INTERNSHIP REPORT

DESIGN OF RADAR DISPLAY USING Qt FRAMEWORK

A report submitted in partial fulfilment of the requirements for

INTERNSHIP

By

DIVYA L

Department of Electronics and Communication Engineering

PRINCE SHRI VENKATESHWARA PADMAVATHY ENGINEERING

COLLEGE, PONMAR, CHENNAI

Under the Supervision of

Dr P. VENKATARAMANA, Scientist/Engineer-SG

Satish Dhawan Space Centre (SDSC)

Indian Space Research Organisation (ISRO)

SRIHARIKOTA

DECLARATION

I hereby declare that the project titled *"Design of Radar Display using Qt Framework"* has been carried out at Sathish Dhawan Space Centre SHAR, Indian Space Research Organisation (ISRO) as part of my technical work. This project is the result of my independent effort and has not been submitted elsewhere for any academic, professional, or institutional purpose. All the work presented in this report is original and completed with sincerity and integrity.

The project was executed under the supervision of Dr P. VENKATARAMANA, Scientist/Engineer-SG, who oversaw its progress and completion. I have independently developed and executed all parts of the project. Data, results, and findings included in the report are genuine and based on actual work performed during the project period. Any external content referenced has been appropriately cited.

Divya L

ACKNOWLEDGMENT

I would like to express my sincere gratitude to all those who supported and guided me during the course of my project titled *"Design of Radar Display using Qt Framework"* at the Indian Space Research Organisation (ISRO). This internship has been a valuable learning experience, and I am truly grateful for the opportunity to be given in such a prestigious organization.

I am especially thankful to my project guide, to Dr **P. Venkataramana,** Sceintist/Engineer -SG, for his expert guidance, technical inputs, and continuous support throughout the internship.

I also extend my heartfelt gratitude **Mr. V. Janarthanan**, Sceintist/Engineer -SF, for his support and encouragement.

Additionally, I wish to recognize the entire MFPCR Staff for their assistance, which made my time at at Sathish Dhawan Space Centre SHAR both productive and enriching.

My sincere thanks to **Mr. M.M.V. Dhanunjaya Rao**, General Manager, RTS, RO, for facilitating my internship and ensuring a smooth onboarding process.

Furthermore, I am immensely grateful to **Dr.K.K.Senthilkumar**, Associate Professor, Prince Shri Venkateshwara Padmavathy Engineering college, Department of ECE for providing the guidance and opportunity to undertake this project.

This experience has been both professionally and personally enriching, and I am deeply appreciative of everyone who contributed to my journey.

CERTIFICATE

This is to certify that the project work entitled "DESIGN OF RADAR DISPLAY USING Qt FRAMEWORK" being submitted by the DIVYA L, bearing Roll Number 411722106025, in partial fulfilment of the requirements for the project, is a record Bonafide work at ISRO carried out under the supervision of Mr. Venkataramana. P, Deputy Manager (MFPCR/RO). The work presented in this report has been carried out by the student under the assigned guide's supervision during the period from **1ST July 2025 to 31st July 2025.**

We wish him all the been in her future endeavors.

Dr. P. Venkataramana
Deputy Manager
MFPCR/RTS/RO
(Supervisor)

M.M.V Dhanumjaya Rao
General Manager
RTS/RO

Contents

CHAPTER 1

ABOUT THE ORGANIZATION

1.1 About SHAR (Satish Dhawan Space Centre)

The Satish Dhawan Space Centre (SDSC), commonly known as SHAR, is the primary spaceport of the

Indian Space Research Organisation (ISRO), located at Sriharikota, a barrier island on the east coast of India in Andhra Pradesh. It is named after Dr. Satish Dhawan, a former chairman of ISRO and a visionary in Indian aerospace engineering.

SDSC SHAR plays a crucial role in launching satellites into orbit using ISRO's fleet of launch vehicles, including the PSLV (Polar Satellite Launch Vehicle), GSLV (Geosynchronous Satellite Launch Vehicle), and LVM3. The centre is equipped with multiple launch pads, vehicle assembly buildings, a solid propellant production plant, telemetry and tracking facilities, and a mission control centre. SHAR's strategic location near the equator provides an advantage for launching geostationary and other types of satellites efficiently. It is one of the most active launch sites in Asia and continues to support India's rapidly growing space program, including satellite deployment, human spaceflight preparation, and interplanetary missions.

The centre not only symbolizes India's self-reliance in space technology but also supports the global space community through commercial launches facilitated by ISRO's commercial arm, NSIL.

1.1.1 Key Facilities at SHAR

- Launch Pads: SHAR has two fully operational launch pads – the First Launch Pad (FLP) and the Second Launch Pad (SLP). These platforms are used for vertical integration and launching of PSLV, GSLV, and LVM3 rockets. A third pad is under consideration to support India's human spaceflight program, Gaganyaan.

- Vehicle Assembly Building (VAB): The VAB is a massive structure where rocket stages and payloads are vertically integrated under controlled conditions. It enables precise alignment and assembly of complex multi-stage launch vehicles.

- Solid Propellant Space Booster Plant (SPROB): This facility is responsible for the production and processing of large solid rocket boosters used in vehicles like PSLV and LVM3. It includes casting, curing, and inspection units to ensure high reliability.

