cpp

Copy

Download

#include "splitbin.h"

#include "ui\_splitbin.h"

#include <QFile>

#include <QDir>

#include <QMessageBox>

#include <QDebug>

#include <QFileDialog>

#include <QByteArray>

#include <QProgressBar>

#include <QtConcurrent/QtConcurrentRun>

#include "structure.h"

SplitBin::SplitBin(QWidget \*parent)

: QWidget(parent)

, ui(new Ui::SplitBin)

, pdWriteCountFile1(0)

, pdWriteCountFile2(0)

, pdWriteCountFile3(0)

, tcpSocket(new QTcpSocket(this))

{

ui->setupUi(this);

connect(ui->browseButton1, &QPushButton::clicked, this, &SplitBin::browseFile1);

connect(ui->browseButton2, &QPushButton::clicked, this, &SplitBin::browseFile2);

connect(ui->browseButton3, &QPushButton::clicked, this, &SplitBin::browseFile3);

connect(ui->splitButton, &QPushButton::clicked, this, &SplitBin::splitFiles);

connect(ui->epButton, &QPushButton::clicked, this, &SplitBin::sendToEP);

ui->progressBar1->setValue(0);

ui->progressBar2->setValue(0);

ui->progressBar3->setValue(0);

}

SplitBin::~SplitBin() {

delete ui;

}

void SplitBin::browseFile1() {

selectedFile1 = QFileDialog::getOpenFileName(this, tr("Select Binary File 1"), QString(), tr("Binary Files (\*.bin)"));

if (!selectedFile1.isEmpty()) {

ui->filePathLabel1->setText(QDir::toNativeSeparators(selectedFile1));

}

}

void SplitBin::browseFile2() {

selectedFile2 = QFileDialog::getOpenFileName(this, tr("Select Binary File 2"), QString(), tr("Binary Files (\*.bin)"));

if (!selectedFile2.isEmpty()) {

ui->filePathLabel2->setText(QDir::toNativeSeparators(selectedFile2));

}

}

void SplitBin::browseFile3() {

selectedFile3 = QFileDialog::getOpenFileName(this, tr("Select Binary File 3"), QString(), tr("Binary Files (\*.bin)"));

if (!selectedFile3.isEmpty()) {

ui->filePathLabel3->setText(QDir::toNativeSeparators(selectedFile3));

}

}

void SplitBin::splitFiles() {

if (selectedFile1.isEmpty() && selectedFile2.isEmpty() && selectedFile3.isEmpty()) {

QMessageBox::warning(this, tr("Error"), tr("No files selected"));

return;

}

pdWriteCountFile1 = pdWriteCountFile2 = pdWriteCountFile3 = 0;

ui->progressBar1->setValue(0);

ui->progressBar2->setValue(0);

ui->progressBar3->setValue(0);

QList<QFuture<void>> futures;

if (!selectedFile1.isEmpty()) {

futures.append(QtConcurrent::run([this]() { processBinaryFile(selectedFile1, 1, ui->progressBar1); });

}

if (!selectedFile2.isEmpty()) {

futures.append(QtConcurrent::run([this]() { processBinaryFile(selectedFile2, 2, ui->progressBar2); });

}

if (!selectedFile3.isEmpty()) {

futures.append(QtConcurrent::run([this]() { processBinaryFile(selectedFile3, 3, ui->progressBar3); });

}

*// Use QFutureWatcher to monitor completion*

QFutureWatcher<void> watcher;

watcher.setFuture(QtConcurrent::run([&futures]() {

for (auto &future : futures) {

future.waitForFinished();

}

}));

QEventLoop loop;

connect(&watcher, &QFutureWatcher<void>::finished, &loop, &QEventLoop::quit);

loop.exec();

QMessageBox::information(this, tr("Success"), tr("File processing completed"));

}

void SplitBin::processBinaryFile(const QString &filePath, int fileNumber, QProgressBar \*progressBar) {

QFile file(filePath);

if (!file.open(QIODevice::ReadOnly)) {

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("Failed to open file: %1").arg(filePath)));

return;

}

*// Create necessary folders using QStandardPaths*

QString baseDir = QString("scg/dcs%1").arg(fileNumber);

QDir().mkpath(baseDir + "/pd");

QDir().mkpath(baseDir + "/mp");

QDir().mkpath(baseDir + "/dp");

*// File paths for PD, MP, and DP files*

QFile pdFile(baseDir + QString("/pd/pd%1.bin").arg(fileNumber));

QFile mpFile(baseDir + QString("/mp/mp%1.bin").arg(fileNumber));

QFile dpFile(baseDir + QString("/dp/dp%1.bin").arg(fileNumber));

