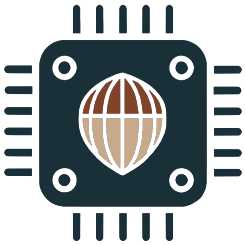
Technische Hochschule Augsburg



Forschungsgruppe EES - PicoNut Projekt
Fakultät Informatik

Technische Hochschule Augsburg

Implementierung einer Grafikausgabe mit GUI für den PicoNut-Simulator

Konrad Armbrecht
konrad.armbrecht@tha.de
Matrikelnummer: 2220037

Betreuer:
Prof. Dr.-Ing. Gundolf Kiefer
gundolf.kiefer@tha.de

Inhaltsverzeichnis

1. Einleitung

In dieser Projektarbeit werden der Hintergrund und die praktische Durchführung der Implementierung einer grafischen Ausgabeschnittstelle mit zugehöriger grafischer Benutzeroberfläche (GUI) für den PicoNut-Simulator beschrieben. Ziel ist es, den bislang textbasierten Simulator um eine visuelle Komponente zu erweitern, die komplexere Formen der Ausgabe und Interaktion ermöglicht.

Hinweis: Zur besseren Lesbarkeit wird im Bericht das generische Maskulinum verwendet. Alle geschlechtlichen Identitäten sind gleichermaßen angesprochen.

1.1. Motivation

Der PicoNut-Simulator dient der Ausführung einer eigens von der Forschungsgruppe „Effiziente Eingebettete Systeme (EES)“ entwickelten, quelloffenen RISC-V-Prozessorarchitektur in einer simulierten Umgebung. Bislang konnte der Simulator lediglich über eine serielle UART-Schnittstelle mit der Außenwelt kommunizieren. Diese Art der Kommunikation ist auf die Darstellung von Zeichen und einfachen ASCII-Animationen beschränkt. Eine visuelle Darstellung komplexerer geometrischer Strukturen oder gar vollständiger Bilder war somit nicht möglich.

Im Gegensatz dazu verfügte das Vorgängerprojekt, der ParaNut-Prozessor, über eine funktionierende Bildausgabe. Diese ermöglichte nicht nur die grafische Darstellung, sondern erlaubte sogar die Ausführung grafischer Anwendungen wie dem Computerspiel DOOM. Dies war dem PicoNut-Prozessor bisher nicht möglich, weil die entsprechende Schnittstelle für die Darstellung nicht existierte.

Die Erweiterung um eine Grafikausgabe stellt einen bedeutenden Fortschritt dar, da sie die Einsatzmöglichkeiten des Forschungsprozessors wesentlich vergrößert.

Sie ermöglicht es, zukünftige Entwicklungen komplexer zu gestalten und mit grafischen Rückmeldungen zu erweitern. Darüber hinaus fungiert eine GUI als zusätzliche Schnittstelle zwischen Mensch und System, wodurch etwa die Steuerung weiterer Systemkomponenten oder die Analyse von Abläufen intuitiver und effizienter gestaltet werden kann.

Diese Arbeit ist der Projektbericht zur Entstehung dieser Schnittstelle. Sie beinhaltet interessante Einblicke in die technischen Abläufe, den Fluss von Informationen innerhalb des PicoNut-Systems sowie das Zusammenspiel seiner Komponenten.

1.2. Ziel der Arbeit

Ziel dieser Arbeit ist die Dokumentation der Implementierung der Schnittstelle zur grafischen Bildausgabe für den PicoNut-Simulator. Außerdem soll auch die Umsetzung einer grafischen Benutzeroberfläche (GUI) mithilfe des Frameworks Qt, welche die Visualisierung von Ausgaben und die Interaktion mit dem simulierten Prozessor ermöglicht, beschrieben werden.

Im Verlauf des Projekts wurden die ursprünglichen Zielsetzungen in enger Abstimmung mit dem Projektteam sowie dem betreuenden Dozenten, Herrn Kiefer, sinnvoll angepasst. Anlass für diese Änderungen war die praktische Erfahrung, die während der Implementierung gewonnen wurde. Durch die vorgenommenen Anpassungen konnte eine praktikable und effizient umsetzbare Lösung realisiert werden.

Der Berichts ermöglicht einen umfassenden Einblick in die Konzeption, Umsetzung und Funktionalität der entwickelten Schnittstelle. Dabei werden sowohl technische Details als auch übergeordnete Designentscheidungen nachvollziehbar dargestellt. Die Arbeit richtet sich insbesondere an Entwickler, Forschende und Studierende im Bereich eingebetteter Systeme, die sich für die Erweiterung von Mikrokontrollern durch grafische Schnittstellen interessieren.

1.3. Aufbau der Arbeit

Die vorliegende Arbeit ist in sechs Kapitel gegliedert:

Kapitel **1. Einleitung** führt in den Projektbericht ein.

Kapitel **2. Grundlagen** behandelt die fachlichen und technischen Grundlagen, die für das Verständnis der Arbeit erforderlich sind. Dazu zählen insbesondere die Architektur der Grafikschnittstelle und die eingesetzten Technologien wie Qt.

Kapitel **3. Stand der Wissenschaft und Technik** stellt den aktuellen Stand der Wissenschaft und Technik dar. Es werden bestehende Ansätze zur grafischen Ausgabe in Simulationsumgebungen analysiert und deren Fähigkeiten aufgezeigt.

Kapitel **4. Praktische Umsetzung** bildet den Hauptteil der Arbeit und beschreibt die Implementierung der entwickelten Lösung. Nach einem Überblick über das Zusammenspiel des Gesamtsystems werden die einzelnen Teile sowie besondere technische Eigenschaften detailliert erläutert.

Kapitel **5. Fazit** fasst die wichtigsten Erkenntnisse zusammen, diskutiert sie und gibt einen Ausblick auf mögliche Weiterentwicklungen des Projekts.

Kapitel **6. Anhang** beinhaltet den im Rahmen der Arbeit entwickelten Programmcode.

2. Grundlagen

In diesem Kapitel werden die technischen und theoretischen Grundlagen beschrieben, die für das Verständnis des PicoNut-Simulators und seiner grafischen Ausgabe relevant sind. Es werden die grundlegenden Prinzipien der Soft-Peripherie vorgestellt, die Datenflüsse von der Peripherie zur GUI dargestellt sowie die Architektur und Funktionsweise der beteiligten Softwaremodule skizziert. Ziel ist es, die theoretischen Voraussetzungen für das Verständnis der Simulation und deren grafischer Ausgabe zu schaffen.

2.1. Das PicoNut-Projekt

Das PicoNut-Projekt ist eine Initiative der Technischen Hochschule Augsburg mit dem Ziel, einen freien und flexibel erweiterbaren RISC-V-Prozessor für Lehr- und Forschungszwecke zu entwickeln. Der Prozessor basiert auf dem offenen RISC-V-Befehlssatz und wird unter anderem in SystemC modelliert, wodurch sich sowohl Peripherie als auch ein leistungsfähiger Simulator aus demselben Quellcode ableiten lassen. Das Herzstück bildet die CPU Nucleus, die über einen Wishbone-kompatiblen Systembus mit weiteren Modulen wie RAM, UART, GPIO sowie Audio- und Grafik-Schnittstellen verbunden ist. Der Programmcode wird mithilfe der RISC-V-GNU-Toolchain in einem Dockercontainer kompiliert, was eine breite Kompatibilität mit allen Betriebssystemen ermöglicht, auf denen Container ausgeführt werden können. Seit dem Projektstart im Jahr 2024 wird der PicoNut kontinuierlich weiterentwickelt und kommt bereits in ersten Lehrveranstaltungen sowie in Abschlussarbeiten zum Einsatz. [1]

2.2. SystemC

SystemC ist eine C++-basierte Modellierungsbibliothek, die die Beschreibung und Simulation digitaler Systeme auf verschiedenen Abstraktionsebenen ermöglicht. Besonders relevant für das PicoNut-Projekt ist die Modellierung auf algorithmischer Ebene. Im Gegensatz zur Register-Transfer-Ebene, bei der Register und deren Transferfunktionen explizit beschrieben werden, erlaubt die algorithmische Modellierung eine deutlich höhere Abstraktion: Das Verhalten ganzer Schaltungskomponenten wird in wenigen Prozessen zusammengefasst, die auch komplexe Kontrollstrukturen wie Schleifen mit dynamischen Abbruchbedingungen enthalten können.

Die Register werden hierbei implizit behandelt, was den Entwicklungsaufwand reduziert und die Übersichtlichkeit erhöht. Aufgrund dieser Abstraktion lassen sich algorithmische Modelle meist schneller entwickeln und sind weniger fehleranfällig. Besonders eignet sich diese Modellierungsmethode für Algorithmen der digitalen Signal- und Bildverarbeitung. Damit ist sie für die Entwicklung der Grafik-Schnittstelle gut geeignet. [2]

2.3. Schnittstellenbeschreibung Grafik-Schnittstelle

Die Grafik-Schnittstelle des PicoNut-Prozessors dient der Bildausgabe und stellt hierzu eine Reihe von Steuerregistern bereit. Diese ermöglichen die Konfiguration von Auflösung und Farbtiefe sowie den Zugriff auf den Framebuffer und eine Farbpalette. Außerdem stehen Read-only-Prüfregister für das Abfragen der verfügbaren Auflösungs- und Farbmodi zur Verfügung. Ziel ist es, eine flexible Grundlage für die Entwicklung einfacher grafischer Ausgaben im Rahmen von Lehre und Projekten zu schaffen.

Die Register der Grafik-Schnittstelle sind im Little-Endian-Format organisiert und ausschließlich über 32-Bit-Zugriffe adressierbar. Zentrale Steuer- und Statusfunktionen sind über das `CONTROL`- und `STATUS`-Register zugänglich. Hierüber lässt sich u. a. die Bildausgabe aktivieren sowie der Betriebszustand der Schnittstelle abfragen.

Die Auswahl der Bildauflösung erfolgt über das `RESOLUTION_MODE`-Register, das mehrere vordefinierte Modi (z. B. 640 × 480, 800 × 600, 1920 × 1080) unterstützt. Über ein zusätzliches Read-only-Register (`RESOLUTION_MODE_SUPPORT`) kann anhand einer Bitmaske (0 = nicht unterstützt, 1 = unterstützt) abgefragt werden, welche Modi in der jeweiligen Implementierung tatsächlich verfügbar sind.

Ein analoger Mechanismus besteht für die Farbtiefe: Das `COLOR_MODE`-Register erlaubt die Auswahl unterschiedlicher Farbmodi mit 1 bis 32 Bit Farbtiefe, darunter sowohl klassische RGB-Formate (z. B. RGB332, RGB565, RGB888) als auch palettierte und Graustufenmodi. Das zugehörige `COLOR_MODE_SUPPORT`-Register signalisiert die unterstützten Modi mithilfe einer Bitmaske.

Die Farbinformationen werden in einer 256 Einträge umfassenden Farbpalette (`COLOR_MAP`, Adressbereich 0x100–0x4FF) hinterlegt. Jeder Eintrag enthält einen 32-Bit-Farbwert (0x00RRGGBB), wobei die oberen 8 Bit ungenutzt bleiben.