- Mission Control Centre (MCC): The MCC monitors and controls launch operations in realtime. It is equipped with advanced tracking, telemetry, and flight safety systems. Engineers and scientists coordinate from here to manage the mission timeline and vehicle trajectory.

Range Operations and Tracking Systems: These systems consist of radar, telemetry stations, and communication links that track the launch vehicle throughout its flight path. They ensure safety, provide real-time data, and support post-launch analysis. Environmental Test Facilities: SDSC also includes facilities for simulating space conditions such as vibration, thermal vacuum, and acoustic tests to ensure satellite and launcher robustness.

Together, these facilities make SHAR one of the most advanced launch complexes in the world, contributing significantly to India's emergence as a key spacefaring nation.

1.2 Range Operations and Tracking Systems

The Range Operations and Tracking Systems at SHAR form the backbone of mission surveillance and flight safety. These systems are responsible for the real-time monitoring, tracking, telemetry acquisition, and flight safety assessment of launch vehicles from liftoff through all critical mission phases. The range consists of a network of radars, electro-optical tracking systems, and telemetry ground stations located both at SHAR and across remote sites

strategically positioned along the vehicle's flight path. Primary tracking is carried out by monostatic and bistatic C-Band and S-Band radars, which provide highly accurate position and velocity data of the rocket in flight. These data are critical for both real-time monitoring and post-flight trajectory analysis. Telemetry systems continuously receive data transmitted by onboard instruments related to vehicle performance, system health, and environmental parameters. This stream is used by mission control and flight safety officials to make instantaneous decisions during launch. The Flight Safety System is integrated with destruct mechanisms that can safely neutralize the vehicle mid-air in the rare case of deviation from the intended flight path or a threat to public safety. This entire range network is managed from the Range Control Centre, where a team of experts continuously monitors compliance with pre-defined safety corridors and mission objectives.

In addition to active tracking, SHAR's range facilities support simulations, rehearsal trials, and countdown procedures, ensuring all systems are tested thoroughly before actual launch. Redundancy in communication and tracking infrastructure ensures high reliability and precision, which is vital for the success of each mission.

The robust coordination between telemetry, radar, tracking stations, and the Mission Control Centre ensures a comprehensive and secure launch environment, significantly contributing to ISRO's excellent record of successful missions.

1.2.1 Radar Tracking

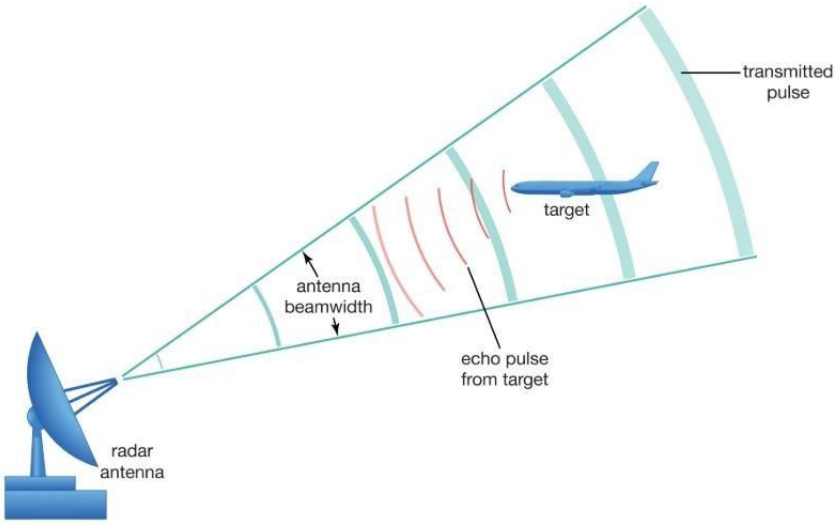

Figure 1: Radar Tracking.

Radar tracking is a critical component of the range operations infrastructure at SHAR. During every launch, high-precision radar systems are deployed to continuously monitor the flight path of the launch vehicle. These radars operate primarily in the C-band and S-band frequencies, offering high accuracy in range and angular resolution.

The radars calculate real-time trajectory parameters such as altitude, velocity, azimuth, elevation, and range rate of the vehicle by sending microwave pulses and analysing the time delay and Doppler shift of the returned signal. The trajectory data acquired from multiple radar stations is fused and processed at the Range Control Centre, where it is used for both mission monitoring and flight safety assessment.

These radar systems are linked to auto-tracking antennas with servo-controlled mechanisms to maintain continuous lock on fast-moving targets. They are capable of tracking the launch vehicle from liftoff to the edge of radar visibility, typically extending hundreds of kilometres downrange.

In the event of any deviation from the nominal flight path, radar-derived data plays a pivotal role in triggering safety mechanisms such as mission hold or flight termination. Due to their reliability and precision, radars are among the first-line instruments used during all launch rehearsals, simulations, and the actual countdown sequence.

Overall, the radar tracking infrastructure at SHAR ensures precise, real-time monitoring of vehicle dynamics, forming an essential pillar of India's space launch capabilities.

