position. This ensures only one list can acquire the priority of compilation. Figure 56 and Figure 57 show the Compare method of sorting lists of files.

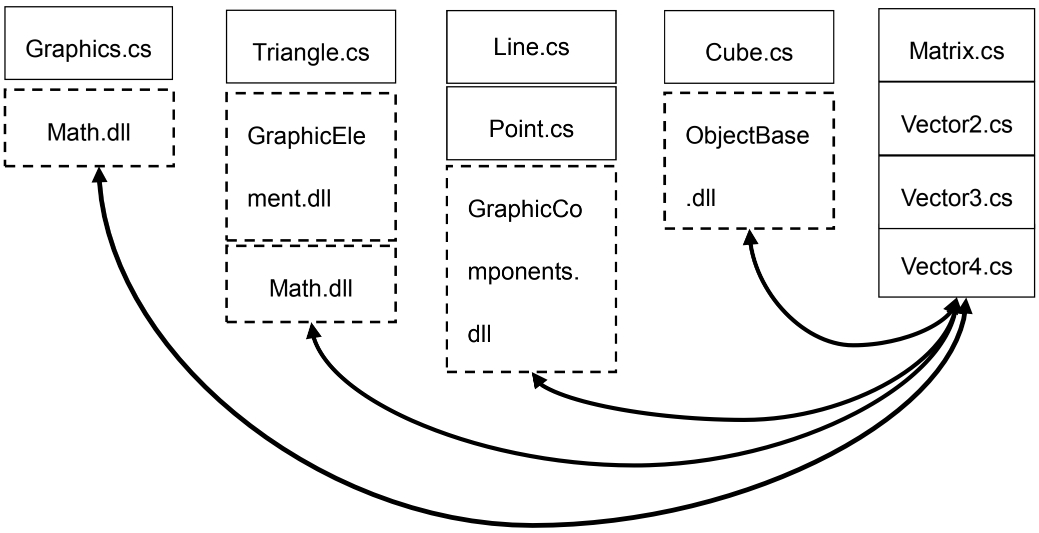
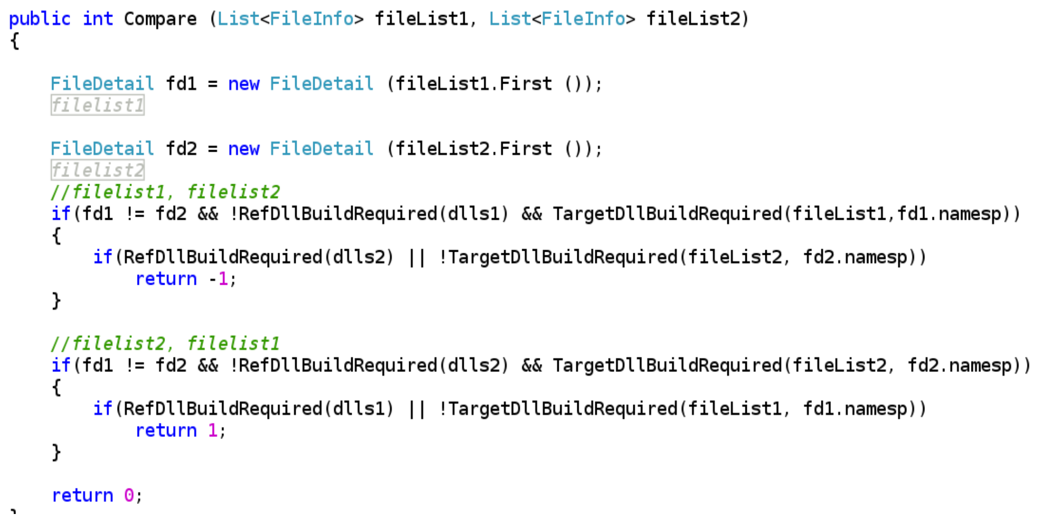


Figure 56 Form and Sort Compilation Requirement Lists



```
public int Compare (List<FileInfo> fileList1, List<FileInfo> fileList2)
{

    FileDetail fd1 = new FileDetail (fileList1.First ());
    filelist1

    FileDetail fd2 = new FileDetail (fileList2.First ());
    filelist2
    //filelist1, filelist2
    if(fd1 != fd2 && !RefDllBuildRequired(dlls1) && TargetDllBuildRequired(fileList1,fd1.namesp))
    {
        if(RefDllBuildRequired(dlls2) || !TargetDllBuildRequired(fileList2, fd2.namesp))
            return -1;
    }

    //filelist2, filelist1
    if(fd1 != fd2 && !RefDllBuildRequired(dlls2) && TargetDllBuildRequired(fileList2, fd2.namesp))
    {
        if(RefDllBuildRequired(dlls1) || !TargetDllBuildRequired(fileList1, fd1.namesp))
            return 1;
    }

    return 0;
}
```

Figure 57 Compare Method of Sorting

The result of the sort is to identify a library to compile first. It is not possible to completely compile entire the program at this stage because of the structure QuickSharp(retained in PiSharp). The sorting and complication planning is done in the C# language module and can only pass back one compilation command to the Build module at a time. Therefore, the idea is to make a loop to keep sorting after the first list from the group gets compiled. Figure 58 illustrates the PiSharp build library process. A new delegate is created in BuildTool module so that the process can get a sorted compilation requirement from the C# language support module after invocation. The process takes the first list from the group to build a library. After this, the Output module accomplishes the compile procedure and outputs result for this library. Then the process removes the first list and updates the rest of files information. This ensures that all files' last modified times are updated correctly. If necessary, the next list of files is compiled. Eventually, the build process will turn to compile the Main project and generate the executable file with all supplied libraries.

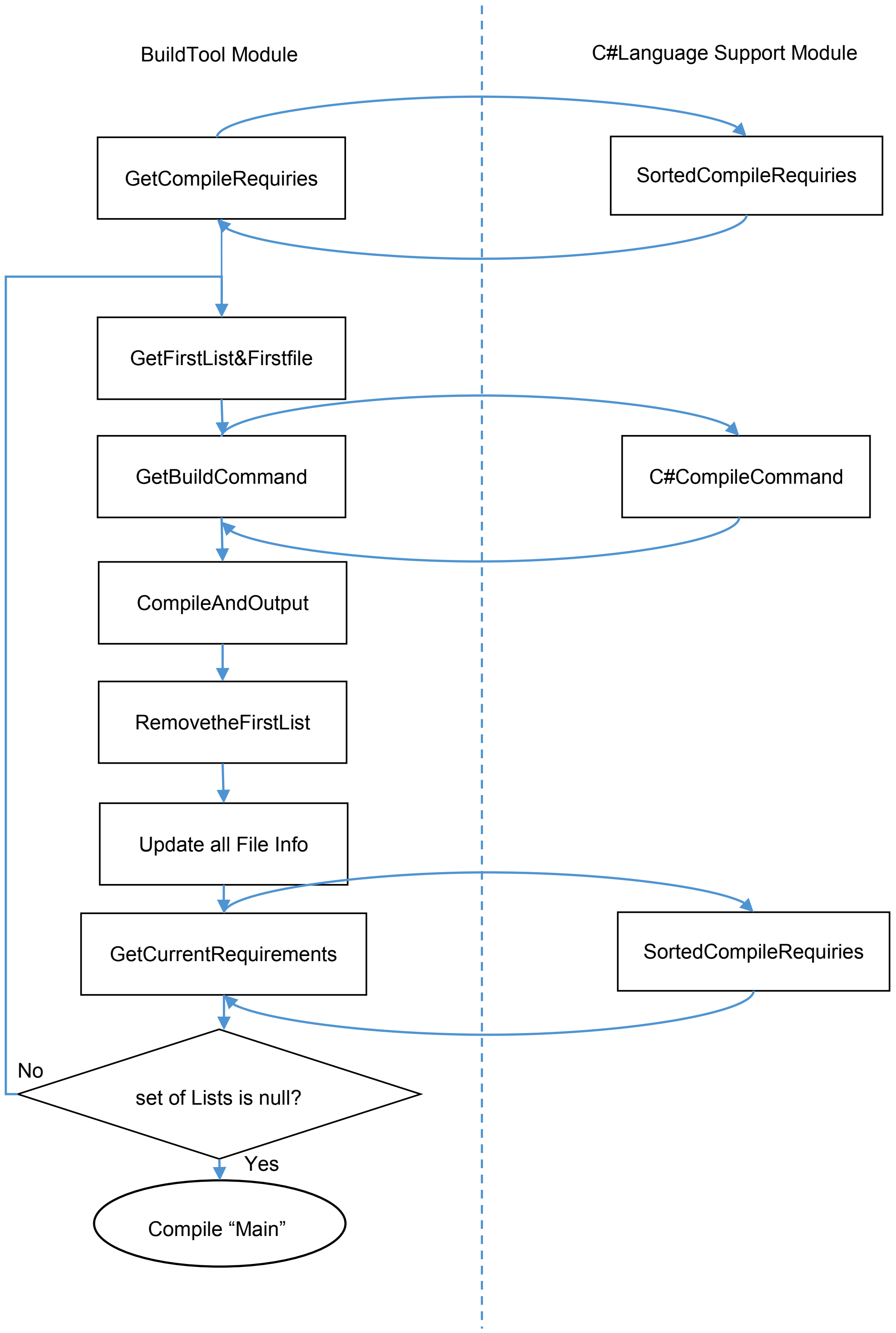
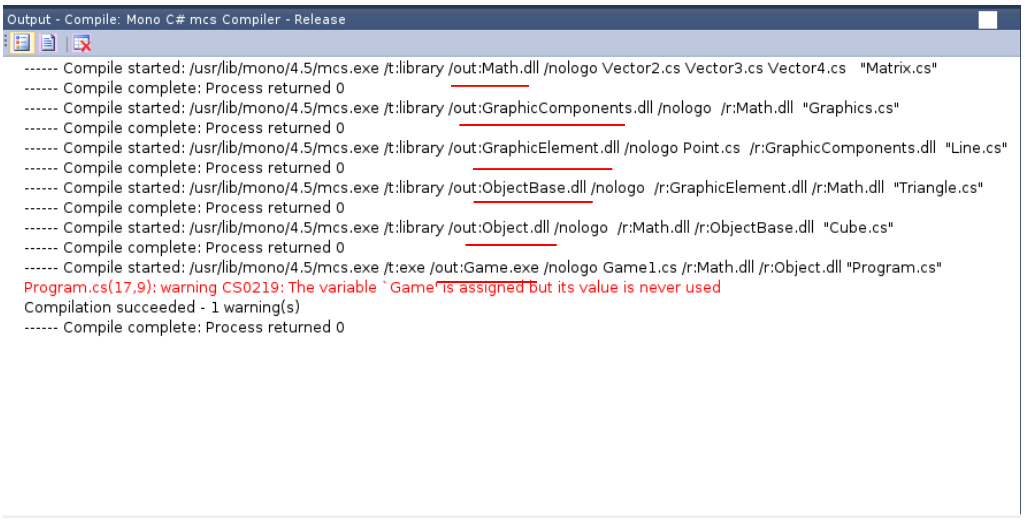