Für die Bildausgabe ist schließlich ein Zeiger auf den Framebuffer über das `FRAMEBUFFER`-Register zu setzen. Zur Synchronisation mit dem Bildaufbau kann über das `SCANLINE`-Register die aktuell ausgegebene Bildzeile abgefragt werden.

Insgesamt sieht die Grafik-Schnittstellenbeschreibung eine einfache, aber erweiterbare Ansteuerung von Bildschirminhalten vor und schafft damit eine gute Grundlage für zukünftige Implementierungen der Schnittstelle.

2.4. Qt Framework und GUI

Qt ist ein plattformübergreifendes Framework zur Entwicklung grafischer Benutzeroberflächen (GUIs). Es unterstützt C++ und Python und bietet zwei grundsätzliche Technologien für UI-Design: `Qt Widgets` für klassische Desktop-Interfaces und `Qt Quick` (QML) für moderne, animierte und touch-optimierte Oberflächen. Die Entwicklung wird durch Tools wie Qt Creator (IDE) und Qt Design Studio (UI-Design) unterstützt. [3]

Für die Bildverarbeitung innerhalb der GUI des PicoNut-Simulators sind die Klassen `QLabel`, `QImage` und `QPixmap` zentrale Komponenten:

- `QLabel` dient der Anzeige von Texten oder Bildern ohne Interaktion. Bilder werden dabei über `setPixmap()` eingebunden. [4]
- `QImage` eignet sich für Bildmanipulation und direkten Pixelzugriff – ideal für I/O-Prozesse und Bildbearbeitung. Für die Darstellung muss ein `QImage`-Objekt in ein `QPixmap`-Objekt umgewandelt werden. [4]
- `QPixmap` ist für die Darstellung von Bildern auf dem Bildschirm optimiert. Dieses Format kann direkt in einem `QLabel` dargestellt werden. [5]

3. Stand der Wissenschaft und Technik

In diesem Kapitel wird der aktuelle Stand der Technik im Bereich der grafischen Ausgabe in Simulations- und Entwicklungsumgebungen dargestellt. Während es zahlreiche ausgereifte Grafik-Architekturen und Algorithmen für anspruchsvolle Anwendungen gibt, zeigen sich diese für den PicoNut-Simulator als überdimensioniert oder unpassend. Ziel der Implementierung der Grafikschnittstelle ist keine weiterentwicklung bestehender Verfahren, sondern die bewusste Reduktion auf eine minimal funktionale und ressourcenschonende Lösung – die Implementierung eines einfachen Framebuffers mit grundlegenden Ein-/Ausgabe Fähigkeiten. Trotzdem gibt es Entwicklungen und Forschungsergebnisse, die einen ähnlichen Anwendungsfall abdecken, die im Folgenden kurz vorgestellt werden.

Einfach gehaltene Grafikschnittstellen für eingebettete Systeme stehen selten im Fokus wissenschaftlicher Forschung. Dennoch existieren einige Ansätze, die sich mit minimalem Ressourcenverbrauch bei der grafischen Ausgabe beschäftigen. Ioan [6] beschreibt eine VGA-Ausgabe auf FPGA-Basis, die durch schematische Optimierung nur 3 % der kombinatorischen und 2 % der sequentiellen Ressourcen eines Spartan-3-Chips beansprucht – ein hardwareorientierter Ansatz für einfache Anzeigehardware ohne komplexe GPU-Funktionalität. Park et al. [7] schlagen eine Softwarelösung für Systeme mit stark eingeschränkten Ressourcen vor: ein textbasiertes User Interface (TUI) mit 30×40-Zeichenauflösung, das unter 7 KB zusätzlichen Speicher benötigt und dennoch interaktive Elemente wie Pop-up-Menüs unterstützt. Zhou et al. [8] entwickeln eine framebufferbasierte Kompressionsarchitektur, die als Hardware-Erweiterung für RISC-V-Prozessoren

Bandbreiten- und Energieeinsparungen erzielt – ein Beispiel für gezielte Effizienzsteigerung auf System-on-Chip-Ebene.

Die in dieser Arbeit realisierte Grafikschnittstelle für das Piconut-System verfolgt ein ähnliches Ziel: eine minimale, aber funktionale Möglichkeit zur Anzeige von Simulationsdaten ohne Einsatz spezialisierter Grafik-Hardware. Dabei wird bewusst auf GPU-Funktionalität verzichtet, zugunsten einer einfachen Speicherstruktur und direkter Softwareunterstützung zur Befüllung und Darstellung. Im Gegensatz zu komplexeren Systemen liegt der Fokus hier nicht auf interaktiven Benutzeroberflächen oder hohen Auflösungen, sondern auf Transparenz, Portabilität und minimaler Systemlast – Aspekte, die auch in den genannten Arbeiten adressiert, aber unterschiedlich umgesetzt wurden.

In der Praxis kommen in eingebetteten Systemen häufig vereinfachte Framebuffer-Lösungen wie LVGL oder µGUI zum Einsatz, für die jedoch kaum akademische Literatur vorliegt. Die Dokumentation beschränkt sich meist auf Herstellerangaben und Open-Source-Projekte. [9], [10]

4. Praktische Umsetzung

Dieses Kapitel beschreibt die konkrete Umsetzung der für das PicoNut-Projekt entwickelten Grafikschnittstelle. Ziel war es, eine schlanke, ressourcenschonende Lösung zu realisieren, die in die bestehende Simulationsumgebung eingebettet werden kann und dort die Ausgabe einfacher grafischer Informationen ermöglicht – ohne den Einsatz spezialisierter Grafik-Hardware oder komplexer Software-Frameworks. Die Implementierung orientiert sich an etablierten Prinzipien der Modularität und Klarheit und gliedert sich in mehrere Teilkomponenten, die im Folgenden einzeln beschrieben werden.

4.1. Überblick über Architektur

Die in dieser Arbeit entwickelte Implementierung verfolgt das Ziel, eine möglichst einfache und leichtgewichtige Grafikschnittstelle für eine Simulationsumgebung bereitzustellen. Im Zentrum steht ein selbst entwickelter Framebuffer, der es erlaubt, Pixelinformationen programmatisch zu setzen und in verschiedenen Datenformaten abzulegen. Dieser Framebuffer stellt die Brücke zwischen der simulierten Hardwareumgebung und der tatsächlichen grafischen Darstellung dar.

Die Architektur ist modular aufgebaut und besteht im Wesentlichen aus zwei Komponenten:

- Der Framebuffer: Diese Komponente verwaltet die Pixeldaten im 32-Bit-Hex-Format in einem eindimensionalen Array. Sie bietet Methoden zum Setzen und Abrufen von Pixelwerten sowie zur Initialisierung und zum Zurücksetzen des Puffers.

- Das GUI-Fenster: Diese Komponente ist für die visuelle Ausgabe des Framebuffers in der Simulationsumgebung zuständig. Hierfür kommt eine Qt-basierte GUI zum Einsatz, die den Framebuffer periodisch ausliest und dessen Inhalt in einem Fenster darstellt.

Der Datenfluss beginnt bei der Software, die von dem PicoNut Simulator ausgeführt wird. Zu Demozwecken wurde ein Programm (`graphics_fancy_random_squares.c`) entwickelt, das Quadrate erzeugt, deren Position, Größe und Farbe zufällig gewählt werden. Diese werden Pixel für Pixel über die bereitgestellten Methoden der Grafikschnittstelle in den Framebuffer geschrieben. Anschließend wird dieser Puffer durch die GUI-Komponente interpretiert und als Bild ausgegeben. Die Aktualisierung erfolgt dabei kontinuierlich mir einer Frequenz von mindestens 30Hz, sodass Änderungen im Framebuffer, wie z. B. der Bildaufbau, in Echtzeit sichtbar werden.

Die folgende Abbildung 1 gibt einen Überblick über diesen Ablauf und zeigt, wie die verschiedenen Komponenten der Grafikschnittstelle zusammenarbeiten:

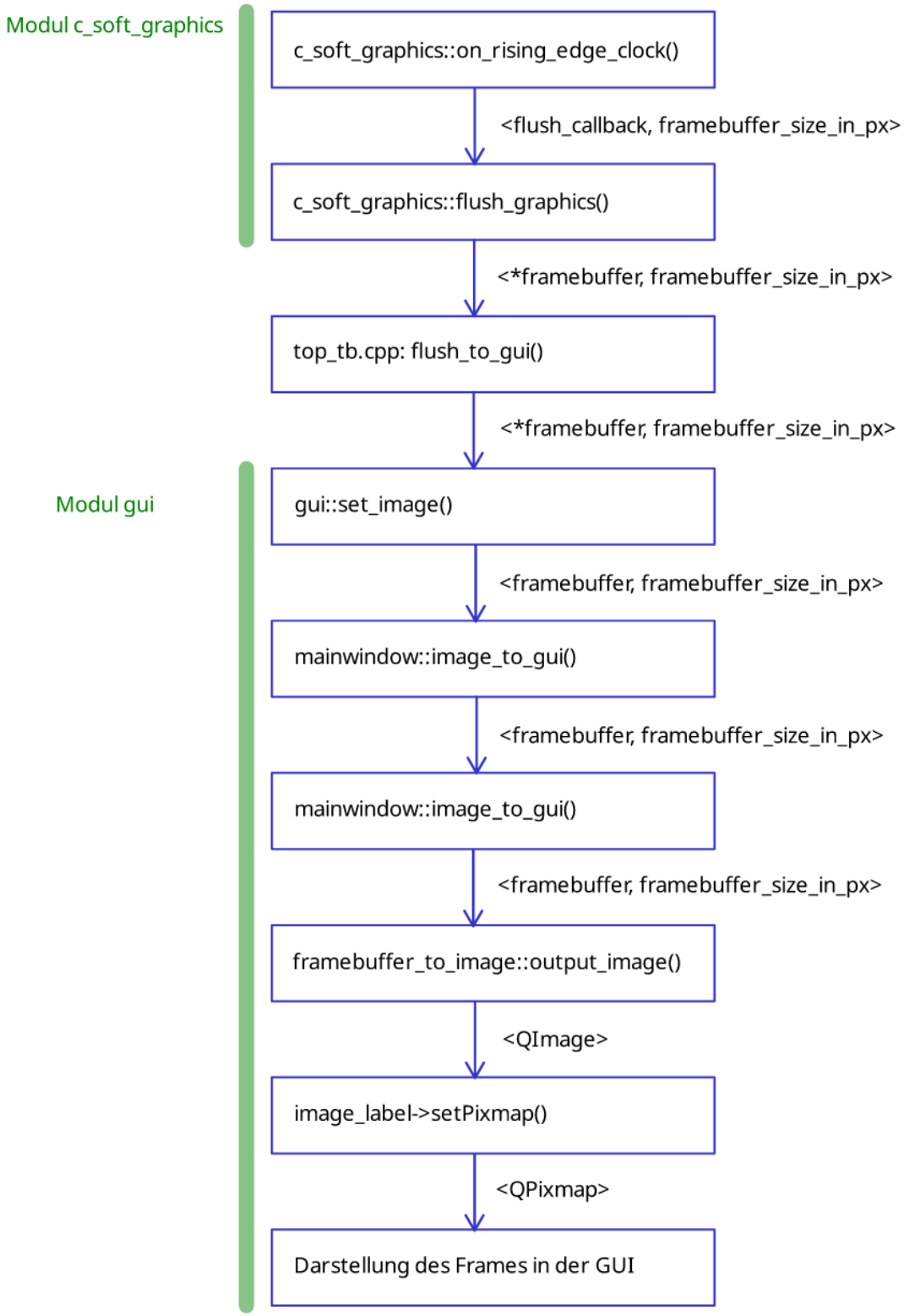


Abbildung 1: Ablaufdiagramm Bildausgabe

4.2. Modulbeschreibung

Das Projekt gliedert sich funktional in zwei Hauptmodule: `c_soft_graphics` und `framebuffer_to_image`. Ersteres befindet sich im gleichnamigen Unterverzeichnisse im Projektordner `hw/peripherals`, während letztes Teil des Unterordners `gui` ist. Dort befinden sich auch die Klassen `gui` und `mainwindow`, die die Funktionsweise der grafischen Oberfläche beschreiben. Beide Module übernehmen klar getrennte Aufgaben: Während `c_soft_graphics` für die zentrale Peripherieemulation inklusive Framebuffer-Datenhaltung verantwortlich ist, übernimmt `framebuffer_to_image` die Konvertierung der rohen Pixeldaten in ein anzeigbares Bildformat für die GUI.