1.2.2 Multi-Functional Pulse Compression Radar (MFPCR):

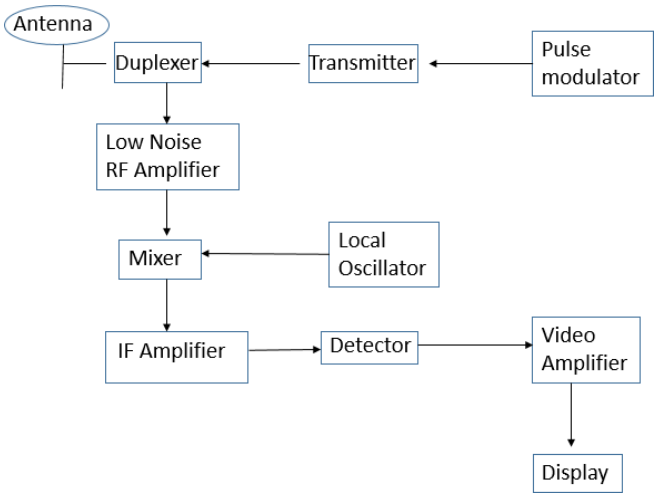

Figure 2: Pulse Compression Radar Block Diagram.

The Multi-Functional Pulse Compression Radar (MFPCR) is one of the most advanced radar systems employed at SHAR for real-time tracking and range safety. It is designed to serve multiple roles, including trajectory tracking, velocity measurement, flight safety assessment, and support for launch rehearsals and simulations.

This radar system operates in the S-band frequency range and uses sophisticated pulse com pression techniques to achieve both high resolution and long-range detection. Unlike conventional radar systems that face a trade-off between range and resolution, MFPCR leverages chirped pulses and matched filtering to compress the pulse width without sacrificing energy, thus achieving high time resolution and accurate target detection.

MFPCR features multiple modes of operation, such as automatic target tracking (ATT), manual tracking, and simulation mode for pre-launch testing. In ATT mode, the radar autonomously locks onto the target (launch vehicle) and maintains continuous tracking throughout the visible trajectory. This auto-tracking is supported by high-speed servo systems and a high-gain parabolic antenna capable of rapid angular adjustments.

A key advantage of MFPCR is its ability to simultaneously perform tracking and telemetry tasks. It can interface with telemetry receivers and optical systems, offering integrated situational awareness during critical phases like stage separation, velocity cut-off, or anomaly detection.

The radar system is highly modular and scalable, with built-in redundancy for mission critical components. Its data is relayed in real-time to the Range Control Centre, contributing to safety decisions and trajectory corrections when required. The use of MFPCR significantly enhances ISRO’s ability to monitor complex launch profiles, including those involving high thrust vehicles and multi-stage separation events.

Overall, the MFPCR exemplifies cutting-edge radar technology in aerospace operations, ensuring precision tracking and robust flight safety for India’s space missions.

CHAPTER 2

INTRODUCTION

2.1 About the Project

Radar systems are widely used in applications like air traffic control, weather forecasting, military surveillance, and autonomous vehicles to detect and track objects using radio waves. Typically, radar data is visualized on specialized screens that display the position, movement, and intensity of targets in a rotating, circular format known as a **Plan Position Indicator (PPI).** Building a digital radar display requires not only signal processing capabilities but also a responsive and interactive **Graphical User Interface (GUI)** to represent radar sweeps and detected targets.

The modern approach to building such systems involves combining signal handling with rich user interface development environments. One such tool is **Qt,** an open-source framework that supports cross-platform C++ development. Qt is especially suited for GUI-intensive applications due to its **QGraphicsView, QTimer,** and layout tools which allow smooth animations and custom visualizations like radar screens, rotating sweeps, and dynamically updating objects. Qt Designer also provides a way to create and test GUI layouts with drag-and-drop ease, accelerating development.

This project is a **simulation-based radar GUI—it does not rely on real radar sensor inputs. Instead, the system generates artificial data**, such as the coordinates of targets, to simulate how a radar scans an area and detects moving or stationary objects. The simulation includes a rotating radar beam, random object generation, and display updates mimicking real-time behavior. This approach allows testing and visualization of radar concepts in software without expensive hardware or signal acquisition tools.

By using Qt and simulated inputs, this project serves as a practical introduction to radar interfaces, real-time graphics, and user interaction design.

2.2 Radar System Overview

Radar (Radio Detection and Ranging) systems operate by transmitting radio waves and detecting the reflections from objects in their path. These reflections help determine an object's distance (range), direction (azimuth), and motion (velocity).

In this project, instead of using real-world radar hardware, the system simulates radar detection using artificial data generation techniques. Targets are randomly placed and tracked as the radar beam (sweep) rotates continuously in a circular fashion, replicating a Plan Position Indicator (PPI). This allows easy testing of tracking logic and GUI rendering without the need for physical radar components.

2.3 Simulation Logic and Architecture

The simulation-based radar GUI is designed to mimic the functional aspects of a real radar system, without relying on actual signal acquisition hardware. The architecture is modular and designed using the Qt framework, which provides a clean separation between the GUI and the radar logic.

2.3.1 Architecture

The radar GUI system is built using Qt's widget-based architecture. The major components include:

- **RadarWidget:** A custom widget that handles radar sweep rendering using QPainter.
- **MainWindow:** The container widget with control buttons and labels for displaying simulated data.
- **QTimer:** Drives the radar sweep rotation.
- **Random Target Generator:** Simulates incoming target coordinates.