Figure 58 organized PiSharp build library process

Figure 59 shows the output statement of the entire PiSharp build command process in a correct sequence and finally generating the expected libraries and an executable file.


Output - Compile: Mono C# mcs Compiler - Release

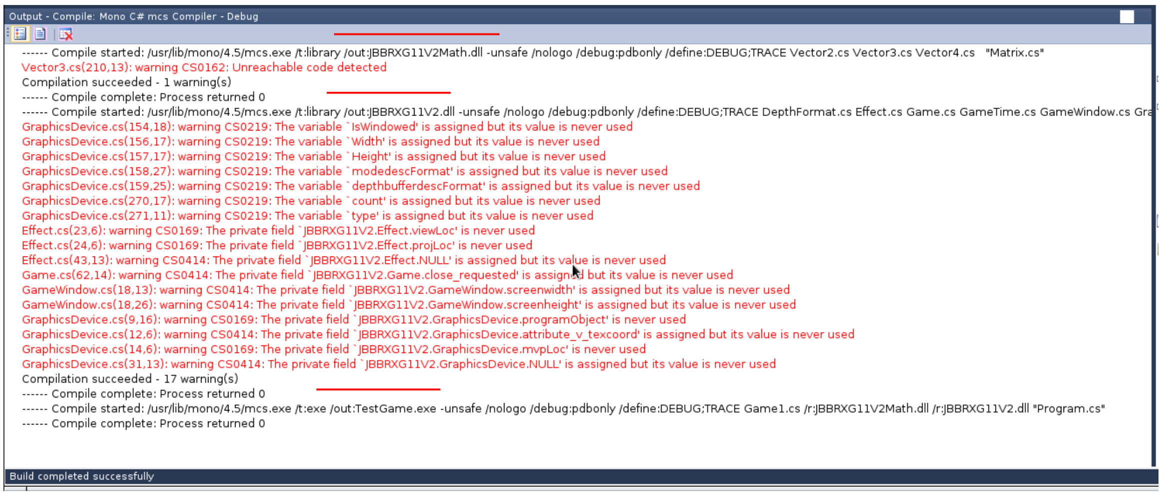
```
------ Compile started: /usr/lib/mono/4.5/mcs.exe /t:library /out:Math.dll /nologo Vector2.cs Vector3.cs Vector4.cs   "Matrix.cs"
------ Compile complete: Process returned 0
------ Compile started: /usr/lib/mono/4.5/mcs.exe /t:library /out:GraphicComponents.dll /nologo  /r:Math.dll  "Graphics.cs"
------ Compile complete: Process returned 0
------ Compile started: /usr/lib/mono/4.5/mcs.exe /t:library /out:GraphicElement.dll /nologo Point.cs  /r:GraphicComponents.dll  "Line.cs"
------ Compile complete: Process returned 0
------ Compile started: /usr/lib/mono/4.5/mcs.exe /t:library /out:ObjectBase.dll /nologo  /r:GraphicElement.dll /r:Math.dll  "Triangle.cs"
------ Compile complete: Process returned 0
------ Compile started: /usr/lib/mono/4.5/mcs.exe /t:library /out:Object.dll /nologo  /r:Math.dll /r:ObjectBase.dll  "Cube.cs"
------ Compile complete: Process returned 0
------ Compile started: /usr/lib/mono/4.5/mcs.exe /t:exe /out:Game.exe /nologo Game1.cs /r:Math.dll /r:Object.dll "Program.cs"
Program.cs(17,9): warning CS0219: The variable `Game' is assigned but its value is never used
Compilation succeeded - 1 warning(s)
------ Compile complete: Process returned 0
```

Figure 59 output window of PiSharp build process

To compile a Wang's(2014) Pi-XNA sample program[38] – "JBBR cube5 intermediate lighting texture" program – Matrix, Vector2, Vector3, Vector4 are mathematical classes under "JBBERXG11V2Math" namespace, other graphics components classes are under "JBBERXG11V2" namespace. The user's XNA-like program is under the "TestGame" namespace. As a consequence, the expected build order is:

1. Build "JBBERXG11V2Math.dll";
2. Build "JBBERXG11V2.dll";
3. Build "TestGame.exe";

The complete output statements are shown in PiSharp's Output window within Figure 60.


Output - Compile: Mono C# mcs Compiler - Debug

```
------ Compile started: /usr/lib/mono/4.5/mcs.exe /t:library /out:JBBRXG11V2Math.dll -unsafe /nologo /debug:pdbonly /define:DEBUG;TRACE Vector2.cs Vector3.cs Vector4.cs   "Matrix.cs"
Vector3.cs(210,13): warning CS0162: Unreachable code detected
Compilation succeeded - 1 warning(s)
------ Compile complete: Process returned 0
------ Compile started: /usr/lib/mono/4.5/mcs.exe /t:library /out:JBBRXG11V2.dll -unsafe /nologo /debug:pdbonly /define:DEBUG;TRACE DepthFormat.cs Effect.cs Game.cs GameTime.cs GameWindow.cs Gra
GraphicsDevice.cs(154,18): warning CS0219: The variable `IsWindowed' is assigned but its value is never used
GraphicsDevice.cs(156,17): warning CS0219: The variable `Width' is assigned but its value is never used
GraphicsDevice.cs(157,17): warning CS0219: The variable `Height' is assigned but its value is never used
GraphicsDevice.cs(158,27): warning CS0219: The variable `modedescFormat' is assigned but its value is never used
GraphicsDevice.cs(159,25): warning CS0219: The variable `depthbufferdescFormat' is assigned but its value is never used
GraphicsDevice.cs(270,17): warning CS0219: The variable `count' is assigned but its value is never used
GraphicsDevice.cs(271,11): warning CS0219: The variable `type' is assigned but its value is never used
Effect.cs(23,6): warning CS0169: The private field `JBBRXG11V2.Effect.viewLoc' is never used
Effect.cs(24,6): warning CS0169: The private field `JBBRXG11V2.Effect.projLoc' is never used
Effect.cs(43,13): warning CS0414: The private field `JBBRXG11V2.Effect.NULL' is assigned but its value is never used
Game.cs(62,14): warning CS0414: The private field `JBBRXG11V2.Game.close_requested' is assigned but its value is never used
GameWindow.cs(18,13): warning CS0414: The private field `JBBRXG11V2.GameWindow.screenwidth' is assigned but its value is never used
GameWindow.cs(18,26): warning CS0414: The private field `JBBRXG11V2.GameWindow.screenheight' is assigned but its value is never used
GraphicsDevice.cs(9,16): warning CS0169: The private field `JBBRXG11V2.GraphicsDevice.programObject' is never used
GraphicsDevice.cs(12,6): warning CS0414: The private field `JBBRXG11V2.GraphicsDevice.attribute_v_texcoord' is assigned but its value is never used
GraphicsDevice.cs(14,6): warning CS0169: The private field `JBBRXG11V2.GraphicsDevice.mvpLoc' is never used
GraphicsDevice.cs(31,13): warning CS0414: The private field `JBBRXG11V2.GraphicsDevice.NULL' is assigned but its value is never used
Compilation succeeded - 17 warning(s)
------ Compile complete: Process returned 0
------ Compile started: /usr/lib/mono/4.5/mcs.exe /t:exe /out:TestGame.exe -unsafe /nologo /debug:pdbonly /define:DEBUG;TRACE Game1.cs /r:JBBRXG11V2Math.dll /r:JBBRXG11V2.dll "Program.cs"
------ Compile complete: Process returned 0
```