4.2.1. `c_soft_graphics`-Klasse

Die Klasse `c_soft_graphics`, implementiert in den Dateien `c_soft_graphics.cpp` und `c_soft_graphics.h`, beschreibt das grafische Peripheriemodul für die Simulationsumgebung. Sie stellt einen linearen Framebuffer sowie eine Sammlung steuernder Register bereit. Die verwendete Registerstruktur entspricht der Schnittstellenbeschreibung und erlaubt das Auslesen grafikspezifischer Informationen wie Auflösung (`WIDTH`, `HEIGHT`) und unterstützter Funktionen (`RESOLUTION_MODE_SUPPORT`, `COLOR_MODE_SUPPORT`), sowie das Setzen der Framebuffer-Größe (`RESOLUTION_MODE`). Die Register sind innerhalb des Grafik-Peripherie-Adressraums zugänglich. Die Klasse ist damit vollständig unabhängig von der Bildausgabe und konzentriert sich auf die reine Datenhaltung und Konfigurierbarkeit der Grafikschnittstelle.

4.2.2. `framebuffer_to_image`-Klasse

Die Klasse `framebuffer_to_image`, implementiert in `framebuffer_to_image.cpp` und `framebuffer_to_image.h`, dient als Brücke zwischen der reinen Pixeldatenhaltung und der grafischen Darstellung in der Benutzeroberfläche. Beim Aufruf wird ein Zeiger auf den Framebuffer zusammen mit dessen Größe übergeben. Die Klasse konvertiert intern die gespeicherten Hex-Farbwerte in ein Qt-kompatibles Format (`QRgb`) und erzeugt daraus ein `QImage`-Objekt, das für die Weiterverarbeitung in der GUI genutzt wird.

Der Aufruf erfolgt zentral im GUI-Modul (`MainWindow::image_to_gui()`), wo die generierte Bildrepräsentation in einem `QLabel` dargestellt wird. Die Darstellung wird dabei dynamisch an die Fenstergröße angepasst. Der folgende Codeauszug zeigt exemplarisch die Integration der `framebuffer_to_image`-Klasse in die GUI-Logik:

```
framebuffer_to_image fb_to_image(framebuffer, framebuffer_size);
output_img = fb_to_image.output_image();
ui->image_label->setPixmap(QPixmap::fromImage(output_img));
```

4.2.3. `gui`-Klasse

Die grafische Benutzeroberfläche (GUI) wurde entwickelt, um dem PicoNut-Simulator eine visuelle Ausgabe zu ermöglichen. Während der Simulator zuvor ausschließlich textbasiert über die Konsole kommunizierte, bietet die GUI nun die Möglichkeit, Inhalte des simulierten Videospeichers als Bild auszugeben. Damit wird ein wesentlicher Schritt zur Erweiterung der Funktionen des PicoNut-Prozessors erreicht.

Zentraler Bestandteil der GUI ist ein `QLabel`, das ein Bild anzeigt, welches aus dem Framebuffer der `c_soft_graphics`-Peripherie erzeugt wird. Die GUI besitzt keine Bedienelemente im klassischen Sinn, weil sie momentan ausschließlich der Anzeige dient. In der Titelleiste des Fensters werden zusätzliche Informationen dargestellt, darunter die aktuelle Auflösung des Framebuffers sowie die Größe des `QLabel`. Diese Angaben sind vor allem zur Diagnose und für die Validierung der grafischen Ausgabe hilfreich.

Technisch gesehen ist die GUI vollständig in den Simulator integriert und läuft ausschließlich auf dem Host-System innerhalb des Docker-Containers. Eine Ausführung auf echter Hardware ist nicht vorgesehen, da die grafische Ausgabe durch die GUI ausschließlich für die Simulation gedacht ist.

Ein zentrales Element der Darstellung ist die Methode `image_to_gui`, welche im Hauptfenster (MainWindow) definiert ist. Sie übernimmt den Pointer auf den Framebuffer sowie dessen Größe, erzeugt daraus ein QImage-Objekt und übergibt dieses an das QLabel. Um eine korrekte Darstellung bei Fenstergrößenänderun-

gen zu garantieren, wurde auf das automatische Skalieren des Labels sowie die Behandlung von Resize-Events gesetzt. Diese Kombination sorgt dafür, dass das Seitenverhältnis beibehalten wird und die Darstellung nicht verzerrt erscheint.

Die GUI ist minimalistisch gehalten – sowohl im Design als auch in der technischen Umsetzung. Die zugehörige .ui-Datei umfasst lediglich das Hauptfenster mit einem zentralen Bildfeld. Durch diese reduzierte und funktionale Architektur ist die GUI ressourcenschonend und robust.

Insgesamt stellt die GUI eine einfache, aber effektive Lösung zur Darstellung der simulierten grafischen Ausgaben dar und bildet damit eine wichtige Komponente des PicoNut-Systems.

4.2.4. `graphics_fancy_random_squares`-Demoprogramm für Bildausgabe

Um die Funktionsweise der entwickelten `c_soft_graphics`-Komponente in Kombination mit der GUI anschaulich zu demonstrieren, wurde das Testprogramm `graphics_fancy_random_squares` entwickelt. Es dient der Validierung der korrekten Ansteuerung des Framebuffers sowie der Darstellung über die grafische Oberfläche und stellt ein einfaches, visuelles Beispiel für die praktische Nutzung des Grafikmoduls dar.

Das Programm erzeugt zufällig gefärbte Quadrate verschiedener Größe an zufälligen Positionen innerhalb des Framebuffers. Jedes Quadrat wird direkt in den simulierten Videospeicher geschrieben, welcher anschließend über die GUI sichtbar wird. Der Inhalt des Framebuffers ändert sich dabei mit jeder Iteration und erzeugt dadurch eine einfache, aber effektive visuelle Rückmeldung über die korrekte Funktion der gesamten Signalkette von der Pixelerzeugung bis zur Darstellung.

Der Aufbau des Programms ist dabei bewusst einfach gehalten: Die Methode `create_square()` erzeugt einen vollständig schwarzen Puffer und zeichnet anschließend ein farbiges Quadrat mit zufälligen Parametern in diesen Speicherbereich. Dieser wird dann in den Framebuffer der `c_soft_graphics`-Peripherie übertragen. Die Farbwerte sind 24-Bit RGB-Werte, die direkt in das erwartete Speicherlayout geschrieben werden.

Obwohl das Programm konzeptionell als Endlosschleife gedacht ist (mittels `while(true)`), wurde die Schleife aus praktischen Gründen auf eine feste Anzahl von Durchläufen (`for (int i = 0; i < 10; ++i)`) begrenzt. Diese Ansatz wurde gewählt, damit das automatisierte Testsystem Jenkins das Demoprogramm ausführen kann. Ein nicht-terminierendes Testprogramm in diesem Kontext würde den dazugehörigen Test fehlschlagen lassen. Für den manuellen Einsatz oder zur Demonstration in Präsentationen kann die Schleife entsprechend wieder zu einer Endlosschleife umgebaut werden.

Das Programm kann mithilfe dieses Befehls im Verzeichnis `sw/applications/graphics_fancy_random_squares/` ausgeführt werden:

```
PN_SYSTEM=refdesign_c_soft_graphics make sim
```

Ein Beispiel in Abbildung 2 zeigt die GUI, während das Demoprogramm ausgeführt wird:

Abbildung 2: Ausführung des `graphics_fancy_random_squares`-Demoprogramms

Insgesamt zeigt das Demoprogramm sehr anschaulich, wie die entwickelte Grafik-Peripherie, der Speicherzugriff und die Visualisierung über die GUI nahtlos zusammenwirken.

4.3. Besondere Herausforderungen und Lösungen

Im Verlauf der Implementierung ergaben sich mehrere technische Herausforderungen, die jeweils spezifische Lösungen erforderten, um die Integration der einzelnen Module und die Ausführung im Gesamtsystem zu gewährleisten. Einige dieser Aspekte werden im Folgenden näher erläutert.

4.3.1. Callback-Mechanismus zur Benachrichtigung der GUI

Da die Klasse `c_soft_graphics` unabhängig von der grafischen Benutzeroberfläche operiert, war eine zentrale Herausforderung die Umsetzung eines Mechanismus zur Benachrichtigung der GUI über Änderungen im Framebuffer. Dieses Problem wurde durch die Einführung eines Callbacks gelöst: Die `c_soft_graphics`-Instanz hält einen Funktionszeiger, der auf eine Methode der GUI zeigt. Diese Methode wird mindestens 30 Mal pro Sekunde getrigger, um eine flüssige Neuzeichnen in der GUI anzustoßen.

4.3.2. Automatische Skalierung des Bild-Labels in der GUI

Die Benutzerfreundlichkeit der GUI wurde durch die automatische Anpassung des Bildbereichs (`QLabel`) an die Fenstergröße erhöht. Hierzu wurde die Eigenschaft `setScaledContents(true)` genutzt, die eine dynamische Größenanpassung des Bildinhalts an die Größe des Widgets erlaubt. Zusätzlich wurde durch das Überschreiben der `resizeEvent`-Methode ein Mechanismus implementiert, der das Beibehalten des Seitenverhältnis des GUI-Fensters erzwingt, wenn es in der Größe verändert wird, damit eine Größenänderung den Bildinhalt nicht verzerrt. Auch diese Eigenschaft wurde programmgesteuert implementiert, um die Verwendbarkeit der Oberfläche zu verbessern.

4.3.3. Anpassung der Buildsysteme (Makefiles)

Ein weiterer nicht-trivialer Aspekt war die Integration der unterschiedlichen Module in ein gemeinsames Buildsystem. Während `c_soft_graphics` und andere Module bereits mit den Makefiles kompiliert werden konnten, stellte die Integration des Qt-Frameworks besondere Anforderungen. Hierzu gehörte insbesondere die Verwendung von automatisch generierten MOC-Dateien und das Anpassen der Qt-Pfade. Die Makefiles wurden entsprechend angepasst, sodass die Qt-libraries als Teil der Build-Kette im Graphics-Modul verfügbar sind. Durch das containerisieren der Build-Tools, wie im nächsten Abschnitt beschrieben, wurde der Build-Prozess vereinfacht.