The radar display consists of:

- A rotating sweep line (like a real radar)
- A static background of concentric circles and crosshairs
- Optional targets displayed at variable locations

This is all achieved without any real signal acquisition, making it an excellent simulation platform

2.3.2 System Components

The key components of the simulation are:

- **RadarWidget:** This is a custom widget derived from Qt's QWidget class. It is responsible for rendering the radar display, including the background grid, concentric circles, crosshairs, and the rotating sweep line. It also draws simulated targets on the radar screen.
- **QTimer:** The sweep line rotation is animated using a QTimer. It triggers the updateSweep() function at a regular interval (e.g., every 100 milliseconds), causing the widget to repaint with an updated sweep angle. This simulates the continuous circular motion of real radar sweeps.

- **MainWindow:** The MainWindow serves as the top-level application window. It contains the control buttons (Start, Stop, etc.), display panels (e.g., range, azimuth, velocity labels), and integrates the RadarWidget.

- **Target Simulation Module**: Since the project doesn't use physical hardware, simulated targets are generated using random coordinates and parameters. These targets are displayed on the radar screen whenever the sweep line passes over them, emulating real-time object detection.

2.3.3 Simulation Workflow

Initialization:
When the application starts, the graphical user interface (GUI) is set up using Qt's widget system. The radar screen, target display area, and control panel are all initialized.

Radar Sweep Activation:
Pressing the **Start** button begins the radar sweep using a QTimer. This timer triggers periodic repaint() events, causing the radar sweep line to rotate in real time.

Target Simulation:
Targets are either randomly placed or periodically updated with simulated data such as azimuth, elevation, and coordinates. These are rendered within the radar area.

Target Detection:
As the sweep line rotates, it checks for intersections with simulated targets. When a match is detected, the target is briefly highlighted and its details (range, azimuth, elevation, angle, direction, etc.) are displayed in **Target Panel 1** and **Target Panel 2**.

User Interaction:
Users can also click on a target directly to view detailed information, allowing for real-time analysis and interaction.

Stop Functionality:
Pressing the **Stop** button halts the radar sweep by stopping the timer and suspends further target detection or UI updates.

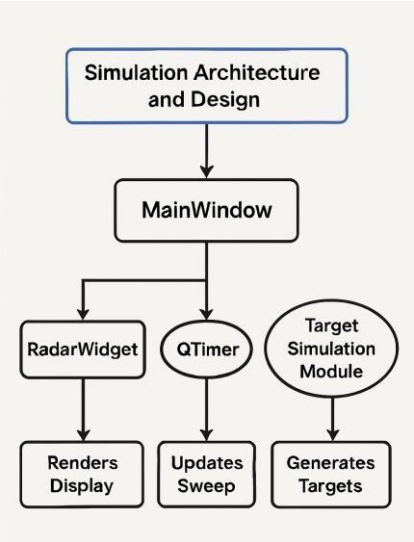

Figure 3 workflow of the GUI design

2.4 Radar Display Logic and GUI Components

The radar simulation project focuses on real-time visualization of radar sweeps and simulated target detection using a custom-built GUI. The interface is designed to mimic the functional appearance of a traditional radar Plan Position Indicator (PPI), allowing users to start and stop the sweep and observe target parameters like range, azimuth, and velocity.

2.4.1 GUI Layout and Interaction

The following image illustrates the graphical interface of the simulation-based radar system. The design mimics a traditional Plan Position Indicator (PPI) commonly used in real radar displays.

This interface was created using Qt Designer and C++ backend code. It includes:

- A rotating sweep line that mimics the radar's scanning motion
- A set of concentric range circles and crosshairs to aid in visual positioning
- A control panel with Start and Stop buttons to manage the simulation
- A target panel showing live updates of simulated target parameters (range, azimuth, and velocity)

This display provides a real-time visual simulation of how a radar would detect and display object information within its detection zone, even though the data here is generated synthetically.

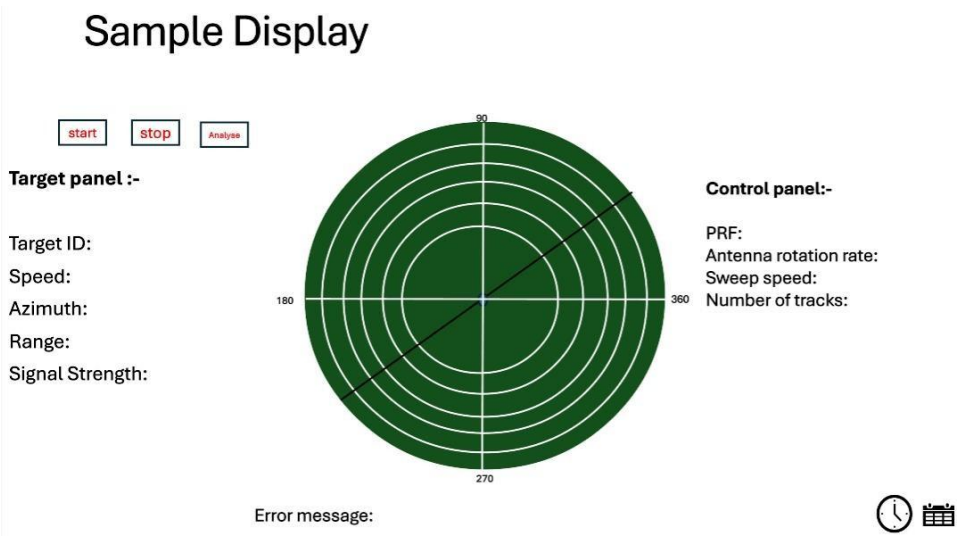

Figure 4 Sample Radar Display

2.4.2 Core Logic for Radar Sweep

- The sweep line is drawn using QPainter and updated by a QTimer, which triggers a repaint every few milliseconds.
- As the angle of the sweep increases from 0° to 360°, it rotates clockwise, simulating radar motion.