Build completed successfully

Figure 60 JBBER cube5 program build process in PiSharp

A “Screenshot” application on Linux distributions cannot capture the graphic running displaying on screen. Therefore, a photograph of running “JBBR cube 5” program through PiSharp IDE is shown in Figure 61 for instead. As a result, the reorganized build process in PiSharp is capable of developing complex XNA-like programs.

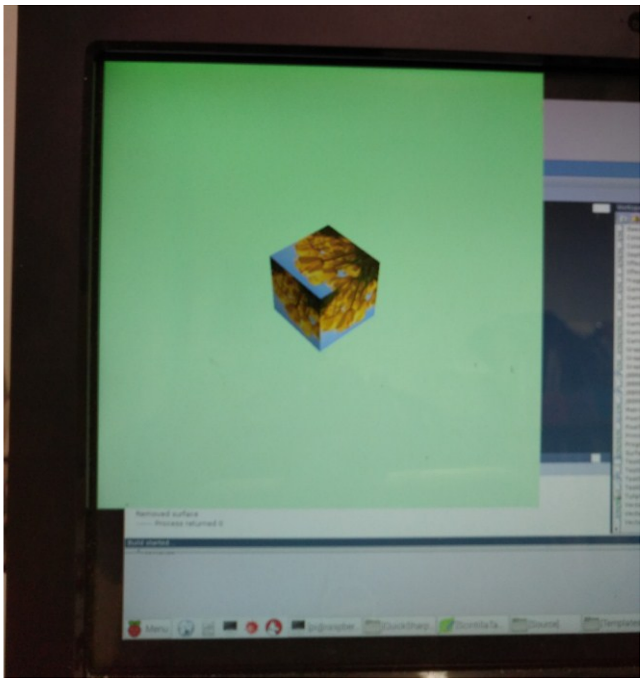
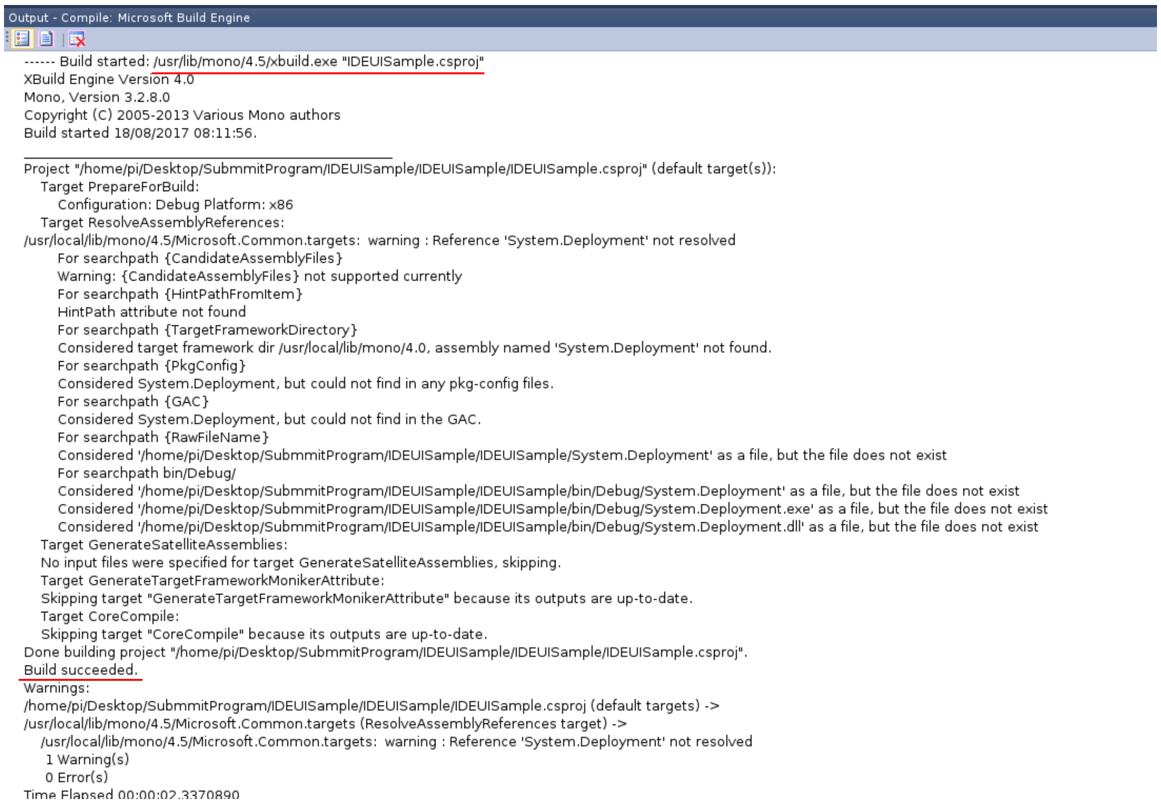


Figure 61 running Pi-XNA program via PiSharp

Wang’s programs contain a number of printing-line statements in order to track the Pi-XNA program workflow. While the Pi-XNA Program is running, a single writing-text procedure can be called tens or hundreds of times. Unlike running with a command terminal, PiSharp is based on windowing system that relies on Window Message system. When the program is displaying graphics, the IDE window stops processing messages and cannot respond. It means that no text will be shown in the output window during graphics is displaying on screen. Considering the performance of Raspberry Pi, in the Wang’s(2014) “JBBR cube5 intermediate lighting texture” program[38], which mainly used during PiSharp project development, most write text calls have been commented out to avoid excessive data forward to the output stream in a short time. The window has to frequently repaint itself, which might lead the IDE to crash.