4.3.4. Ausführung innerhalb des PicoNut-Docker-Containers

Das Projekt wurde innerhalb des speziell für den PicoNut-Simulator vorgesehenen Docker-Containers gebaut. Dies erforderte zusätzliche Konfigurationen, um Qt-basierte Anwendungen innerhalb des Containers compilierbar zu machen. Hierzu zählten die Freigabe von Display-Umgebungsvariablen und das Einbinden von Qt-libraries. Durch diese Maßnahmen konnte die grafische Ausgabe direkt aus dem Container heraus erfolgen.

4.4. Zusammenfassung

Mit der beschriebenen Implementierung wurde eine minimalistische, aber funktionsfähige Grafikschnittstelle realisiert, die das Schreiben von Bilddaten in einen Framebuffer ermöglicht und diese Daten anschließend in einer grafischen Benutzeroberfläche visualisiert. Die Lösung zeigt, dass bereits mit sehr begrenzten Mitteln – ohne komplexe Grafikprozessoren oder umfangreiche Middleware – eine funktionale Bildausgabe in einem simulierten eingebetteten System umsetzbar ist.

Die entwickelte Architektur zeichnet sich durch eine klare Modularisierung aus: Die Trennung der Klassen zur Datenhaltung (`c_soft_graphics`) und Visualisierung (`gui`) erlaubt eine einfache Erweiterbarkeit. Neue Visualisierungskomponenten oder alternative Ausgabemedien könnten ohne tiefgreifende Änderungen am bestehenden System angebunden werden. Ebenso erlaubt der Callback-Mechanismus eine reaktive Kommunikation zwischen Backend und GUI, was zukünftige Funktionen wie ereignisgesteuerte Aktualisierungen begünstigt.

Eine detaillierte Dokumentation des Programmcodes befindet sich im Anhang dieser Arbeit. Sie ergänzt die Beschreibung in diesem Kapitel und hält den Stand der Entwicklung zum Zeitpunkt der Abgabe fest.

5. Fazit

In diesem Kapitel werden die zentralen Ergebnisse der Arbeit zusammengefasst, die entwickelte Lösung bewertet und ein Ausblick auf mögliche Weiterentwicklungen gegeben. Ziel ist es, die wesentlichen Erkenntnisse einzuordnen und Perspektiven für zukünftige Arbeiten aufzuzeigen.

5.1. Zusammenfassung der Arbeit

Ziel dieser Arbeit war es, eine einfache, aber funktionale Schnittstelle zur grafischen Ausgabe innerhalb der Simulationsumgebung des PicoNut-Systems zu entwickeln. Im Fokus standen dabei folgende Ziele:

1. Die Entwicklung einer Qt-basierten Anwendung zur Bildausgabe für den Simulator.
2. Die Implementierung der `c_soft_graphics`-Komponente, die als Peripheriemodul einen Videospeicher (Framebuffer) und mehrere Kontrollregister bereitstellt und über den Systembus mit der Speicherverwaltungseinheit (memu) kommuniziert.
3. Schaffen der Grundlage für die Bedienung des PicoNut-Simulators mithilfe eines Joysticks, einer Tastatur oder Steuerelementen der Qt-Anwendung.

Zur Umsetzung dieser Ziele kamen vor allem C++ und das Qt-Framework zum Einsatz. Dabei wurden zwei zentrale Bestandteile implementiert: Zum einen die `c_soft_graphics`-Klasse als modulare Komponente für den Speicherzugriff und die Datenhaltung für grafische Darstellung, inklusive Steuerregistern und Framebuffer. Diese wurde erfolgreich in das bestehende PicoNut-System eingebunden. Zum anderen eine grafische Benutzeroberfläche, die den Framebuffer-Inhalt visualisiert.

Das wichtigste Ergebnis dieser Arbeit ist die erfolgreiche Umsetzung der grafischen Ausgabe innerhalb der PicoNut-Simulationsumgebung. Die Qt-GUI bietet eine benutzerfreundliche Visualisierung des Framebuffer-Inhalts und bildet damit eine wertvolle Erweiterung der bisherigen, rein textbasierten Ausgabe des Simulators. Die entwickelten Schnittstellen sind funktional, modular und bilden eine solide Basis für künftige Erweiterungen.

Neben der Implementierung entstand außerdem eine umfassende Dokumentation, die sowohl die Nutzung der Grafik-Peripherie und der GUI als auch die technischen Details der Schnittstelle beschreibt.

5.2. Diskussion der Ergebnisse

Die in dieser Arbeit gesetzten Ziele wurden im Wesentlichen vollständig erreicht. Die grafische Benutzeroberfläche wurde erfolgreich mit Qt umgesetzt und ermöglicht die visuelle Darstellung des Framebuffers. Die `c_soft_graphics`-Komponente wurde mit einem eigenen Framebuffer sowie mehreren Steuerregistern implementiert und in das PicoNut-System integriert. Lediglich optionale Erweiterungen, wie z. B. Eingabe über Tastatur oder Buttons, wurden im Rahmen dieser Arbeit nicht umgesetzt.

Vorteile der entwickelten Lösung sind der modulare Aufbau und die klare Gliederung der Funktionen in eigene Klassen, was die Wartbarkeit erhöht und zukünftige Erweiterungen erleichtert. Die Trennung zwischen der GUI und der Grafikschnittstelle stellt gängige Praxis in der objektorientierten Softwareentwicklung dar.

Allerdings gibt es auch Einschränkungen. Die Lösung ist stark an das Qt-Framework gebunden, was die Portierbarkeit auf andere Plattformen einschränken kann. Eine leichtere, weniger abhängige Architektur, z. B. auf Socket-Kommunikation basierend, könnte diese Abhängigkeiten reduzieren.

Im Verlauf der Arbeit ergaben sich einige technische und konzeptionelle Herausforderungen, die pragmatisch gelöst wurden. So wurden die ursprünglich gesetzten Ziele – etwa die Entwicklung eines eigenständigen Videoprotokolls für die Socket-Kommunikation – im Team diskutiert und zugunsten einer Integration von Qt verworfen. Auch die Idee, den Framebuffer in den Hauptspeicher des PicoNut-

Systems zu verlagern und per DMA anzubinden, wurde überlegt. Aufgrund der erhöhten Komplexität und technischer Einschränkungen der verwendeten Simulationsumgebung wurde dieser Ansatz letztlich verworfen, und der Framebuffer blieb Teil der `c_soft_graphics`-Peripheriekomponente.

5.3. Ausblick

Die im Rahmen dieser Arbeit entwickelte Lösung bildet eine solide Grundlage für zukünftige Erweiterungen im PicoNut-Projekt. Mehrere Weiterentwicklungen sind denkbar, sowohl auf funktionaler als auch auf technischer Ebene.

Ein naheliegender nächster Schritt wäre die Integration einer Audioausgabe und -steuerung in die bestehende GUI. Dadurch ließe sich die Simulator-Funktionalität erweitern und um zusätzliche Ausgabeformen ergänzen. Auch die Implementierung von Eingabemöglichkeiten, z. B. über Tastatur oder Joystick, wäre eine wertvolle Erweiterung. Damit könnte der Simulator nicht nur Ergebnisse anzeigen, sondern auch interaktive Szenarien ermöglichen.

Auf technischer Ebene wäre die Ergänzung von Zeichenfunktionen oder einfachen Rasterisierungsalgorithmen im c_soft_graphics-Modul denkbar. Dies würde die grafische Ausdruckskraft der Peripherie erheblich steigern und z. B. das Zeichnen von Linien, Kreisen oder Text ermöglichen, ohne dass dazu komplexe Logik innerhalb ausgeführter Programme nötig wäre.

Auch forschungsnahe Anschlussfragen ergeben sich: Insbesondere die Performance könnte durch gezielte Optimierungen weiter verbessert werden. Ein potenzieller Ansatz wäre, nur die Pixel in den Framebuffer zu übertragen oder zu aktualisieren, deren Inhalte sich seit der letzten Bildausgabe geändert haben. Solche Verbesserungen im Rahmen zukünftiger Arbeiten könnten die Bildaufbauzeiten deutlich reduzieren.

Literaturverzeichnis

[1] PicoNut Projekt, „PicoNut - Forschungsgruppe Effiziente Eingebettete Systeme“. Zugegriffen: 30. Mai 2025. [Online]. Verfügbar unter: https://ees.tha.de/piconut/index.html

[2] F. Kesel und R. Bartholomä, *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs: Einführung mit VHDL und SystemC*. Oldenbourg Wissenschaftsverlag, 2013. doi: 10.1524/9783486747157.

[3] B. Coepp, *Introducing Qt 6: Learn to Build Fun Apps & Games for Mobile & Desktop in C++*. Berkeley, CA: Apress, 2022. doi: 10.1007/978-1-4842-7490-3.

[4] „QLabel Class | Qt Widgets | Qt 6.9.0“. Zugegriffen: 30. Mai 2025. [Online]. Verfügbar unter: https://doc.qt.io/qt-6/qlabel.html

[5] „QPixmap Class | Qt GUI | Qt 6.9.0“. Zugegriffen: 30. Mai 2025. [Online]. Verfügbar unter: https://doc.qt.io/qt-6/qpixmap.html#details

[6] A. D. Ioan, „Designing an optimal single chip FPGA video interface for embedded systems“, in *2010 3rd International Symposium on Electrical and Electronics Engineering (ISEEE)*, Sep. 2010, S. 58–63. doi: 10.1109/ISEEE.2010.5628542.

[7] J. Park, N. Baek, und H. Lee, „Design of a small footprint embedded graphics system“, in *The 1st IEEE Global Conference on Consumer Electronics 2012*, Okt. 2012, S. 187–188. doi: 10.1109/GCCE.2012.6379574.

[8] Y. Zhou, X. Jin, T. Xiang, und D. Zha, „Enhancing energy efficiency of RISC-V processor-based embedded graphics systems through frame buffer

compression“, *Microprocessors and Microsystems*, Bd. 77, S. 103140, Sep. 2020, doi: 10.1016/j.micpro.2020.103140.

[9] G. İşnas und N. Şenyer, „Comparison of TouchGFX and LVGL Embedded Hardware GUI Libraries“, *Gazi Üniversitesi Fen Bilimleri Dergisi Part C: Tasarım ve Teknoloji*, Bd. 9, Nr. 3, S. 373–384, Sep. 2021, doi: 10.29109/gujsc.915163.