- Targets are represented as dots or blips placed at certain coordinates. When the sweep line intersects with a target's position (based on angular threshold), the information panel is updated.

2.4.3 Data Simulation

Since the project does not use real hardware or sensors, target data is simulated:

- Target coordinates (x, y) are generated randomly within the radar's radius.
- Range is calculated using Euclidean distance from the center.
- Azimuth is derived from the angle of the coordinate.
- Velocity values are generated for realism but remain arbitrary.
- **RPM (Revolutions Per Minute)**
  This indicates how many full rotations the radar antenna makes per minute.
- **Sweep Speed (degrees/second)**
  Since one full rotation is **360 degrees**, the sweep speed is calculated as:

Sweep Speed = (RPM× 360)/(60×PRF)

The project has evolved from a basic static display to a more modular and interactive design. A control panel has been introduced to configure key parameters, and the target panel now displays target details in a generic format. Additionally, a second target panel (Target Panel B) has been developed to show specific target information when a target is selected from the radar. This structure improves clarity, usability, and paves the way for future integration with real radar input. This project also contains the real timing records of the target as well as the GUI display.

Chapter 3

Methodology

3.1 Tools and Technologies Used

In the development of the Radar GUI Simulator, various software tools and programming technologies were employed to ensure a smooth design and implementation process. Each tool played a specific role in building the graphical interface, managing the backend logic, and compiling the project. The following subsections provide an overview of the tools and technologies utilized.

3.1.1 Qt Creator

Qt Creator is an integrated development environment (IDE) widely used for developing cross-platform applications using the Qt application framework. It offers features such as code highlighting, automatic completion, real-time syntax checking, debugging support, and a user-friendly interface for managing project files. In this project, Qt Creator served as the primary environment for writing C++ code and designing the user interface components of the radar GUI.

3.1.2 Qt Framework

The Qt framework is a comprehensive C++ library that provides tools for creating user interfaces and managing application behaviour. It includes modules for GUI creation, event handling, timers, and widget manipulation. The radar simulator made extensive use of Qt's GUI and Core modules to render the rotating radar sweep, control animations, and manage user interactions.

3.1.3 Qt Designer

Qt Designer is a visual layout tool that is integrated with Qt Creator. It allows developers to construct and edit GUI components in a drag-and-drop manner. Instead of manually writing code for the user interface, the GUI of the radar simulator—including buttons, display areas, and layout arrangements—was designed using Qt Designer, enabling a more intuitive and visual development process.

3.1.6 GNU Compiler Collection (G++)

G++ is the C++ compiler from the GNU Compiler Collection. It was used to compile the project source code into executable binaries. The compiler was configured and invoked automatically by the Qt build system using qmake and make commands. Compilation logs and errors were managed through the Qt Creator interface or the Linux terminal.

3.1.7 Linux Terminal

The Linux terminal was used for executing build commands, running the compiled application, and handling project configuration through command-line instructions. It was

also used to resolve build issues, clean the project directory, and perform manual updates when necessary.

3.2 Compiler Version Compatibility and Challenges

During the development of this project, one of the major challenges encountered was ensuring compatibility with an older version of the GNU Compiler Collection (GCC), specifically version 4.4.6. This version, commonly found in legacy Linux environments such as RHEL 6.x or CentOS 6, lacks support for several modern C++ features that are available in newer versions of the standard.

GCC 4.4.6 primarily supports C++98/C++03 standards, and many features introduced in C++11 and later—such as lambda expressions, nullptr, auto type deduction, range-based for loops, std::thread, and smart pointers—are either unsupported or partially supported. Our initial implementation, designed using modern development practices, relied on some of these newer features, particularly in terms of signal-slot connections, object handling, and more readable C++ constructs.

Due to this, the following difficulties were faced:

- Code using nullptr had to be replaced with NULL to avoid compilation errors.
- Range-based for loops had to be rewritten using traditional for-loop syntax.
- Any use of auto or other modern type inference had to be reverted to explicit types.
- Some Qt signal-slot syntax, which was introduced in Qt 5 with C++11 support, had to be replaced with the older SIGNAL () and SLOT() macros compatible with Qt 4 and C++98.

To resolve these issues:

- The project code was reviewed and refactored to ensure strict C++98 compatibility where required.
- Separate conditional compilation was used in some cases using macros to support both versions, where practical.
- The final version of the code was tested and confirmed to run correctly on both the old GCC 4.4.6 environment and a newer setup using GCC 6.9.1, which supported more modern features.

This process helped us ensure backward compatibility without compromising functionality. It also improved our understanding of legacy code maintenance and portability across different system environments.

Chapter 4

4.1 Simulation Code Using Qt 3.3 with Legacy GCC Compiler

This chapter presents the code representation of the core implementation used in our radar simulation project. The code outlines the logic implemented in the radarwidget.cpp file, which is central to our Qt-based graphical user interface. The design aims to ensure clarity in the logical flow while abstracting implementation-level specifics, making it suitable for understanding and future development.