6.2.3. Build Program from a Project File

As far as Visual Studio project is concerned, PiSharp users would prefer to develop and run sample programs that has been premade or precompiled on a Windows platform. The basic idea is to traverse every file in the current working directory and detect if a “*.csproj” project file exists. Then the build command is configured in a project language support module rather than C# module. It assigned with a “xbuild” compiler which commonly used for C# project on Mono platform. The output statement of build IDEUISample project is shown in Figure 62.



```
------ Build started: /usr/lib/mono/4.5/xbuild.exe "IDEUISample.csproj"
XBuild Engine Version 4.0
Mono, Version 3.2.8.0
Copyright (C) 2005-2013 Various Mono authors
Build started 18/08/2017 08:11:56.

__________________________________________________
Project "/home/pi/Desktop/SubmmitProgram/IDEUISample/IDEUISample/IDEUISample.csproj" (default target(s)):
  Target PrepareForBuild:
    Configuration: Debug Platform: x86
  Target ResolveAssemblyReferences:
/usr/local/lib/mono/4.5/Microsoft.Common.targets:  warning : Reference 'System.Deployment' not resolved
    For searchpath {CandidateAssemblyFiles}
    Warning: {CandidateAssemblyFiles} not supported currently
    For searchpath {HintPathFromItem}
    HintPath attribute not found
    For searchpath {TargetFrameworkDirectory}
    Considered target framework dir /usr/local/lib/mono/4.0, assembly named 'System.Deployment' not found.
    For searchpath {PkgConfig}
    Considered System.Deployment, but could not find in any pkg-config files.
    For searchpath {GAC}
    Considered System.Deployment, but could not find in the GAC.
    For searchpath {RawFileName}
    Considered '/home/pi/Desktop/SubmmitProgram/IDEUISample/IDEUISample/System.Deployment' as a file, but the file does not exist
    For searchpath bin/Debug/
    Considered '/home/pi/Desktop/SubmmitProgram/IDEUISample/IDEUISample/bin/Debug/System.Deployment' as a file, but the file does not exist
    Considered '/home/pi/Desktop/SubmmitProgram/IDEUISample/IDEUISample/bin/Debug/System.Deployment.exe' as a file, but the file does not exist
    Considered '/home/pi/Desktop/SubmmitProgram/IDEUISample/IDEUISample/bin/Debug/System.Deployment.dll' as a file, but the file does not exist
  Target GenerateSatelliteAssemblies:
  No input files were specified for target GenerateSatelliteAssemblies, skipping.
  Target GenerateTargetFrameworkMonikerAttribute:
  Skipping target "GenerateTargetFrameworkMonikerAttribute" because its outputs are up-to-date.
  Target CoreCompile:
  Skipping target "CoreCompile" because its outputs are up-to-date.
Done building project "/home/pi/Desktop/SubmmitProgram/IDEUISample/IDEUISample/IDEUISample.csproj".
Build succeeded.
Warnings:
/home/pi/Desktop/SubmmitProgram/IDEUISample/IDEUISample/IDEUISample.csproj (default targets) ->
/usr/local/lib/mono/4.5/Microsoft.Common.targets (ResolveAssemblyReferences target) ->
  /usr/local/lib/mono/4.5/Microsoft.Common.targets:  warning : Reference 'System.Deployment' not resolved
   1 Warning(s)
   0 Error(s)
Time Elapsed 00:00:02.3370890
```

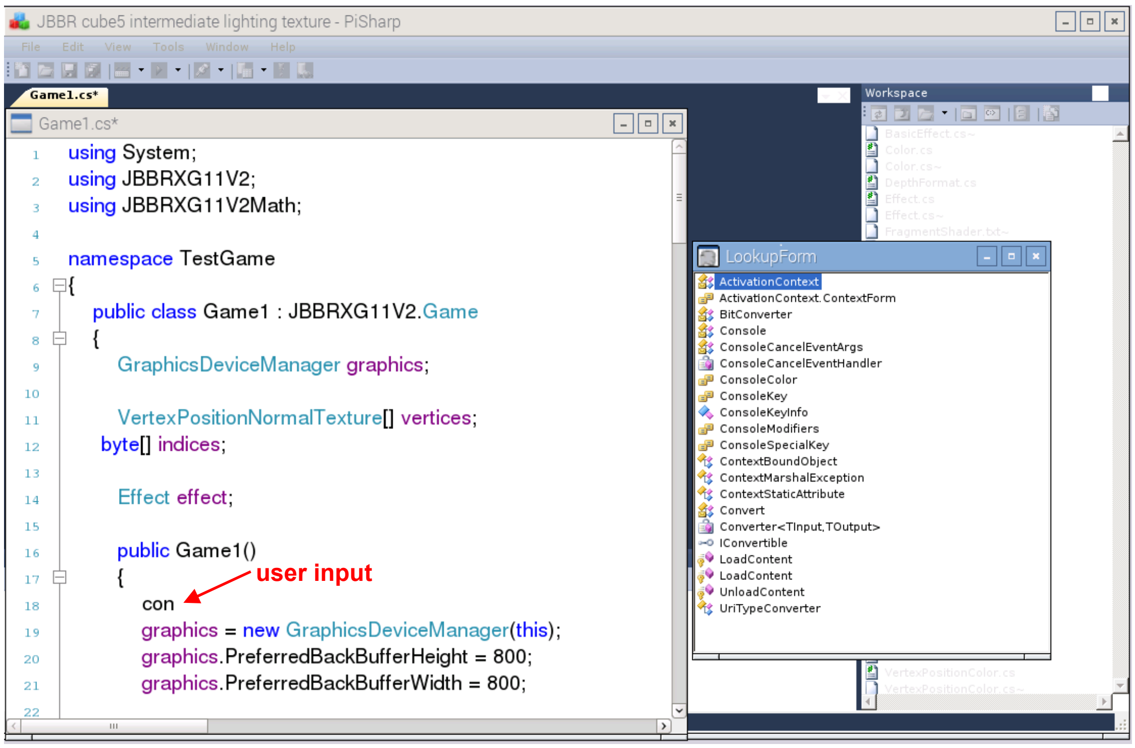
Figure 62 build project with “xbuild” compiler

Users can create a new “Run” tool for the project by using mono.exe runtime in PiSharp buildtool setting window. The output path and other common options can be customized. However, any windowing project is not recommended to run with PiSharp which conflict with the GTK-based Scintilla text editor that will lead to “not respond”.

Chapter 7. Auto-Completion(IntelliSense)

Code auto-completion brings significant advantages for developers. It makes programming efficient and reduces typing errors. Code completion is implemented in PiSharp. Managing pop up window caused a number of issues which have been discussed in Chapter 5. PiSharp project improved in filtering all loaded classes, members, namespaces, by type-in letters. For instance, a user just has type three letters: "con", then the lookup pop- up form shows possible completions. Note that it finds all matches with the letter sequence, not just matches with the letters at the start.

The following the screenshot Figure 63 shows Code Auto-completion in PiSharp system.



```
using System;
using JBBRXG11V2;
using JBBRXG11V2Math;

namespace TestGame
{
    public class Game1 : JBBRXG11V2.Game
    {
        GraphicsDeviceManager graphics;

        VertexPositionNormalTexture[] vertices;
      byte[] indices;

        Effect effect;

        public Game1()
        {
            con
            graphics = new GraphicsDeviceManager(this);
            graphics.PreferredBackBufferHeight = 800;
            graphics.PreferredBackBufferWidth = 800;
```

```
ActivationContext
ActivationContext.ContextForm
BitConverter
Console
ConsoleCancelEventArgs
ConsoleCancelEventHandler
ConsoleColor
ConsoleKey
ConsoleKeyInfo
ConsoleModifiers
ConsoleSpecialKey
ContextBoundObject
ContextMarshalException
ContextStaticAttribute
Convert
Converter<TInput,TOutput>
IConvertible
LoadContent
LoadContent
UnloadContent
UriTypeConverter
```

Figure 63 code Auto-completion in PiSharp

QuickSharp was capable of providing interactive source level debugging. The Scintilla library is capable of supporting such debugging. However, this has not yet built into PiSharp.

Chapter 8. Conclusion and Future Work

The intended use of the PiSharp project is to make an ease for beginners developing a XNA-like program on a Raspberry-Pi. Eventually, we achieve our goals. PiSharp is capable of:

1. Managing and navigating a directory of source files
2. Display a file in a code text editor
3. Display code with syntax highlight
4. Automatically discovering program library structure from code namespaces
5. Compiling libraries and programs automatically with recompilation avoided if source code has not been updated
6. Compiling and running from the IDE with reorganized build system
7. Editing more than one file at a time through GTK+ environment
8. Providing code compilation for global and local names of variables, methods and classes.

The automated build system has been tested on a 3D graphics and mathematics system and allows simple XNA-like programs in a single source file to generate animated 3D images. It requires little specialist knowledge to use and so should be suitable for novice programmers. PiSharp runs on the Raspberry Pi and can compile and run simple graphics programs in approximately 10 seconds. It brings the world of convenient automated programming to the small Raspberry Pi system.

Future work

There are two important ways which the current PiSharp system is incomplete. The first is that the editor window does not properly dock with the main IDE window. Instead, PiSharp text editors present as float windows. The system is usable but it would be helpful to fix this problem. It will be difficult to fix.

The second incompleteness is debugging. As Scintilla has supporting features this should be possible, and is the highest priority for further work.

Appendix A

Mono Installation Instructions

To run the code in this project, it is necessary to install Mono components – the open source implementation of .NET Framework for C#. At the time of writing, there was an issue with Mono on the Raspberry Pi – it had errors in handling hardware floating point. Instead of using the standard version of Mono, therefore, code from an experimental branch of the Mono project in which the floating point handling had been repaired was used.