[10] „µGUI - free Open Source GUI module for embedded systems | Embedded Lightning“. Zugegriffen: 31. Mai 2025. [Online]. Verfügbar unter: https://embeddedlightning.com/ugui/

6. Anhang

6.1. c_soft_graphics-Klasse

6.1.1. c_soft_graphics.h

```
/**
 * @addtogroup c_soft_graphics
 * @author Konrad Armbrecht
 *
 * This module is used to simulate the graphics peripheral.
 * The module is implemented as a soft peripheral and can be used in the
 * simulation environment.
 *
 * The module has a register for storing the output-images, called
framebuffer.
 * The framebuffer is set up as a single large array where all output-
image lines
 * are stored successively. The size is calculated by width and height
given from
 * the constructor. With a 10 × 10 px resolution, you have 100 array
elements, and the
 * framebuffer[10] element is the first pixel of the second output-image
line.
 * The module has a 32-bit memory interface and can be accessed by the
 * soft peripheral interface.
 *
 * Note: The module is not synthesizable and is only used for simulation
purposes.
 */
```

```
#ifndef __C_SOFT_GRAPHICS_H__
#define __C_SOFT_GRAPHICS_H__

#include "c_soft_peripheral.h"
#include <base.h>
#include <cstdint> // For fixed width integer types
#include <cinttypes> //for properly formatted output
#include <functional>
#include <piconut-config.h>
#include <softmemu.h>


class c_soft_graphics : public c_soft_peripheral
{
public:

public:
    /**
     * @brief Registers of the c_soft_graphics module
     */
    struct regs_t
    {
        uint32_t control; ///< Control register, not in use in this
implementation.
        uint32_t width; ///< Graphics output resolution width.
        uint32_t height; ///< Graphics output resolution height.
        uint32_t resolution_mode; ///< Available resolution modes.
        uint32_t resolution_mode_support; ///< Number of supported modes,
to check if the requested mode exists. Read only.
        uint32_t color_mode; ///< Available color mode in bit.
        uint32_t color_mode_support; ///< Number of color modes, to check
if the requested mode exists. Read only.
        uint32_t *framebuffer;  ///< Pixel color information. Pointer for
variable size.
    };

    uint32_t framebuffer_size_in_px;
```

```
    uint64_t base_address;
    uint64_t size;

    /**
     * @brief Register address offsets
     */
    enum e_regs
    {
        CONTROL = 0x0,
        WIDTH = 0x4,
        HEIGHT = 0x8,
        RESOLUTION_MODE = 0xC,
        RESOLUTION_MODE_SUPPORT = 0x10,
        COLOR_MODE = 0x14,
        COLOR_MODE_SUPPORT = 0x18,
        FRAMEBUFFER = 0x1C
    };

    /// Supported resolution modes, for readability
    enum class ResolutionMode : uint32_t {
        Mode_3x2     = 0,
        Mode_80x60   = 1,
    };

    /// Supported color modes, for readability
    enum class ColorMode : uint32_t {
        Mode_RGB888 = 0,
    };

    /**
     * @brief Constructor
     *
     * @param base_address base address of the peripheral in the address
space of the simulation
     * @param resolution_mode resolution mode (recommended for
performance reasons: "1" -> 60x80 px)
     * @param color_mode color depth in bits per pixel (here only "4" is
implemented -> 32 bit)
```

```
     * @param clock_frequency maximum value for interrupt counter
     * @param flush_func Callback function to flush the framebuffer to
GUI.
     * @param callback_signal_graphics_interrupt Callback function to
signal graphics interrupts.
     */
    c_soft_graphics(
        uint64_t base_address,
        ResolutionMode resolution_mode,
        ColorMode color_mode,
        uint32_t clock_frequency,
        void (*flush_func)(uint32_t* framebuffer, uint32_t
framebuffer_size_in_px),
        std::function<void(bool)> callback_signal_graphics_interrupt =
nullptr
    );

    /**
     * @brief Destructor
     */
    ~c_soft_graphics();

    /**
     * @brief Flushes the graphics framebuffer using the provided
callback function.
     *
     * @param flush_to_gui Callback function to send framebuffer data to
the GUI.
     */

    void flush_graphics(void (*callback)(uint32_t*, uint32_t), uint32_t
framebuffer_size);

    /**
     * @brief Register a callback function for graphics interrupt
changes.
     *
     * @param callback The callback function
```

```
     */
    void register_meip_callback(std::function<void(bool)> callback);

    /**
     * @brief Triggers interrupts when max_counter value is reached.
     * This function is automatically called on every rising edge of the
clock
     */
    void on_rising_edge_clock() override;

    // c_soft_peripheral
    const char* get_info() override;
    bool is_addressed(uint64_t adr) override;

    uint32_t read32(uint64_t adr) override;
    void write32(uint64_t adr, uint32_t data) override;

private:
    void clear_framebuffer();
    void create_framebuffer(uint32_t new_size);

    // c_soft_peripheral
    char name[32] = "Graphics";
    regs_t registers;
    void (*flush_callback)(uint32_t* framebuffer, uint32_t
framebuffer_size_in_px);

    // interrupt registers
    uint32_t meip;                // graphics interrupt external signal
register
    uint64_t counter;             // interrupt counter
    uint32_t clock_frequency;     // piconut system clock frequency
    uint32_t v_sync_frequency;    // image refresh rate
    uint32_t max_counter;         // maximum value for interrupt counter

    // Interrupt callback function
    std::function<void(bool)> callback_signal_graphics_interrupt;
};
```

```
#endif
```

6.1.2. c_soft_graphics.cpp

```
#include "c_soft_graphics.h"
#include <iostream>
#include <cstring> // Für std::memcpy

c_soft_graphics::c_soft_graphics(
    uint64_t base_address,
    ResolutionMode resolution_mode,
    ColorMode color_mode,
    uint32_t clock_frequency,
    void (*flush_to_gui)(uint32_t* framebuffer, uint32_t
framebuffer_size_in_px),
    std::function<void(bool)> callback_signal_graphics_interrupt)
    : base_address{base_address}
    , clock_frequency(clock_frequency)
    , flush_callback(flush_to_gui)
    ,
callback_signal_graphics_interrupt(callback_signal_graphics_interrupt)
{
    // init
    registers.framebuffer = nullptr;
    registers.control = 0;

    // set number of last possible resolution_mode-case here (adapt to
number of resolution modes, see switch case in write32 function)
    // based on resolution_mode, framebuffer is created in
resize_framebuffer function, called by write32 function.
    registers.resolution_mode_support = 1;
    write32(base_address + RESOLUTION_MODE,
static_cast<uint32_t>(resolution_mode));

    // set number of last possible color_mode-case here (adapt to number
of color modes, see switch case in write32 function)
    registers.color_mode_support = 0;
    write32(base_address + COLOR_MODE,
```

```
static_cast<uint32_t>(color_mode));

    // fb_size_in_px gets calulated in write32 function. Add ~100 for 25
registers, even if less are used. Mult by 4 to get size in bytes.
    size = (framebuffer_size_in_px + 100) * 4;


    // v-sync counter, used in on_rising_edge_clock function
    // in theory, you'd calculate 60Hz like this, but the simulation is
too slow -> max_counter is set to a static practical value
    // counter = 0;
    // v_sync_frequency = 60; // Hz
    // max_counter = clock_frequency / v_sync_frequency;
    max_counter = 10000;

    // Initial state: no interrupts pending
    if(callback_signal_graphics_interrupt)
    {
        callback_signal_graphics_interrupt(false);
    }
}

c_soft_graphics::~c_soft_graphics()
{
    delete[] registers.framebuffer;
}

const char* c_soft_graphics::get_info()
{
    uint64_t last_address = base_address + size * 4 - 1;
    static char info[128]; // Static to avoid memory management issues
    snprintf(info, sizeof(info), "Name: %s,\nBase Address: 0x%" PRIx64
" ,\nSize: %" PRIu64 " B\nLast Address: 0x%" PRIx64 "\n", name,
base_address, size, last_address);
    return info;
}

bool c_soft_graphics::is_addressed(uint64_t adr)
```

```
{
    return adr >= base_address && adr < size + base_address;
}

uint32_t c_soft_graphics::read32(uint64_t adr)
{
    uint32_t internal_address = adr - this->base_address;

    // set last two bits to 0
    internal_address &= ~0b11;

    // skip switch case when framebuffer is addressed
    if(internal_address > 0x18)
    {
        return registers.framebuffer[(internal_address - FRAMEBUFFER) >>
2];
    }

    // Read only from start of register
    switch(internal_address)
    {
        case CONTROL:
            PN_WARNING("Read from control register. Register is not in
use.");
            return registers.control;
            break;
        case WIDTH:
            return registers.width;
            break;
        case HEIGHT:
            return registers.height;
            break;
        case RESOLUTION_MODE:
            return registers.resolution_mode;
            break;
        case RESOLUTION_MODE_SUPPORT:
            return registers.resolution_mode_support;
            break;
```

```
        case COLOR_MODE:
            return registers.color_mode;
            break;
        case COLOR_MODE_SUPPORT:
            return registers.color_mode_support;
            break;
        default:
            PN_WARNING("Invalid read32 address");
            return 0;
    }
}

void c_soft_graphics::write32(uint64_t adr, uint32_t data)
{
    // set last two bits to 0
    uint32_t internal_address = (adr - this->base_address) & ~(0b11);

    // skip switch if adr is in framebuffer
    if((internal_address >= 0x1C) && (internal_address <= 0x1C +
registers.width * registers.height * 4))
    {
        registers.framebuffer[(internal_address - FRAMEBUFFER) >> 2] =
data;
    }
    else
    {
        // Write only from start of register
        switch(internal_address)
        {
            case CONTROL:
                PN_WARNING("Written to control register. Register is not
in use.");
                break;
            case WIDTH:
                registers.width = data;
                break;
            case HEIGHT:
                registers.height = data;
```

```
                break;
            case RESOLUTION_MODE:
                if(data <= registers.resolution_mode_support)
                {
                    registers.resolution_mode = data;
                    switch(data)
                    {
                        case 0:
                            create_framebuffer(3 * 2);
                            registers.width = 3;
                            registers.height = 2;
                            framebuffer_size_in_px = registers.width *
registers.height;
                            PN_INFOF(("Graphics resolution mode %d set.
Width: %d, height: %d, framebuffer size in px: %d", data,
registers.width, registers.height, framebuffer_size_in_px));
                            break;
                        case 1:
                            create_framebuffer(80 * 60);
                            registers.width = 80;
                            registers.height = 60;
                            framebuffer_size_in_px = registers.width *
registers.height;
                            PN_INFOF(("Graphics resolution mode %d set.
Width: %d, height: %d, framebuffer size in px: %d", data,
registers.width, registers.height, framebuffer_size_in_px));
                            break;
                        default:
                            PN_WARNING("Invalid resolution_mode.");
                            return;
                    }
                }
                else
                {
                    PN_WARNINGF(("Attempted to write invalid
resolution_mode '%d'. Aborted. Previously set value '%d' remains.", data,
registers.resolution_mode));
                }
```

```
                break;
            case COLOR_MODE:
                if(data <= registers.color_mode_support)
                {
                    registers.color_mode = data;
                    switch(data)
                    {
                        case 0:
                            // 24 bit true color, alpha channel not used
-> 32 bit per pixel for simple memory fit
                            registers.color_mode = 0;
                            PN_INFO("Graphics color mode 0 set. Color
depth in bit: 24 (RGB888)");
                            break;
                        default:
                            PN_WARNING("Invalid color_mode.");
                            return;
                    }
                }
                else
                {
                    PN_WARNINGF(("Attempted to write invalid color_mode
'%d'. Aborted. Previously set value '%d' remains.", data,
registers.color_mode));
                }
                break;
            default:
                PN_WARNING("Invalid write32 address");
                return;
        }
    }
}

void c_soft_graphics::flush_graphics(void (*callback)(uint32_t*,
uint32_t), uint32_t framebuffer_size)
{
    if(flush_callback)
    {
```

```
        callback(registers.framebuffer, framebuffer_size_in_px);
    }
}

void c_soft_graphics::clear_framebuffer()
{
    std::fill(registers.framebuffer, registers.framebuffer +
(registers.width * registers.height), 0);
    PN_INFO("Framebuffer cleared.");
}

void c_soft_graphics::create_framebuffer(uint32_t new_size)
{
    uint32_t* new_framebuffer = new uint32_t[new_size]();
    if(registers.framebuffer != nullptr)
    {
        PN_WARNING("Changing framebuffer size during runtime not
supported. Please restart simulation and change resolution mode in
costructor, or call clear_framebuffer() when changing resolution.");
    }
    registers.framebuffer = new_framebuffer;
    framebuffer_size_in_px = new_size;
}

void c_soft_graphics::on_rising_edge_clock()
{
    // Increment the graphics v-sync counter
    counter++;

    // If the counter counted to 1/60s (60Hz), trigger interrupt.
    // In this case max counter is reduced to 1000 because software
simulation runs too slow and a cpu cycle needs more than 10ns -> 60Hz
    if(counter >= max_counter)
    {
        if(callback_signal_graphics_interrupt)
        {
            callback_signal_graphics_interrupt(true);
            PN_INFO("Graphics interrupt triggered");
```

```
        }

        flush_graphics(flush_callback, framebuffer_size_in_px);
        counter = 0;
    }
    else
    {
        if(callback_signal_graphics_interrupt)
        {
            callback_signal_graphics_interrupt(false);
        }
    }
}

void c_soft_graphics::register_meip_callback(std::function<void(bool)>
callback)
{
    callback_signal_graphics_interrupt = callback;
    PN_INFOF(("c_soft_graphics: Registered MEIP callback"));
}
```