4.1.1 Code: radarwidget.cpp

```
#include "radarwidget.h"

#include <qpainter.h>

#include <qpoint.h>

#include <qrect.h>

#include <qglobal.h>

#include<stdlib.h>

#include <math.h>

RadarWidget::RadarWidget(QWidget* parent, const char* name)

: QWidget(parent, name), angle(90){

timer = new QTimer(this);

connect (timer, SIGNAL (timeout ()), this, SLOT(updateSweep()));

// Do not start timer automatically}

RadarWidget::~RadarWidget() {}

void RadarWidget::startTimer(){

if (!timer->isActive())

timer->start(100);}

void RadarWidget::stopTimer(){
```

```
if (timer->isActive())
timer->stop();}
void RadarWidget::paintEvent(QPaintEvent*){
QPainter p(this);
int w = width(), h = height();
int r = (w < h ? w : h)/2 ;
QPoint center(w / 2 , h / 2);
 p.fillRect(rect(), Qt::darkCyan);
 p.setPen(Qt::white);
for (int i = 1; i <= 7; ++i){
          int radius = r* i/7;
          QRect circleRect(center.x() - radius,center.y() - radius ,2* radius,2* radius);
       p.drawEllipse(circleRect);}
   p.drawLine(center.x(), center.y() - r, center.x(), center.y() + r);
   p.drawLine(center.x() - r, center.y(), center.x() + r, center.y());
   double rad = angle * M_PI / 180.0;
   int x = center.x() + r * cos(rad);
   int y = center.y() - r * sin(rad);
   QPen sweepPen(Qt::black);
   sweepPen.setWidth(2);
   p.setPen(sweepPen);
   p.drawLine(center, QPoint(x, y));
   p.setBrush(Qt::red);
```

```
for(int i = 0; i< numTargets ; ++i){
     QPoint pt = targets [i];
      int dx = pt.x() - center.x();
      int dy = pt.y() - center.y();
      double dist = sqrt(dx * dy + dy * dy );
       if (dist <= r){
                p.drawEllipse(pt.x(),pt.y() ,6,6 );}}}
void RadarWidget::updateSweep(){
    angle = (angle + 1) % 360;
    update();}
void RadarWidget::generateTargets(){
int w= width() ,h=height();
QPoint center ( w/2 , h/2);
int r = ( w <h ? w : h) / 2-20;
numTargets = 4; // fixed number of targets
for ( int i = 0 ; i< numTargets ; ++i){
double t = (rand() % 360 ) * M_PI / 180;
double d = (rand() % ( r - 20))+ 20 ; // avoid center clutter
int x= center.x()+ (int)( d* cos(t));
 int y = center.y()- (int)(d* sin(t));
targets[i] = QPoint (x,y);}}
```

Output:

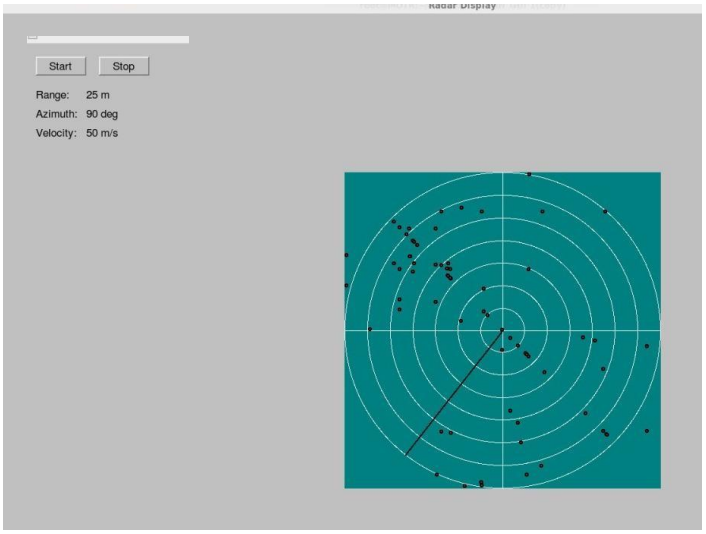

Figure 5 Radar Display using Qt3.3 with GCC 4.4.6 compiler

4.1.2 Challenges with Legacy Compiler

We used GCC version 4.4.6, which introduced the following difficulties:

- Lack of support for modern C++11 features (e.g., auto, range-based loops).
- Need for explicit pointer handling and manual memory management.
- Compatibility issues with some Qt constructs required code simplification.

To overcome these:

- Rewrote logic using legacy-compatible syntax.
- Avoided use of advanced features and maintained backward-compatible Qt methods.
- Verified output in both GCC 4.4.6 and newer GCC 6.9.1 environments to ensure consistency.

4.2. Simulation Code Using Qt 17.0.0 with Qt 6.9.1 MinGW 64 bit-Debug Compiler

The project was also tested and executed using GCC version 6.9.1. This modern compiler offered several improvements:

- **Support for Modern Features**: Enabled the use of features like auto, range-based loops, and lambda expressions, which improved code clarity and efficiency.

- **Simplified Code Structure**: In the newer version, we were able to eliminate the need for separate RadarWidget.cpp/h files. All related functionalities were integrated directly into MainWindow.cpp/h, simplifying the codebase.