The Method of Mono Hard-Float component installation is from Danson's (2013) blog[39]:

1. Launch a command terminal window and ensure Raspbian system is fully up to date:

```
sudo apt-get update
sudo apt-get upgrade
```

2. Install a bunch of dependencies:

```
sudo apt-get install gettext  build-essential  git-core  automake  libtool  libglib2.0-dev
```

3. Download and install the latest version of Mono from apt source

```
sudo apt-get install mono-complete
```

4. Download a GitHub newest version of Mono components

```
sudo git clone git://github.com/mono/mono.git
```

5. Once the clone is complete then move into the mono directory

```
cd mono
```

6. Switch Mono project source from latest version to 3.12.0-hotfix. The Mono 4.0.0 or later version requires Mono 3.8.0 or later version to build "mcs" compiler. However, the apt source only provides a Mono 3.2.8 version, which means the GitHub Mono 4.0.0 or later version is not able to be installed.

```
sudo git checkout mono-3.12.0-tls-hotfix
sudo git submodule init
sudo git submodule update
```

7. Run an auto-generating script to ensure the configuration of make is correct

```
sudo ./autogen.sh --prefix=/usr/local --enable-nls=no
```

8. Use “make” command to compile

```
sudo make
```

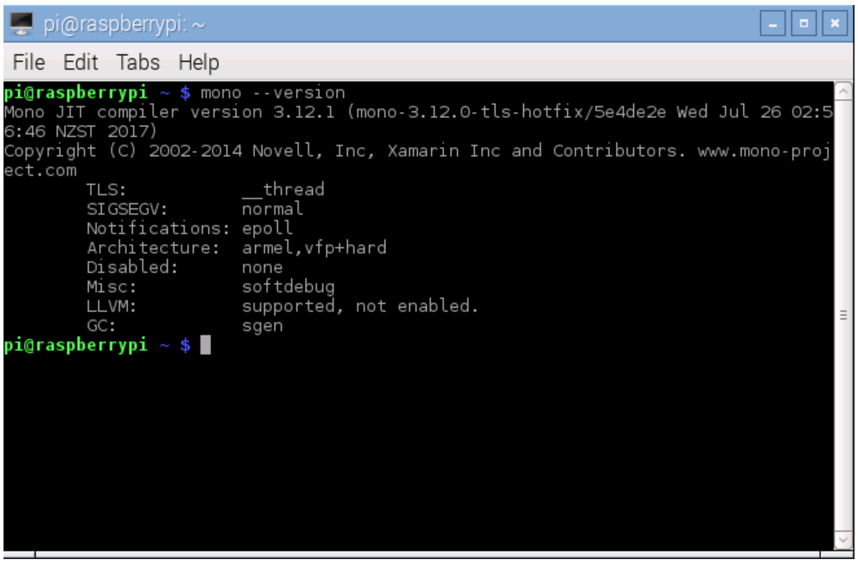
9. Ignore the errors and warnings, then “make install”

```
sudo make install
```

10. Should be able to see the current version of Mono that is running on R-Pi after installation.

```
mono --version
```

Figure 64 shows display the current Mono version runs on Raspbian.



```
pi@raspberrypi ~ $ mono --version
Mono JIT compiler version 3.12.1 (mono-3.12.0-tls-hotfix/5e4de2e Wed Jul 26 02:5
6:46 NZST 2017)
Copyright (C) 2002-2014 Novell, Inc, Xamarin Inc and Contributors. www.mono-proj
ect.com
        TLS:           __thread
        SIGSEGV:       normal
        Notifications: epoll
        Architecture:  armel,vfp+hard
        Disabled:      none
        Misc:          softdebug
        LLVM:          supported, not enabled.
        GC:            sgen
pi@raspberrypi ~ $
```

Figure 64 Mono version: 3.12.0-tls-hoxfix

For Raspberry Pi Model B or B+ with ARMv6 architecture,

commands:

```
cd ~
https://dl.dropboxusercontent.com/u/98507800/s-config/mono_2_11_4_armv6hf_binary.tgz
cd /
sudo tar zxvf ./mono_2_11_4_armv6hf_binary.tgz
sudo ldconfig
sudo apt-get install libgdiplus
```

Note: For the first time running QuickSharp IDE with mono, a root command may be required to access the given key.

Appendix B

MoMA Scan Results

	Assembly	Version	Missing	Not Implemented	Todo	P/Invoke
✓	ICSharpCode.SharpZLib.dll	0.86.0.518	0	0	0	0
✓	QuickSharp.BuildTools.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.Cassini.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.AspNet.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.CSharp.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.DotNet.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.Html.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.JScript.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.MSSCE.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.MSSql.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.MySQL.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.ObjectBrowser.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.Sql.dll	2.0.0.26942	0	0	0	0
✓	QuickSharp.CodeAssist.SQLite.dll	2.0.0.26942	0	0	0	0

⊟	❗	QuickSharp.Core.dll	2.0.0.26942	0	0	3	3

Calling Method	Method with [MonoTodo]	Reason
ToolStripMenuItem GetMenuItemByName (string)	ToolStripItem[] ToolStripItemCollection.Find (string, bool)	searchAllChildren parameter isn't used
void LoadSingleToolStrip ()	void ToolStripPanel.Join (ToolStrip)	Not implemented
void LoadMultipleToolStrips ()	void ToolStripPanel.Join (ToolStrip, int)	Not implemented

Calling Method	P/Invoke Method	P/Invoke Library
void SetScheme ()	int DisplayInformation.GetCurrentThemeName (StringBuilder, int, StringBuilder, int, StringBuilder, int)	uxtheme.dll
void SetScheme ()	int VS2008ColorTable/DisplayInformation.GetCurrentThemeName (StringBuilder, int, StringBuilder, int, StringBuilder, int)	uxtheme.dll
void DeleteWithUndo (string)	int FileTools.SHFileOperation (FileTools/SHFILEOPSTRUCT&)	shell32.dll

	✓	QuickSharp.DocumentTemplates.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Editor.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Explorer.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.FindInFiles.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.AspNet.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.CSharp.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.Dbml.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.Html.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.JScript.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.Mono.dll	2.0.0.26942	0	0	0	0

	✓	QuickSharp.Language.MSIL.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.Proj.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.Resx.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.VBNet.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.Wsdl.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.Xml.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Language.Xsd.dll	2.0.0.26942	0	0	0	0
⊟	!	QuickSharp.Output.dll	2.0.0.26942	0	0	1	0

Calling Method	Method with [MonoTodo]	Reason
void .ctor (string)	void Control.set_CheckForIllegalCrossThreadCalls (bool)	Stub, value is not used

	✓	QuickSharp.Persistence.SQLite.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.SDKTools.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.SqlEditor.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.SQLiteManager.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.SqlManager.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.TextEditor.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Tools.dll	2.0.0.26942	0	0	0	0
	✓	QuickSharp.Workspace.dll	2.0.0.26942	0	0	0	0
⊟	!	ScintillaNet.dll	2.0.5372.27148	0	0	1	7

Calling Method	Method with [MonoTodo]	Reason
bool Print (bool)	void PrintDialog.set_UseEXDialog (bool)	Stub, not implemented, will always use default dialog

Calling Method	P/Invoke Method	P/Invoke Library
void WndProc (Message&)	IntPtr NativeMethods.SetParent (IntPtr, IntPtr)	user32.dll
void WndProc (Message&)	bool NativeMethods.GetUpdateRect (IntPtr, RECT&, bool)	user32.dll
CreateParams get_CreateParams ()	IntPtr NativeMethods.LoadLibrary (string)	kernel32
void set_AllowDrop (bool)	void NativeMethods.DragAcceptFiles (IntPtr, bool)	shell32.dll
void handleFileDrop (IntPtr)	int NativeMethods.DragQueryFileA (IntPtr, uint, IntPtr, int)	shell32.dll
void handleFileDrop (IntPtr)	int NativeMethods.DragQueryFileA (IntPtr, uint, IntPtr, int)	shell32.dll
void handleFileDrop (IntPtr)	int NativeMethods.DragFinish (IntPtr)	shell32.dll

⊟	●	System.Data.SQLite.dll	1.0.66.0	0	8	3	81

Calling Method	Method that Throws NotImplementedException
void Initialize (string)	void DbConnectionStringBuilder.GetProperties (Hashtable)
DataTable GetSchema ()	DataTable DbConnection.GetSchema (string, String[])
DataTable GetSchema (string)	DataTable DbConnection.GetSchema (string, String[])
DataTable GetSchemaTable (bool, bool)	DataTable DbConnection.GetSchema (string, String[])
DataTable GetSchemaTable (bool, bool)	DataTable DbConnection.GetSchema (string, String[])
void .ctor (SQLiteDataReader, SQLiteStatement)	DataTable DbConnection.GetSchema (string, String[])
void .ctor (SQLiteDataReader, SQLiteStatement)	DataTable DbConnection.GetSchema (string, String[])
void .ctor (SQLiteDataReader, SQLiteStatement)	DataTable DbConnection.GetSchema (string, String[])