6.1.3. c_soft_graphics_tb.cpp

```
#include <stdint.h>
#include <systemc.h>
#include <softmemu.h>
#include <base.h>
#include "c_soft_graphics.h"

#define PERIOD_NS 10.0

// Testbench global variable for validating flush callback trigger
uint32_t flush_triggered_times = 0;


// initialize TB signals
sc_signal<bool> PN_NAME(clk);
sc_signal<bool> PN_NAME(reset);
```

```
// IPort Signals ...
sc_signal<bool> PN_NAME(stb_iport);
sc_signal<sc_uint<32>> PN_NAME(adr_iport);
sc_signal<sc_uint<4>> PN_NAME(bsel_iport);
sc_signal<sc_uint<32>> PN_NAME(rdata_iport);
sc_signal<bool> PN_NAME(ack_iport);
// DPort Signals ...
sc_signal<bool> PN_NAME(stb_dport);
sc_signal<bool> PN_NAME(we_dport);
sc_signal<sc_uint<32>> PN_NAME(adr_dport);
sc_signal<sc_uint<32>> PN_NAME(wdata_dport);
sc_signal<sc_uint<4>> PN_NAME(bsel_dport);
sc_signal<sc_uint<32>> PN_NAME(rdata_dport);
sc_signal<bool> PN_NAME(ack_dport);

void run_cycle(int cycles = 1)
{
    for (int i = 0; i < cycles; i++)
    {
        clk = 0;
        sc_start(PERIOD_NS / 2, SC_NS);
        clk = 1;
        sc_start(PERIOD_NS / 2, SC_NS);
    }
}

/**
 * @brief simulating a read request from core
 *
 * @param adr address for the read request
 * @param bsel byte select for the read request
 * @param IorD if true, read from IPort, else read from DPort
 * @return uint32_t the requested data
 */
uint32_t read_request(uint64_t adr, uint8_t bsel, bool IorD)
{
    uint32_t data_var;
    if (IorD)
```

```
    {
        // IPort read from memory ...
        stb_iport = 1;          // set the strobe signal
        adr_iport = adr;        // set the address
        bsel_iport = bsel;      // set to 32 bit read
        run_cycle(1);
        stb_iport = 0;          // reset stb signal
        // RUN CYCLE SET TO "2" TO AVOID MEMORY WRITE FAILURE
        run_cycle(2);
        data_var = rdata_iport.read(); // read the data
    }
    else
    {
        stb_dport = 1;          // set the strobe signal
        we_dport = 0;           // reset the write enable signal
        adr_dport = adr;        // set the address
        bsel_dport = bsel;      // set to 32 bit read
        run_cycle(1);
        stb_dport = 0;          // reset stb signal
        // RUN CYCLE SET TO "2" TO AVOID MEMORY WRITE FAILURE
        run_cycle(2);
        data_var = rdata_dport.read(); // read the data
    }
    return data_var;
}

/**
 * @brief simulating a write request from core
 *
 * @param data data to write
 * @param adr address to write to
 * @param bsel byte select for the write request
 */

void write_request(uint32_t data, uint64_t adr, uint8_t bsel)
{
    // DPort write to memory ...
    stb_dport = 1;            // set the strobe signal
```

```
    we_dport = 1;             // set the write enable signal
    adr_dport = adr;   // set the address
    wdata_dport = data; // set the data to write
    bsel_dport = bsel;         // set to 32 bit write
    run_cycle(1);
    stb_dport = 0; // reset stb signal
    we_dport = 0;  // reset we signal
    run_cycle(2);
}

/**
 * @brief Sends framebuffer data to the console.
 *
 * @param framebuffer Pointer to the framebuffer data.
 */

void flush_to_gui(uint32_t *framebuffer, uint32_t framebuffer_size)
{
    // No flushing to the GUI, as Qt is not available in the graphics
periphery, but only in top_tb.
    // Counting function calls instead to check counter value in
testbench code below to ensure proper callback trigger function.
    flush_triggered_times++;
}

int sc_main(int argc, char **argv)
{

    pn_parse_enable_trace_core = 1;
c_soft_graphics_tb.cpp      // enable core trace dump
    PN_PARSE_CMD_ARGS(argc, argv);                          // parse
command line arguments
    sc_trace_file *tf = PN_BEGIN_TRACE("softmemu_tb");      // create
trace file
    FILE *coreDumpFile = fopen(NUCLEUS_TRACE_FILE, "a");    // open
coreDump file for core requests

    // Initialiaze the Design under Testing (DUT)
```

```
    m_soft_memu dut_inst{"dut_inst"}; // this is the Design name needed
by the svc_tool

    // connects signals from TOP to TB
    dut_inst.clk(clk);
    dut_inst.reset(reset);

    // IPort Signal Map...
    dut_inst.stb_iport(stb_iport);
    dut_inst.adr_iport(adr_iport);
    dut_inst.bsel_iport(bsel_iport);

    dut_inst.rdata_iport(rdata_iport);
    dut_inst.ack_iport(ack_iport);
    // DPort Signal Map...
    dut_inst.stb_dport(stb_dport);
    dut_inst.we_dport(we_dport);
    dut_inst.adr_dport(adr_dport);
    dut_inst.wdata_dport(wdata_dport);
    dut_inst.bsel_dport(bsel_dport);
    dut_inst.rdata_dport(rdata_dport);
    dut_inst.ack_dport(ack_dport);


    // Traces of Signals
    dut_inst.Trace(tf, pn_cfg_vcd_level); // Trace signals of the DUT
    // traces of local signals here

    uint32_t clock_frequency = (1.0/(PERIOD_NS*1e-9));

    // create peripheral for simulator:
    std::unique_ptr<c_soft_graphics> graphics =
std::make_unique<c_soft_graphics> (
        CFG_GRAPHICS_BASE_ADDRESS,
        c_soft_graphics::ResolutionMode::Mode_80x60,
        c_soft_graphics::ColorMode::Mode_RGB888,
        clock_frequency,
        flush_to_gui
```

```
    );

    // start simulation
    sc_start();
    PN_INFO("\n\t\t*****Simulation started*****");


    // ************** Add testbench code here ****************

    // add the peripheral to the DUT
    dut_inst.add_peripheral(CFG_GRAPHICS_BASE_ADDRESS,
std::move(graphics));
    dut_inst.list_all_peripherals();

    // search for peripheral in the list of peripherals
    c_soft_peripheral *found_peripheral =
dut_inst.find_peripheral(CFG_GRAPHICS_BASE_ADDRESS);
    if (found_peripheral)
    {
        // cast the found peripheral to graphics object
        c_soft_graphics *graphics_ptr = dynamic_cast<c_soft_graphics
*>(found_peripheral);
        if (graphics_ptr)
        {
            // If the peripheral is correctly identified as a graphics
object, dump its contents
            PN_INFOF(("Found graphics peripheral at address 0x%llx.",
CFG_GRAPHICS_BASE_ADDRESS));
        }
        else
        {
            //std::cerr << "Found peripheral is not a graphics object."
<< std::endl;
            PN_ERROR("Found peripheral is not a graphics object.");
        }
    }
    else
    {
```

```
        PN_ERRORF(("No graphics peripheral found at address 0x%llx.",
CFG_GRAPHICS_BASE_ADDRESS));
    }

    // *************** Testbench setup ***************

    uint32_t width = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::WIDTH, 0xF, true);
    uint32_t height = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::HEIGHT, 0xF, true);
    uint32_t framebuffer_size = width * height;

    PN_INFO("");
    PN_INFO("==== Graphics Peripheral Parameters ====");
    PN_INFOF(("Framebuffer Width  : %d px", width));
    PN_INFOF(("Framebuffer Height : %d px", height));
    PN_INFOF(("Framebuffer Size   : %d pixels", framebuffer_size));
    PN_INFOF(("Framebuffer Size   : 0x%zx pixels (hex)",
framebuffer_size));
    PN_INFO("=======================================");
    PN_INFO("");
    PN_INFO(" *************** Running Graphics tests *************** ");
    PN_INFO("");

    // write first pixel of framebuffer
    PN_INFO("Test: write first pixel of framebuffer");
    write_request(0x00FFFFFF, CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::FRAMEBUFFER, 0xF);
    uint32_t data_after_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::FRAMEBUFFER, 0xF, true);
    PN_ASSERTM(data_after_write == 0x00FFFFFF, "First pixel not written
to framebuffer.");

    // write last pixel of framebuffer
    PN_INFO("Test: write last pixel of framebuffer");
    write_request(0x00123456, CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::FRAMEBUFFER + (framebuffer_size - 1) * 4, 0xF); // final
pixel in framebuffer
```

```
    data_after_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::FRAMEBUFFER + (framebuffer_size - 1) * 4, 0xF, true);
    PN_ASSERTM(data_after_write == 0x00123456, "Last pixel not written to
framebuffer.");

    // write to registers
    // control
    PN_INFO("Test: write to registers");
    uint32_t data_before_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::CONTROL, 0xF, true);
    write_request(0x00000001, 0x40000000, 0xF); // write control register
- warning expected
    data_after_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::CONTROL, 0xF, true);
    PN_ASSERTM(data_before_write == data_after_write, "Write to control
register was possible. Should not be. Register is currently read only.");

    // color mode
    data_before_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::COLOR_MODE, 0xF, true);
    write_request(50, CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::COLOR_MODE, 0xF); // invalid color mode
    data_after_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::COLOR_MODE, 0xF, true);
    PN_ASSERTM(data_before_write == data_after_write, "Write '50' to
color mode register was possible. Should not be. '50' is not a valid
color mode.");

    write_request(0, CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::COLOR_MODE, 0xF); // invalid color mode
    data_after_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::COLOR_MODE, 0xF, true);
    PN_ASSERTM(data_after_write == 0, "Write '0' to color mode register
was not possible. Should be. '0' is a valid color mode.");

    // resolution mode
    data_before_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::RESOLUTION_MODE, 0xF, true);
```

```
    write_request(50, CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::RESOLUTION_MODE, 0xF); // invalid resolution mode
    data_after_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::RESOLUTION_MODE, 0xF, true);
    PN_ASSERTM(data_before_write == data_after_write, "Write '50' to
resolution mode register was possible. Should not be. '50' is not a valid
resolution mode.");

    // resolution mode
    write_request(0x00000000, CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::RESOLUTION_MODE, 0xF);
    data_after_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::RESOLUTION_MODE, 0xF, true);
    PN_ASSERTM(data_after_write == 0x00000000, "Write '0x00000000' to
resolution mode register was not possible. Should be. '0x00000000' is a
valid resolution mode.");

    // resolution mode
    write_request(0x00000001, CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::RESOLUTION_MODE, 0xF);
    data_after_write = read_request(CFG_GRAPHICS_BASE_ADDRESS +
c_soft_graphics::RESOLUTION_MODE, 0xF, true);
    PN_ASSERTM(data_after_write == 0x00000001, "Write '0x00000001' to
resolution mode register was not possible. Should be. '0x00000001' is a
valid resolution mode.");

    // wait for flush callback
    PN_ASSERTM(flush_triggered_times == 0, "Flush callback triggered too
early");
    run_cycle(1700000);
    PN_ASSERTM(flush_triggered_times >= 1, "Flush callback trigger
missing (first time).");
    run_cycle(1700000);
    PN_ASSERTM(flush_triggered_times >= 2, "Flush callback trigger
missing (second time)");

    PN_INFO("");
    PN_INFO(" *************** Finished Graphics tests successfully
```

```
********** \n");
    return 0;
}
```