- **Enhanced Qt Compatibility**: Modern Qt constructs were supported without compatibility issues, reducing the need for workarounds.

- **Improved Debugging and Performance**: Faster compilation and better diagnostic messages made the development process smoother.

- **Memory Safety**: Availability of modern memory management tools such as smart pointers improved safety and reliability.

Despite simplifying the structure for the modern compiler, we ensured that the output remained consistent with the legacy version, verifying cross-version portability and correctness.

4.2.1. Code: mainwindow.cpp

```
#include "mainwindow.h"

#include <QPainter>

#include <QTime>

#include <cmath>

MainWindow::MainWindow(QWidget *parent)

    : QMainWindow(parent), sweepAngle(0.0) {

    timer = new QTimer(this);

    connect(timer, &QTimer::timeout, this, &MainWindow::updateRadar);

    central = new QWidget(this);

    setCentralWidget(central);

    startButton = new QPushButton("Start", this);

    stopButton = new QPushButton("Stop", this);

    analyzeButton = new QPushButton("Analyze", this);

    targetInfo = new QLabel("Target Info", this);
```

```
    targetInfo->setWordWrap(true);
    QVBoxLayout *layout = new QVBoxLayout(central);
    layout->addWidget(startButton);
    layout->addWidget(stopButton);
    layout->addWidget(analyzeButton);
    layout->addWidget(targetInfo);
    layout->setAlignment(Qt::AlignTop);
    connect(startButton, &QPushButton::clicked, this, &MainWindow::startRadar);
    connect(stopButton, &QPushButton::clicked, this, &MainWindow::stopRadar);
    connect(analyzeButton, &QPushButton::clicked, this, &MainWindow::analyzeTargets);
    resize(600, 600);
    generateTargets();}
MainWindow::~MainWindow() {}
void MainWindow::startRadar() {
    timer->start(100); // update every 100ms
}
void MainWindow::stopRadar() {
    timer->stop();
}
void MainWindow::analyzeTargets() {
    QString info;
    for (const Target &t : targets) {
        info += QString("Azimuth: %1°, Range: %2 km, Strength: %3\n")
```

```
                .arg(t.azimuth, 0, 'f', 1)
                .arg(t.range, 0, 'f', 1)
                .arg(t.signalStrength);
    }
    targetInfo->setText(info);
}
void MainWindow::generateTargets() {
    targets.clear();
    QStringList strengths = {"Weak", "Moderate", "Strong"};
    for (int i = 0; i < 10; ++i) {
        Target t;
        t.azimuth = QRandomGenerator::global()->generateDouble() * 360.0;
        t.range = QRandomGenerator::global()->generateDouble() * 200.0 + 50.0; // 50–250 km
        int idx = QRandomGenerator::global()->bounded(strengths.size());
        t.signalStrength = strengths.at(idx);
        targets.append(t);
    }
}
void MainWindow::updateRadar() {
    sweepAngle += 1.0;
    if (sweepAngle >= 360.0)
        sweepAngle -= 360.0;
    updateTargetInfoNearSweep();
```

```
    update();
}
void MainWindow::updateTargetInfoNearSweep() {
    QString info;
    const double threshold = 2.0; // ±2 degrees from sweep line
    for (const Target &t : targets) {
        double diff = std::fmod(std::fabs(t.azimuth - sweepAngle), 360.0);
        if (diff > 180.0) diff = 360.0 - diff;
        if (diff <= threshold) {
            info += QString("Azimuth: %1°, Range: %2 km, Strength: %3\n")
                        .arg(t.azimuth, 0, 'f', 1)
                        .arg(t.range, 0, 'f', 1)
                        .arg(t.signalStrength);
        }
    }
    if (info.isEmpty()) {
        info = "No targets under sweep.";
    }
    targetInfo->setText(info);
}
void MainWindow::paintEvent(QPaintEvent *) {
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing)
```

```
int size = qMin(width(), height()) - 40;
QPoint center(width() / 2, height() / 2);
    painter.fillRect(rect(), QColor(0, 0, 64));    // Blue background
    // Draw range rings
    painter.setPen(QPen(Qt::green, 1));
    for (int r = 50; r <= 250; r += 50) {
        painter.drawEllipse(center, r, r);
        painter.drawText(center.x() + r, center.y(), QString("%1 km").arg(r));
    }
    // Draw sweep line
    painter.setPen(QPen(Qt::green, 2));
    double radians = qDegreesToRadians(sweepAngle);
    int length = 250;
    QPoint end(center.x() + length * std::cos(radians),
               center.y() - length * std::sin(radians));
    painter.drawLine(center, end);
    // Draw red targets
    painter.setBrush(Qt::red);
    painter.setPen(Qt::NoPen);
    for (const Target &t : targets) {
        double angle = qDegreesToRadians(t.azimuth);
        double dist = t.range;
        QPoint pos(center.x() + dist * std::cos(angle),
```

```
            center.y() - dist * std::sin(angle));

    painter.drawEllipse(pos, 5, 5);

}}
```

Output:

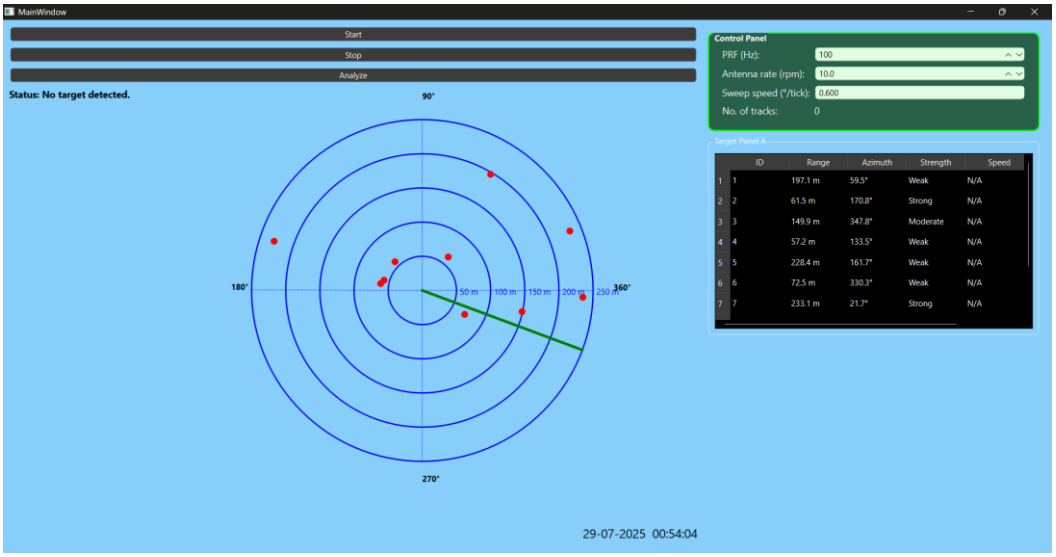

Figure 6 *Radar display with a sweeping line, a target detected with parameters in target panel A and with the details of display in the control panel.*

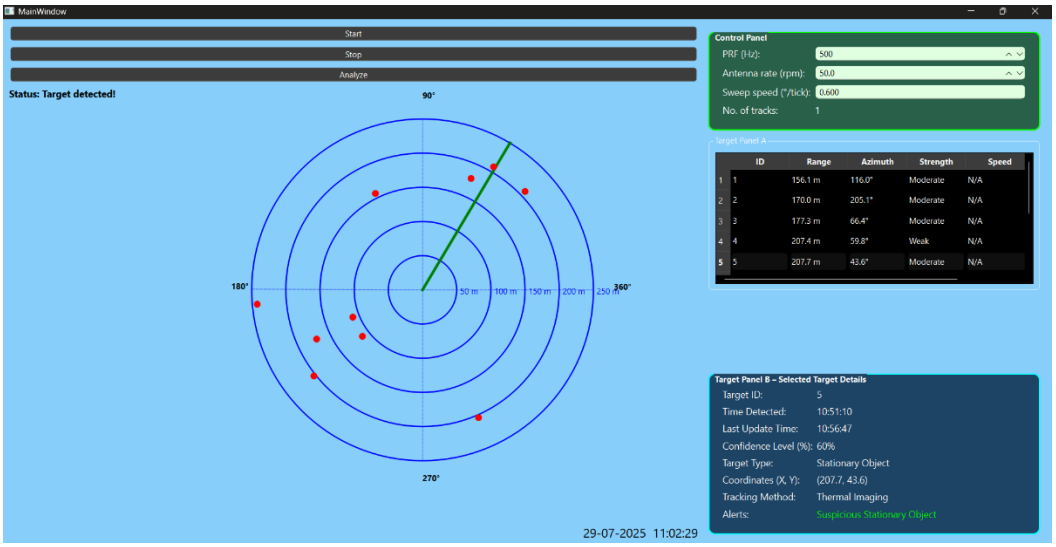

Figure 6 *Display of target Panel B with detailed information about the selected target.*

Chapter 5

Conclusion

5.1 Summary

This project successfully demonstrated the implementation of a radar-based target detection and tracking interface using Qt in both legacy and modern development environments. The core functionality was developed and validated using the older GCC 4.4.6 compiler, which posed significant challenges due to limited support for modern C++ features and Qt constructs. Through careful restructuring and code simplification—particularly by separating logic into files like radarwidget.cpp and mainwindow.cpp—we ensured compatibility and maintained output integrity.

Transitioning to GCC 6.9.1 allowed us to streamline the implementation by consolidating functionality into mainwindow.cpp and mainwindow.h, leveraging newer language features and reducing code complexity. The project output remained consistent across both environments, confirming the robustness of our design and logic.

In summary, the project enhanced our understanding of GUI-based radar simulation, cross-version software compatibility, and adaptive development practices—laying a strong foundation for further enhancements and embedded systems integration in the future.

5.2 References

- The Qt Company, Qt 4.8 Reference Documentation, The Qt Company Ltd., 2011. Available at: https://doc.qt.io/qt-4.8/
- J. Blanchette and M. Summerfield, C++ GUI Programming with Qt 4, 2nd ed., Prentice Hall, 2008.
- GCC Team, "GCC, the GNU Compiler Collection," GNU Project, [Online]. Available: https://gcc.gnu.org/
- Qt Documentation, "Qt Widgets - Application Development Framework," Qt Company, [Online]. Available: https://doc.qt.io/qt-5/qtwidgets-module.html
- Bjarne Stroustrup, The C++ Programming Language, 4th ed., Addison- Wesley, 2013