Calling Method	Method with [MonoTodo]	Reason
void Initialize (string)	void DbConnectionStringBuilder.GetProperties (Hashtable)	

void Prepare (PreparingEnlistment)	void PreparingEnlistment.Prepared ()	
void .ctor (Transaction)	Enlistment Transaction.EnlistVolatile (IEnlistmentNotification, EnlistmentOptions)	EnlistmentOptions being ignored

Calling Method	P/Invoke Method	P/Invoke Library
string SQLiteLastError (SQLiteConnectionHandle)	IntPtr UnsafeNativeMethods.sqlite3_errmsg_interop (IntPtr, Int32&)	System.Data.SQLite.DLL
void FinalizeStatement (SQLiteStatementHandle)	int UnsafeNativeMethods.sqlite3_finalize_interop (IntPtr)	System.Data.SQLite.DLL
void CloseConnection (SQLiteConnectionHandle)	int UnsafeNativeMethods.sqlite3_close_interop (IntPtr)	System.Data.SQLite.DLL
void ResetConnection (SQLiteConnectionHandle)	IntPtr UnsafeNativeMethods.sqlite3_next_stmt (IntPtr, IntPtr)	System.Data.SQLite.DLL
void ResetConnection (SQLiteConnectionHandle)	int UnsafeNativeMethods.sqlite3_reset_interop (IntPtr)	System.Data.SQLite.DLL
void ResetConnection (SQLiteConnectionHandle)	int UnsafeNativeMethods.sqlite3_exec (IntPtr, Byte[], IntPtr, IntPtr, IntPtr&)	System.Data.SQLite.DLL
bool IsAutocommit (SQLiteConnectionHandle)	int UnsafeNativeMethods.sqlite3_get_autocommit (IntPtr)	System.Data.SQLite.DLL
void Cancel ()	void UnsafeNativeMethods.sqlite3_interrupt (IntPtr)	System.Data.SQLite.DLL
string get_SQLiteVersion ()	IntPtr UnsafeNativeMethods.sqlite3_libversion ()	System.Data.SQLite.DLL
int get_Changes ()	int UnsafeNativeMethods.sqlite3_changes (IntPtr)	System.Data.SQLite.DLL
void Open (string, SQLiteOpenFlagsEnum, int, bool)	int UnsafeNativeMethods.sqlite3_open_interop (Byte[], int, IntPtr&)	System.Data.SQLite.DLL
void SetTimeout (int)	int UnsafeNativeMethods.sqlite3_busy_timeout (IntPtr, int)	System.Data.SQLite.DLL
bool Step (SQLiteStatement)	int UnsafeNativeMethods.sqlite3_step (IntPtr)	System.Data.SQLite.DLL
int Reset (SQLiteStatement)	int UnsafeNativeMethods.sqlite3_reset_interop (IntPtr)	System.Data.SQLite.DLL
SQLiteStatement Prepare (DbCommandBuilder), string, SQLiteStatement, uint, String&)	int UnsafeNativeMethods.sqlite3_prepare_interop (IntPtr, IntPtr, int, IntPtr&, IntPtr&, Int32&)	System.Data.SQLite.DLL
void Bind_Double (SQLiteStatement, int, double)	int UnsafeNativeMethods.sqlite3_bind_double (IntPtr, int, double)	System.Data.SQLite.DLL
void Bind_Int32 (SQLiteStatement, int, int)	int UnsafeNativeMethods.sqlite3_bind_int (IntPtr, int, int)	System.Data.SQLite.DLL

void Bind_Int64 (SQLiteStatement, int, Int64)	int UnsafeNativeMethods.sqlite3_bind_int64 (IntPtr, int, Int64)	System.Data.SQLite.DLL
void Bind_Text (SQLiteStatement, int, string)	int UnsafeNativeMethods.sqlite3_bind_text (IntPtr, int, Byte[], int, IntPtr)	System.Data.SQLite.DLL
void Bind_DateTime (SQLiteStatement, int, DateTime)	int UnsafeNativeMethods.sqlite3_bind_text (IntPtr, int, Byte[], int, IntPtr)	System.Data.SQLite.DLL
void Bind_Blob (SQLiteStatement, int, Byte[])	int UnsafeNativeMethods.sqlite3_bind_blob (IntPtr, int, Byte[], int, IntPtr)	System.Data.SQLite.DLL
void Bind_Null (SQLiteStatement, int)	int UnsafeNativeMethods.sqlite3_bind_null (IntPtr, int)	System.Data.SQLite.DLL
int Bind_ParamCount (SQLiteStatement)	int UnsafeNativeMethods.sqlite3_bind_parameter_count (IntPtr)	System.Data.SQLite.DLL
string Bind_ParamName (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_bind_parameter_name_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
int Bind_ParamIndex (SQLiteStatement, string)	int UnsafeNativeMethods.sqlite3_bind_parameter_index (IntPtr, Byte[])	System.Data.SQLite.DLL
int ColumnCount (SQLiteStatement)	int UnsafeNativeMethods.sqlite3_column_count (IntPtr)	System.Data.SQLite.DLL
string ColumnName (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_name_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
TypeAffinity ColumnAffinity (SQLiteStatement, int)	TypeAffinity UnsafeNativeMethods.sqlite3_column_type (IntPtr, int)	System.Data.SQLite.DLL
string ColumnType (SQLiteStatement, int, TypeAffinity&)	IntPtr UnsafeNativeMethods.sqlite3_column_decltype_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
string ColumnOriginalName (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_origin_name_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
string ColumnDatabaseName (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_database_name_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
string ColumnTableName (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_table_name_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
void ColumnMetaData (string, string, string, String&, String&, Boolean&, Boolean&, Boolean&)	int UnsafeNativeMethods.sqlite3_table_column_metadata_interop (IntPtr, Byte[], Byte[], Byte[], IntPtr&, IntPtr&, Int32&, Int32&, Int32&, Int32&, Int32&)	System.Data.SQLite.DLL

double GetDouble (SQLiteStatement, int)	double UnsafeNativeMethods.sqlite3_column_double (IntPtr, int)	System.Data.SQLite.DLL
int GetInt32 (SQLiteStatement, int)	int UnsafeNativeMethods.sqlite3_column_int (IntPtr, int)	System.Data.SQLite.DLL
Int64 GetInt64 (SQLiteStatement, int)	Int64 UnsafeNativeMethods.sqlite3_column_int64 (IntPtr, int)	System.Data.SQLite.DLL
string GetText (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_text_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
DateTime GetDateTime (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_text_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
Int64 GetBytes (SQLiteStatement, int, int, Byte[], int, int)	int UnsafeNativeMethods.sqlite3_column_bytes (IntPtr, int)	System.Data.SQLite.DLL
Int64 GetBytes (SQLiteStatement, int, int, Byte[], int, int)	IntPtr UnsafeNativeMethods.sqlite3_column_blob (IntPtr, int)	System.Data.SQLite.DLL
int AggregateCount (IntPtr)	int UnsafeNativeMethods.sqlite3_aggregate_count (IntPtr)	System.Data.SQLite.DLL
void CreateFunction (string, int, bool, SQLiteCallback, SQLiteCallback, SQLiteFinalCallback)	int UnsafeNativeMethods.sqlite3_create_function_interop (IntPtr, Byte[], int, int, IntPtr, SQLiteCallback, SQLiteCallback, SQLiteFinalCallback, int)	System.Data.SQLite.DLL
void CreateFunction (string, int, bool, SQLiteCallback, SQLiteCallback, SQLiteFinalCallback)	int UnsafeNativeMethods.sqlite3_create_function_interop (IntPtr, Byte[], int, int, IntPtr, SQLiteCallback, SQLiteCallback, SQLiteFinalCallback, int)	System.Data.SQLite.DLL
void CreateCollation (string, SQLiteCollation, SQLiteCollation)	int UnsafeNativeMethods.sqlite3_create_collation (IntPtr, Byte[], int, IntPtr, SQLiteCollation)	System.Data.SQLite.DLL
void CreateCollation (string, SQLiteCollation, SQLiteCollation)	int UnsafeNativeMethods.sqlite3_create_collation (IntPtr, Byte[], int, IntPtr, SQLiteCollation)	System.Data.SQLite.DLL
int ContextCollateCompare (CollationEncodingEnum, IntPtr, string, string)	int UnsafeNativeMethods.sqlite3_context_collcompare (IntPtr, Byte[], int, Byte[], int)	System.Data.SQLite.DLL
int ContextCollateCompare (CollationEncodingEnum, IntPtr, Char[], Char[])	int UnsafeNativeMethods.sqlite3_context_collcompare (IntPtr, Byte[], int, Byte[], int)	System.Data.SQLite.DLL
CollationSequence GetCollationSequence	IntPtr UnsafeNativeMethods.sqlite3_context_collseq (IntPtr, Int32&, Int32&, Int32&)	System.Data.SQLite.DLL