6.2. gui-Klasse

6.2.1. gui.h

```
/**
 * @class gui
 * @author Konrad Armbrecht
 *
 * This module provides a Qt-based GUI that is available to the piconut
simulator.
 * Currently implemented are functions for the image output of the
graphics peripheral.
 * The simulator terminates when the GUI window is closed.
 */

#ifndef GUI_H
#define GUI_H

#include "mainwindow.h"
#include <QApplication>

class gui
{

public:
    /**
     * @brief Constructor. Initializes the GUI and creates a MainWindow
instance.
     */
    gui(int& argc, char** argv);


    /**
     * @brief Destructor.
```

```
     */
    virtual ~gui();

    /**
     * @brief Displays the image from the framebuffer in the GUI by
instructing the
     * mainwindow class accordingly
     * @param framebuffer Pointer to the framebuffer data.
     * @param framebuffer_size Size of the framebuffer in pixel.
     *
     */
    void set_image(uint32_t* framebuffer, uint32_t framebuffer_size);

    /**
     * @brief Checks if the main window is currently open. Used for
terminating
     * the simulation when closing the GUI.
     * @return True if the main window is open, otherwise false.
     */
    bool is_gui_window_open() const;

    void processEvents();

    void quit();

private:
    int argc = 0;
    char** argv = nullptr;
    QApplication* app;
    MainWindow* mwindow; ///< Pointer to the main window instance.
};
#endif // GUI_H
```

6.2.2. gui.cpp

```
#include <QApplication>
#include "gui.h"
#include "framebuffer_to_image.h"
```

```
gui::gui(int& argc, char** argv)
    : app{new QApplication(argc, argv)}
{
    app->setApplicationName("PicoNut Simulator");

    mwindow = new MainWindow();
    mwindow->show();
}

gui::~gui()
{
}

// Gets called from top_tb for framebuffer flush
void gui::set_image(uint32_t* framebuffer, uint32_t framebuffer_size)
{
    if(mwindow == nullptr)
    {
        return;
    }

    mwindow->image_to_gui(framebuffer, framebuffer_size);
}

void gui::processEvents()
{
    app->processEvents();
}

void gui::quit()
{
    mwindow->close();
    app->quit();
}

bool gui::is_gui_window_open() const
{
```

```
    if(mwindow == nullptr)
    {
        return false;
    }

    return mwindow->is_mainwindow_open();
}
```

6.2.3. mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QLabel>
#include <QResizeEvent>

QT_BEGIN_NAMESPACE
namespace Ui {
class MainWindow;
}
QT_END_NAMESPACE

/**
 * @class mainwindow
 * @author Konrad Armbrecht
 *
 * This class provides a Qt-based GUI window that displays the assembled
output image.
 * It is responsible for filling the QLabel with the output image.
 *
 * The GUI adapts dynamically to changes in window size while preserving
the
 * correct aspect ratio. Additionally, it allows for simulation control,
as
 * the simulator is terminated when the main window is closed.
 */
class MainWindow : public QMainWindow
{
```

```
    Q_OBJECT

public:
    /**
     * @brief Constructor.
     * @param parent The parent widget (default: nullptr).
     *
     * Initializes the UI components and sets up the image display area.
     */
    MainWindow(QWidget* parent = nullptr);

    /**
     * @brief Destructor.
     */
    virtual ~MainWindow();

    /**
     * @brief Updates the GUI with the latest framebuffer content.
     * @param framebuffer Pointer to the framebuffer data.
     * @param framebuffer_size Size of the framebuffer in pixel.
     *
     * Converts the framebuffer content into a `QPixmap` and displays it
     * in the GUI. The image adapts to the QLabel size dynamically.
     */
    void image_to_gui(uint32_t* framebuffer, uint32_t framebuffer_size);

    /**
     * @brief Check if the main window is open.
     * @return True if the main window is open, otherwise false.
     */
    bool is_mainwindow_open() const;

protected:
    /**
     * @brief Handles window resize events.
     * @param event The resize event.
     *
     * Ensures that the window maintains the correct aspect ratio based
```

```
     * on the framebuffer resolution.
     */
    void resizeEvent(QResizeEvent* event) override;

    void closeEvent(QCloseEvent *event) override;

private:
    Ui::MainWindow* ui;          ///< Pointer to the UI components.
    QImage output_image;         ///< Stores the converted framebuffer
image.
    bool mainwindow_open = true; ///< Tracks whether the main window is
open.
};
#endif // MAINWINDOW_H
```

6.2.4. mainwindow.cpp

```
#include <piconut-config.h>
#include <iostream>

#include "mainwindow.h"
#include "mainwindow_ui.h"
#include "framebuffer_to_image.h"

#include <QLabel>
#include <QImage>
#include <QVBoxLayout>
#include <QWidget>
#include <QDir>

bool initial_resized = false;

MainWindow::MainWindow(QWidget* parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
    , output_image{1000, 1000, QImage::Format_RGB32}
{
    ui->setupUi(this);
    // ui->image_label->setMinimumSize(CFG_GRAPHICS_RESOLUTION_WIDTH,
```

```
CFG_GRAPHICS_RESOLUTION_HEIGHT); // wird in Z. 41 gesetzt
}

MainWindow::~MainWindow()
{
}

void MainWindow::image_to_gui(uint32_t* framebuffer, uint32_t
framebuffer_size)
{
    // Image scales with the size of the QLabel (when the window size is
changed)
    ui->image_label->setScaledContents(true);

    if(framebuffer == nullptr)
    {
        qWarning() << "Framebuffer is null!";
        return;
    }

    // Convert framebuffer to output image
    framebuffer_to_image fb_to_image(framebuffer, framebuffer_size);
    output_image = fb_to_image.output_image();

    ui->image_label->setMinimumSize(output_image.width(),
output_image.height());

    if(output_image.isNull())
    {
        qWarning() << "Output image is null!";
        return;
    }

    // Convert image to pixmap in order to display image
    ui->image_label->setPixmap(QPixmap::fromImage(output_image));

    //adapt MainWindow size to 800 x 573 as "default" size
    if(!initial_resized)
```

```
    {
        ui->image_label->setSizePolicy(QSizePolicy::Expanding,
QSizePolicy::Expanding);

        ui->image_label->resize(900, 500);
        //resize(ui->image_label->sizeHint());
        initial_resized = true;
    }

    setWindowTitle(QString("Image resolution: %1x%2 px | Label size:
%3x%4 px")
                       .arg(output_image.width())
                       .arg(output_image.height())
                       .arg(ui->image_label->width())
                       .arg(ui->image_label->height()));
}

// Override resize event handler to maintain the aspect ratio of the
window and its contents
void MainWindow::resizeEvent(QResizeEvent* event)
{
    QSize label_size = ui->image_label->size();
    int label_pixel_count = label_size.width() * label_size.height();
    QSize newSize = event->size();

    // Force the aspect ratio depending on the resolution
    int newWidth = newSize.width();
    int newHeight = static_cast<int>(newWidth * output_image.height() /
static_cast<float>(output_image.width()));
    resize(newWidth, newHeight);

    QMainWindow::resizeEvent(event);

    // setWindowTitle(QString("Image resolution: %1x%2 px | Label size:
%3x%4 px")
    //                    .arg(output_image.width())
    //                    .arg(output_image.height())
    //                    .arg(ui->image_label->width())
```

```
    //                    .arg(ui->image_label->height()));
}

void MainWindow::closeEvent(QCloseEvent *event)
{
    mainwindow_open = false;
    event->accept();
}


bool MainWindow::is_mainwindow_open() const
{
    return mainwindow_open;
}
```

6.2.5. framebuffer_to_image.h

```
#ifndef FRAMEBUFFERTOIMAGE_H
#define FRAMEBUFFERTOIMAGE_H
#include <QImage>
#include <QVector>

/**
 * @class framebuffer_to_image
 * @brief A class for converting framebuffer data into a Qt-compatible
QImage.
 * @author Konrad Armbrecht
 *
 * This class is responsible for converting the frame buffer content into
a Qt
 * displayable format. The class can be used in the simulation
environment.
 *
 * Within that class an empty QVector "qrgb_array" is created that will
be filled
 * with Qt readable pixel information. To fill the QVector, the
framebuffer content
 * is read address by address. The containing hex values are converted to
QRgb values
```

```
 * and copied into the QVector. The QRgb values are then used to assemble
the output
 * image, which is finally being returned.
 */
class framebuffer_to_image {
public:

    /**
     * @brief Constructor
     *
     * @param framebuffer Pointer to the framebuffer containing pixel
data.
     * @param framebuffer_size_in_px Size of the framebuffer in bytes.
     */
    framebuffer_to_image(uint32_t *framebuffer, uint32_t
framebuffer_size_in_px);

    /**
     * @brief Converts the raw framebuffer data into QRgb format and
creates a QImage from it.
     *
     * @return QImage containing the processed framebuffer data.
     */
    QImage output_image();

private:

    /**
     * @brief Initializes a 2D pixel array.
     *
     * @param width Width of the framebuffer in pixels.
     * @param height Height of the framebuffer in pixels.
     */
    void create_pixel_array(uint16_t width, uint16_t height);

    /**
     * @brief Converts framebuffer data into a QRgb array.
     */
```

```
    void framebuffer_to_vector();

    /**
     * @brief Converts the QRgb array into a QImage.
     */
    void array_to_image();

    uint32_t* framebufferptr;           ///< Pointer to the framebuffer
data.
    QVector<QVector<QRgb>> qrgb_array;  ///< 2D array storing converted
QRgb values.
    QImage output_qimage;               ///< The final output image.
    uint32_t framebuffer_width;
    uint32_t framebuffer_height;
};

#endif // FRAMEBUFFERTOIMAGE_H
```

6.2.6. framebuffer_to_image.cpp

```
#include "framebuffer_to_image.h"
#include <QDebug>
#include <QRgb>
#include <cstdint>
#include <piconut-config.h>
#include <iostream>

framebuffer_to_image::framebuffer_to_image(uint32_t *framebuffer,
uint32_t framebuffer_size_in_px)
{
    // Values hardcoded, because current implementation of graphics only
supports these resolution modes
    framebufferptr = framebuffer;
    if(framebuffer_size_in_px == 6)
    {
        framebuffer_width = 3;
        framebuffer_height = 2;
    }
    else if (framebuffer_size_in_px == 4800)
```

```
    {
        framebuffer_width = 80;
        framebuffer_height = 60;
    }
    else
    {
        qWarning() << "framebuffer_to_image: framebuffer size not
supported. Supported sizes are 4800 and 6";
    }

    output_qimage = QImage(framebuffer_width, framebuffer_height,
QImage::Format_RGB32);
    this->create_pixel_array(framebuffer_width, framebuffer_height);
}

void framebuffer_to_image::create_pixel_array(uint16_t width, uint16_t
height)
{
    //Colordepth will have no effect for now. Colordepth is always 24
bit, pixel colors can be all possible hex values
    //Create 2D-array with default color-value 1 (white) for all pixel
    qrgb_array = QVector<QVector<QRgb>>(height, QVector<QRgb>(width, 1));
}

void framebuffer_to_image::framebuffer_to_vector()
{

    if(framebufferptr == nullptr)
    {
        qDebug() << "framebuffer_to_image.cpp: framebuffer is empty";
        return;
    }
    // Iterate through the array and convert the values
    for (int y = 0; y < framebuffer_height; ++y) {
        for (int x = 0; x < framebuffer_width; ++x) {
            // Get the current pixel value from the one-dimensional
framebuffer array
            uint32_t hexValue = framebufferptr[y * framebuffer_width +
```

```
x];

            // Split hex value into red, green and blue components
            int red = (hexValue >> 16) & 0xFF;  // Upper 8 Bits
            int green = (hexValue >> 8) & 0xFF; // Middle 8 Bits
            int blue = hexValue & 0xFF;         // Lower 8 Bits

            // Convert hex value to QRgb
            QRgb qRgbValue = qRgb(red, green, blue);

            // Insert into the two-dimensional QVector
            qrgb_array[y][x] = qRgbValue;
        }
    }
}

void framebuffer_to_image::array_to_image()
{
    for (int y = 0; y < framebuffer_height; ++y) {
        for (int x = 0; x < framebuffer_width; ++x) {

            // Set pixel color from given array position
            output_qimage.setPixel(x, y, qrgb_array[y][x]);
        }
    }
}

QImage framebuffer_to_image::output_image()
{
    framebuffer_to_vector();
    array_to_image();
    return output_qimage;
}
```

6.2.7. mainwindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
```

```
<widget class="QMainWindow" name="MainWindow">
 <property name="geometry">
  <rect>
   <x>0</x>
   <y>0</y>
   <width>800</width>
   <height>600</height>
  </rect>
 </property>
 <property name="windowTitle">
  <string>MainWindow</string>
 </property>
 <widget class="QWidget" name="centralwidget">
  <layout class="QGridLayout" name="gridLayout">
   <item row="0" column="0">
    <widget class="QLabel" name="image_label">
     <property name="text">
      <string>TextLabel</string>
     </property>
    </widget>
   </item>
  </layout>
 </widget>
 <widget class="QMenuBar" name="menubar">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>800</width>
    <height>23</height>
   </rect>
  </property>
 </widget>
 <widget class="QStatusBar" name="statusbar"/>
</widget>
<resources/>
<connections/>
</ui>
```

6.2.8. gui_tb.cpp

```
#include <base.h>
#include <gui.h>
#include <systemc.h>

int sc_main(int argc, char** argv)
{
    int argc2 = 0;
    gui gui_inst(argc2, nullptr);

    // GUI will be opened for a few seconds and terminate.
    PN_INFO("Test: GUI window");
    for(int i = 0; i < 1000000; i++)
    {
        gui_inst.processEvents();
    }

    PN_INFO("Test: successfully completed");

    return 0;
}
```

6.3. refdesign_c_soft_graphics

6.3.1. top_tb

```
#include <stdint.h>
#include <systemc.h>
#include <piconut-config.h>

#include "top.h"
#include <c_soft_uart.h>
#include <c_soft_graphics.h>
#include <gui.h>

//----------------------------------------

#define PERIOD_NS 10.0
```

```
#define debug

// initialize TB signals
sc_signal<bool> PN_NAME(clk);
sc_signal<bool> PN_NAME(reset);

// Setup GUI
gui* gui_window = nullptr;

void run_cycle(int cycles = 1)
{
    for(int i = 0; i < cycles; i++)
    {
        clk = 0;
        sc_start(PERIOD_NS / 2, SC_NS);
        clk = 1;
        sc_start(PERIOD_NS / 2, SC_NS);
    }
}

// Sends framebuffer data to GUI to display it
void flush_to_gui(uint32_t* framebuffer, uint32_t framebuffer_size_in_px)
{
    gui_window->set_image(framebuffer, framebuffer_size_in_px);
}

int sc_main(int argc, char** argv)
{
    pn_parse_enable_trace_core = 1;                  // enable core
trace dump
    pn_cfg_enable_application_path = 1;              // enable
application path in program args
    PN_PARSE_CMD_ARGS(argc, argv);                   // parse command
line arguments
    sc_trace_file* tf = PN_BEGIN_TRACE("piconut_tb"); // create trace
file

    // Initialize GUI window
```

```
    int argc2 = 0;
    gui_window = new gui(argc2, nullptr);

    m_top dut_inst{"dut_inst"}; // Initialize the Design under Testing
(DUT), this is the Design name needed by the svc_too

    // Connect the signals
    dut_inst.clk(clk);
    dut_inst.reset(reset);

    uint32_t clock_frequency = (1.0 / (PERIOD_NS * 1e-9));

    // create peripheral for simulator:
    std::unique_ptr<c_soft_graphics> graphics =
std::make_unique<c_soft_graphics>(
        CFG_GRAPHICS_BASE_ADDRESS,
        c_soft_graphics::ResolutionMode::Mode_80x60,
        c_soft_graphics::ColorMode::Mode_RGB888,
        clock_frequency,
        flush_to_gui);

    dut_inst.piconut->simmemu->add_peripheral(CFG_GRAPHICS_BASE_ADDRESS,
std::move(graphics));

    std::unique_ptr<c_soft_uart> uart =
std::make_unique<c_soft_uart>(0x22, CFG_WB_UART_BASE_ADDRESS);
    dut_inst.piconut->simmemu->add_peripheral(CFG_WB_UART_BASE_ADDRESS,
std::move(uart));

    // connects signals from TOP to TB
    dut_inst.piconut->simmemu->load_elf(pn_cfg_application_path);
    dut_inst.piconut->simmemu->list_all_peripherals();

    // Traces of Signals
    dut_inst.Trace(tf, pn_cfg_vcd_level); // Trace signals of the DUT
    // traces of local signals here

    sc_start(SC_ZERO_TIME); // start simulation
```

```
    PN_INFO("\n\t\t*****Simulation started*****");

    // Testbench code here
    reset = 1;
    run_cycle(); // end with a wait
    reset = 0;
    run_cycle(2);

    while(dut_inst.piconut->state_is_not_halt() && gui_window-
>is_gui_window_open())
    {
        gui_window->processEvents();
        run_cycle();
    }

    PN_END_TRACE();
    PN_INFO("\n\t\t*****Simulation complete*****");

    return 0;
}
```

6.4. graphics_fancy_random_squares-Software

6.4.1. graphics_fancy_random_squares.c

```
// Execute this file with 'PN_SYSTEM=refdesign_c_soft_graphics make sim'

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>

// Function to create a randomly sized and colored square for the
framebuffer
uint32_t* create_square(int width, int height, int total_pixels)
{
    uint32_t* pixel_array = (uint32_t*)malloc(total_pixels *
```

```
sizeof(uint32_t));
    if (!pixel_array) {
        perror("Failed to allocate memory for framebuffer");
        return NULL;
    }

    // Initialize the array with black (set all pixels to black)
    for (int i = 0; i < total_pixels; ++i) {
        pixel_array[i] = 0x00000000;  // Black
    }

    // Calculate a random square size between 2x2 and heightxheight
    int square_size = rand() % (height - 1) + 2;  // Random size between
2 and height

    // Calculate a random position for the square
    int start_x = rand() % (width - square_size);  // Random x-coordinate
    int start_y = rand() % (height - square_size); // Random y-coordinate

    // Generate a random color value (32-bit ARGB)
    uint32_t random_color = (rand() % 256) << 16 |    // Red value (R)
                            (rand() % 256) << 8  |    // Green value (G)
                            (rand() % 256);           // Blue value (B)

    // Set the pixels of the square to the random color
    for (int y = start_y; y < start_y + square_size; ++y) {
        for (int x = start_x; x < start_x + square_size; ++x) {
            if (x < width && y < height) {  // Ensure the square is not
written outside the array
                pixel_array[y * width + x] = random_color; // Set the
pixel to the random color
            }
        }
    }

    return pixel_array;
}
```

```
int main() {

    printf("######### Starting graphics_fancy_random_squares
#########\n\n");

    // Initialize the random number generator
    srand(6758);

    // Pointer to the graphics module
    volatile uint32_t* graphics = (uint32_t*)0x40000000;

    // Read the framebuffer size
    int width = *(graphics + 0x1);
    int height = *(graphics + 0x2);
    int framebuffersize = width * height;

    // Is intended as a while loop, but implemented as a for loop for
Jenkins test
    for (int i = 0; i < 10; ++i)
    {
        // Create a randomly sized and colored square for the framebuffer
        uint32_t* square = create_square(width, height, framebuffersize);

        // Write array, filled with the square, to the graphics
framebuffer
        for (int i = 0; i < framebuffersize; i++) {
            *(graphics + 0x7 + i) = square[i];
        }

        // Free the memory
        free(square);
    }

    printf("\n######### Terminating graphics_fancy_random_squares
#########\n\n");

    return 0;
```

```
}
```