(SQLiteFunction, IntPtr)		
Int64 GetParamValueBytes (IntPtr, int, Byte[], int, int)	int UnsafeNativeMethods.sqlite3_value_bytes (IntPtr)	System.Data.SQLite.DLL
Int64 GetParamValueBytes (IntPtr, int, Byte[], int, int)	IntPtr UnsafeNativeMethods.sqlite3_value_blob (IntPtr)	System.Data.SQLite.DLL
double GetParamValueDouble (IntPtr)	double UnsafeNativeMethods.sqlite3_value_double (IntPtr)	System.Data.SQLite.DLL
int GetParamValueInt32 (IntPtr)	int UnsafeNativeMethods.sqlite3_value_int (IntPtr)	System.Data.SQLite.DLL
Int64 GetParamValueInt64 (IntPtr)	Int64 UnsafeNativeMethods.sqlite3_value_int64 (IntPtr)	System.Data.SQLite.DLL
string GetParamValueText (IntPtr)	IntPtr UnsafeNativeMethods.sqlite3_value_text_interop (IntPtr, Int32&)	System.Data.SQLite.DLL
TypeAffinity GetParamValueType (IntPtr)	TypeAffinity UnsafeNativeMethods.sqlite3_value_type (IntPtr)	System.Data.SQLite.DLL
void ReturnBlob (IntPtr, Byte[])	void UnsafeNativeMethods.sqlite3_result_blob (IntPtr, Byte[], int, IntPtr)	System.Data.SQLite.DLL
void ReturnDouble (IntPtr, double)	void UnsafeNativeMethods.sqlite3_result_double (IntPtr, double)	System.Data.SQLite.DLL
void ReturnError (IntPtr, string)	void UnsafeNativeMethods.sqlite3_result_error (IntPtr, Byte[], int)	System.Data.SQLite.DLL
void ReturnInt32 (IntPtr, int)	void UnsafeNativeMethods.sqlite3_result_int (IntPtr, int)	System.Data.SQLite.DLL
void ReturnInt64 (IntPtr, Int64)	void UnsafeNativeMethods.sqlite3_result_int64 (IntPtr, Int64)	System.Data.SQLite.DLL
void ReturnNull (IntPtr)	void UnsafeNativeMethods.sqlite3_result_null (IntPtr)	System.Data.SQLite.DLL
void ReturnText (IntPtr, string)	void UnsafeNativeMethods.sqlite3_result_text (IntPtr, Byte[], int, IntPtr)	System.Data.SQLite.DLL
IntPtr AggregateContext (IntPtr)	IntPtr UnsafeNativeMethods.sqlite3_aggregate_context (IntPtr, int)	System.Data.SQLite.DLL
void SetPassword (Byte[])	int UnsafeNativeMethods.sqlite3_key (IntPtr, Byte[], int)	System.Data.SQLite.DLL
void ChangePassword (Byte[])	int UnsafeNativeMethods.sqlite3_rekey (IntPtr, Byte[], int)	System.Data.SQLite.DLL
void SetUpdateHook (SQLiteUpdateCallback)	IntPtr UnsafeNativeMethods.sqlite3_update_hook (IntPtr, SQLiteUpdateCallback, IntPtr)	System.Data.SQLite.DLL
void SetCommitHook (SQLiteCommitCallback)	IntPtr UnsafeNativeMethods.sqlite3_commit_hook (IntPtr, SQLiteCommitCallback, IntPtr)	System.Data.SQLite.DLL
void SetRollbackHook (SQLiteRollbackCallback)	IntPtr UnsafeNativeMethods.sqlite3_rollback_hook (IntPtr, SQLiteRollbackCallback, IntPtr)	System.Data.SQLite.DLL

int GetCursorForTable (SQLiteStatement, int, int)	int UnsafeNativeMethods.sqlite3_table_cursor (IntPtr, int, int)	System.Data.SQLite.DLL
Int64 GetRowIdForCursor (SQLiteStatement, int)	int UnsafeNativeMethods.sqlite3_cursor_rowid (IntPtr, int, Int64&)	System.Data.SQLite.DLL
void GetIndexColumnExtendedInfo (string, string, string, Int32&, Int32&, String&)	int UnsafeNativeMethods.sqlite3_index_column_info_interop (IntPtr, Byte[], Byte[], Byte[], Int32&, Int32&, IntPtr&, Int32&)	System.Data.SQLite.DLL
void Open (string, SQLiteOpenFlagsEnum, int, bool)	int UnsafeNativeMethods.sqlite3_open16_interop (Byte[], int, IntPtr&)	System.Data.SQLite.DLL
void Bind_Text (SQLiteStatement, int, string)	int UnsafeNativeMethods.sqlite3_bind_text16 (IntPtr, int, string, int, IntPtr)	System.Data.SQLite.DLL
string ColumnName (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_name16_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
string GetText (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_text16_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
string ColumnOriginalName (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_origin_name16_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
string ColumnDatabaseName (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_database_name16_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
string ColumnTableName (SQLiteStatement, int)	IntPtr UnsafeNativeMethods.sqlite3_column_table_name16_interop (IntPtr, int, Int32&)	System.Data.SQLite.DLL
string GetParamValueText (IntPtr)	IntPtr UnsafeNativeMethods.sqlite3_value_text16_interop (IntPtr, Int32&)	System.Data.SQLite.DLL
void ReturnError (IntPtr, string)	void UnsafeNativeMethods.sqlite3_result_error16 (IntPtr, string, int)	System.Data.SQLite.DLL
void ReturnText (IntPtr, string)	void UnsafeNativeMethods.sqlite3_result_text16 (IntPtr, string, int, IntPtr)	System.Data.SQLite.DLL

⊟	●	WeifenLuo.WinFormsUI.Docking.dll	2.3.1.28039	0	0	0	23

Calling Method	P/Invoke Method	P/Invoke Library
Control ControlAtPoint (Point)	IntPtr NativeMethods.WindowFromPoint (Point)	user32.dll
void set_DockPanel (DockPanel)	int NativeMethods.SetWindowPos (IntPtr, IntPtr, int, int, int, int, FlagsSetWindowPos)	User32.dll

void set_BorderStyle (BorderStyle)	int NativeMethods.GetWindowLong (IntPtr, int)	user32.dll
void set_BorderStyle (BorderStyle)	int NativeMethods.GetWindowLong (IntPtr, int)	user32.dll
void set_BorderStyle (BorderStyle)	int NativeMethods.SetWindowLong (IntPtr, int, int)	user32.dll
void set_BorderStyle (BorderStyle)	int NativeMethods.SetWindowLong (IntPtr, int, int)	user32.dll
void WndProc (Message&)	int NativeMethods.ShowScrollBar (IntPtr, int, int)	user32.dll
void UpdateStyles ()	int NativeMethods.SetWindowPos (IntPtr, IntPtr, int, int, int, int, FlagsSetWindowPos)	User32.dll
bool BeginDrag ()	bool NativeMethods.DragDetect (IntPtr, Point)	User32.dll
void Show (bool)	int NativeMethods.ShowWindow (IntPtr, Int16)	User32.dll
void Activate (IDockContent)	IntPtr NativeMethods.SetFocus (IntPtr)	User32.dll
void Activate (IDockContent)	IntPtr NativeMethods.SetFocus (IntPtr)	User32.dll
void SetActivePane ()	IntPtr NativeMethods.GetFocus ()	User32.dll
IntPtr CoreHookProc (int, IntPtr, IntPtr)	IntPtr NativeMethods.CallNextHookEx (IntPtr, int, IntPtr, IntPtr)	user32.dll
IntPtr CoreHookProc (int, IntPtr, IntPtr)	IntPtr NativeMethods.CallNextHookEx (IntPtr, int, IntPtr, IntPtr)	user32.dll
void Install ()	int NativeMethods.GetCurrentThreadId ()	Kernel32.dll
void Install ()	IntPtr NativeMethods.SetWindowsHookEx (HookType, NativeMethods/HookProc, IntPtr, int)	user32.dll
void Uninstall ()	int NativeMethods.UnhookWindowsHookEx (IntPtr)	user32.dll
void WndProc (Message&)	uint NativeMethods.SendMessage (IntPtr, int, uint, uint)	User32.dll
void WndProc (Message&)	uint NativeMethods.SendMessage (IntPtr, int, uint, uint)	User32.dll
void WndProc (Message&)	uint NativeMethods.SendMessage (IntPtr, int, uint, uint)	User32.dll
void TestDrop (IDockDragSource, DockOutlineBase)	uint NativeMethods.SendMessage (IntPtr, int, uint, uint)	User32.dll
void CheckFloatWindowDispose ()	bool NativeMethods.PostMessage (IntPtr, int, uint, uint)	User32.dll

✓	QuickSharp.exe	2.0.0.26942	0	0	0	0
✓	QuickSharp.vshost.exe	10.0.0.0	0	0	0	0
	Totals		**0**	**8**	**8**	**114**

Appendix C

Table for complete Messages Mapping

Windows Messages	Scintilla Messages
EM_CANPASTE	SCI_CANPASTE
EM_CANUNDO	SCI_CANUNDO
EM_EMPTYUNDOBUFFER	SCI_EMPTYUNDOBUFFER
EM_FINDTEXTEX	SCI_FINDTEXT
EM_FORMATRANGE	SCI_FORMATRANGE
EM_GETFIRSTVISIBLELINE	SCI_GETFIRSTVISIBLELINE
EM_GETLINECOUNT	SCI_GETLINECOUNT
EM_GETSELTEXT	SCI_GETSELTEXT
EM_GETTEXTRANGE	SCI_GETTEXTRANGE
EM_HIDESELECTION	SCI_HIDESELECTION
EM_LINEINDEX	SCI_ POSITIONFROMLINE
EM_LINESCROLL	SCI_LINESCROLL
EM_REPLACESEL	SCI_REPLACESEL
EM_SCROLLCARET	SCI_SCROLLCARET
SCI_SETREADONLY	EM_SETREADONLY
WM_CLEAR	SCI_CLEAR
WM_COPY	SCI_COPY
WM_CUT	SCI_CUT
WM_GETTEXT	SCI_GETTEXT
WM_SETTEXT	SCI_SETTEXT
WM_GETTEXTLENGTH	SCI_GETTEXTLENGTH
WM_PASTE	SCI_PASTE
WM_UNDO	SCI_UNDO

Appendix D

Table for full generic macros supported by QuickSharp

Macro	Description
${SRC_PATH}	Source file path
${SRC_FILE}	Source file name
${SRC_NAME}	Source file name without extension
${SRC_EXT}	Source file extension
${OUT_PATH}	Target output file path
${OUT_FILE}	Target output file name
${OUT_NAME}	Target output file name without extension
${OUT_EXT}	Target output file extension
${IDE_HOME}	QuickSharp installation directory
${USR_HOME}	QuickSharp user data directory
${USR_DOCS}	User's "My Documents" folder
${SYSTEM}	Windows system directory (usually C:\WINDOWS\system32)
${PFILES}	Windows "Program Files" directory
${WORKSPACE}	Workspace directory name
${COMMON_OPT}	Option flags while building
${EMBEDDED_OPT}	The QuickSharp’s native features to manage build process embedded in source code
${RUNTIME_OPT}	Runtime arguments passed to a program and embedded in the source

References

1. Raspberry Pi Foundation. *What is a Raspberry Pi*. Retrieved from http://www.raspberrypi.org/help/what-is-a-raspberry-pi/
2. Raspberry Pi Foundation. *Raspberry Pi – 2006 Edition*. Retrieved from http://www.raspberrypi.org/raspberry-pi-2006-edition/
3. Rasp.TV. (2017. February 28). *New Raspberry Pi Family Photo 28 Feb 2017*. Retrieved from http://raspi.tv/2017/new-raspberry-pi-family-photo-28-feb-2017
4. Raspberry Pi Foundation. *RASPBERRY PI 2 MODEL B*. Retrieved from https://www.raspberrypi.org/products/raspberry-pi-2-model-b/
5. Raspberry Pi Foundation. *FAQS: Performance and Cost Consideration*. Retrieved from https://www.raspberrypi.org/help/faqs/#topCost
6. Wikipedia Editor. *Raspberry Pi*. Retrieved from http://en.wikipedia.org/wiki/Raspberry_Pi
7. Raspberry Pi Foundation. *Downloads*. Retrieved from https://www.raspberrypi.org/downloads/
8. Raspberry Pi Foundation. *FAQS: WILL IT RUN WINE (OR WINDOWS, OR OTHER X86 SOFTWARE)*. Retrieved from https://www.raspberrypi.org/help/faqs/#softwareX86
9. UDOO. *UDOO QUAD/DUAL*. Retrieved from https://www.udoo.org/udoo-dual-and-quad/
10. Wikipedia Editor. *UDOO*. Retrieved from https://en.wikipedia.org/wiki/UDOO
11. eLinux wiki. *UDOO*. Retrieved from http://elinux.org/UDOO
12. SolidRun. *HummingBoard SolidRun.* Retrieved from https://www.solid-run.com/freescale-imx6-family/hummingboard/
13. Beagleboard. *Beagleboard Black.* Retrieved from https://beagleboard.org/black
14. Promit's Ventspace. *DirectX/XNA Phase Out Continues*. Retrieved from https://ventspace.wordpress.com/2013/01/30/directxxna-phase-out-continues/
15. Wang, L. (2014). *XNA-like 3D Graphics Programming on the Raspberry Pi* (Master's thesis. University of Waikato, Hamilton, New Zealand). Retrieved from

http://researchcommons.waikato.ac.nz/handle/10289/8802

16. Boud, J. (2012). *Extending SlimDXna to Use XNA 4 and DirectX 11* (Dissertation, University of Waikato, Hamilton, New Zealand)
17. Code MSDN Microsoft. *Visual Basic XNA.* Retrieved from https://code.msdn.microsoft.com/windowsapps/Visual-Basic-XNA-29cd4963
18. Wikipedia Editor. *Microsoft_XNA.* Retrieved from https://en.wikipedia.org/wiki/Microsoft_XNA
19. CodePlex. *JBBRXG11*. Retrieved from https://jbbrxg11.codeplex.com/
20. Raspberry Pi Foundation. *Libraries Codecs Oss.* Retrieved from https://www.raspberrypi.org/blog/libraries-codecs-oss/
21. Khronos Group. *OpenGL ES 2_X Overview* Retrieved from https://www.khronos.org/opengles/2_X/
22. Long, S. *Introducing PIXEL*. Retrieved from https://www.raspberrypi.org/blog/introducing-pixel/
23. Wikipedia Editor. *X Window System.* Retrieved from https://en.wikipedia.org/wiki/X_Window_System
24. QuickSharp. *QuickSharp*. Retrieved from http://quicksharp.sourceforge.net/
25. Standard ECMA-335. *Common Language Infrastructure (CLI).* Retrieved from http://www.ecma-international.org/publications/standards/Ecma-335.htm
26. Scintilla. *Scintilla Documentation*. Retrieved from http://www.scintilla.org/
27. DockPanel Suite. *DockPanel Suite Documentation*. Retrieved from http://docs.dockpanelsuite.com/en/latest/
28. Microsoft. *Microsoft takes .NET open source and cross-platform, adds new development capabilities with Visual Studio 2015, .NET 2015 and Visual Studio Online*. Retrieved from https://news.microsoft.com/2014/11/12/microsoft-takes-net-open-source-and-cross-platform-adds-new-development-capabilities-with-visual-studio-2015-net-2015-and-visual-studio-online/
29. Mono. *Mono Documentation*. Retrieved from http://www.mono-project.com/docs/about-mono/

30. Mono. *Stetic GUI Designer*. Retrieved from http://www.monodevelop.com/documentation/stetic-gui-designer/
31. Pobst, J. *Porting Winforms Applications*. Retrieved from http://www.mono-project.com/docs/gui/winforms/porting-winforms-applications/
32. GNU. *Code-Gen-Options*. Retrieved from https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html
33. Man7. Linux Programmer's Manual. Retrieved from: http://man7.org/linux/man-pages/man3/dlopen.3.html
34. Microsoft MSDN. Message Structure. Retrieved from https://msdn.microsoft.com/enus/library/system.windows.forms.message(v=vs.110).aspx
35. Mayank. *The Linux filesystem explained*. Retrieved from http://freeos.com/articles/3102
36. Github. *SharpZipLib*. Retrieved from http://icsharpcode.github.io/SharpZipLib/
37. Mono. *Guide: Fixing issues MoMA finds.* Retrieved from http://www.mono-project.com/archived/moma/issue-descriptions/
38. Wang. L. (2014). *ExamplePrograms* (Test programs of Master's thesis. University of Waikato, Hamilton, New Zealand). Retrieved from http://researchcommons.waikato.ac.nz/handle/10289/8802
39. Danson, N. (2013). *Building Mono on a Raspberry Pi(Hard Float).* Retrieved from https://neildanson.wordpress.com/2013/12/10/building-mono-on-a-raspberry-pi-hard-float/