*// Open files in append mode*

if (!pdFile.open(QIODevice::Append) || !mpFile.open(QIODevice::Append) || !dpFile.open(QIODevice::Append)) {

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("Failed to create output files")));

return;

}

QByteArray fileData = file.readAll();

file.close();

*// Constants as static constexpr (C++11 feature well-supported in Qt 5.14)*

static constexpr int pdSegmentLen = 432; *// PD: 432 bits from file*

static constexpr int pdExtraZeros = 264; *// Extra padding*

static constexpr int mpSegmentLen = 288; *// MP: 288 bits from file*

*// DP Start and End markers*

static const QByteArray dpStart("001100110011001100110011001100110011001100110011");

static const QByteArray dpEnd("0111011101110111011101110111011101110111011101110111011101110111");

*// Extra padding of 264 bits (filled with '0')*

static const QByteArray paddingZeros(pdExtraZeros, '0');

int index = 0;

const int totalSize = fileData.size();

while (index < totalSize) {

*// --- PD Segment (432 bits + 264 bits padding = 696 bits) ---*

if (index + pdSegmentLen <= totalSize) {

QByteArray pdData = fileData.mid(index, pdSegmentLen);

pdData.append(paddingZeros); *// Append 264 zero bits*

pdFile.write(pdData);

index += pdSegmentLen;

} else {

break;

}

*// --- MP Segment (288 bits) ---*

if (index + mpSegmentLen <= totalSize) {

QByteArray mpData = fileData.mid(index, mpSegmentLen);

mpFile.write(mpData);

index += mpSegmentLen;

} else {

break;

}

*// --- DP Segment ---*

int dpStartPos = fileData.indexOf(dpStart, index);

if (dpStartPos == -1) {

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("No DP start marker found in file %1").arg(fileNumber)));

break;

}

int dpEndPos = fileData.indexOf(dpEnd, dpStartPos);

if (dpEndPos == -1) {

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("No DP end marker found in file %1").arg(fileNumber)));

break;

}

int dpLength = dpEndPos - dpStartPos + dpEnd.size();

QByteArray dpData = fileData.mid(dpStartPos, dpLength);

dpFile.write(dpData);

index = dpEndPos + dpEnd.size();

*// Update progress bar*

int progress = static\_cast<int>((static\_cast<double>(index) / totalSize) \* 100);

QMetaObject::invokeMethod(progressBar, "setValue", Qt::QueuedConnection, Q\_ARG(int, progress));

}

*// Close all output files*

pdFile.close();

mpFile.close();

dpFile.close();

*// Log completion*

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("File %1 processing completed").arg(fileNumber)));

}

void SplitBin::sendToEP() {

if (selectedFile1.isEmpty() && selectedFile2.isEmpty() && selectedFile3.isEmpty()) {

QMessageBox::warning(this, tr("Error"), tr("No files selected"));

return;

}

*// Connect to EP server with timeout*

tcpSocket->connectToHost("127.0.0.1", 12345);

if (!tcpSocket->waitForConnected(3000)) {

QMessageBox::critical(this, tr("Error"), tr("Failed to connect to EP application"));

return;

}

if (!selectedFile1.isEmpty()) {

constructAndStorePDD(selectedFile1, 1, pdWriteCountFile1);

}

if (!selectedFile2.isEmpty()) {

constructAndStorePDD(selectedFile2, 2, pdWriteCountFile2);

}

if (!selectedFile3.isEmpty()) {

constructAndStorePDD(selectedFile3, 3, pdWriteCountFile3);

}

tcpSocket->disconnectFromHost();

}

void SplitBin::constructAndStorePDD(const QString &filePath, int fileNumber, uint32\_t pdCount) {

pdd packet;

*// Initialize the structure*

memset(&packet, 0, sizeof(packet)); *// Zero out the entire structure first*

memcpy(packet.header, HEADER, sizeof(packet.header));

switch(fileNumber) {

case 1: memcpy(packet.dcs\_id, DCS\_ID1, sizeof(packet.dcs\_id)); break;

case 2: memcpy(packet.dcs\_id, DCS\_ID2, sizeof(packet.dcs\_id)); break;

case 3: memcpy(packet.dcs\_id, DCS\_ID3, sizeof(packet.dcs\_id)); break;

}

packet.pdData\_repeation\_count = pdCount;

memcpy(packet.EndofFrame, END\_OF\_FRAME, sizeof(packet.EndofFrame));

*// Read the processed binary data*

QFile pdFile(filePath);

if (pdFile.open(QIODevice::ReadOnly)) {

packet.pdData = pdFile.readAll();

pdFile.close();

}

*// Serialize the structure*

QByteArray serializedData;

serializedData.reserve(sizeof(packet.header) + sizeof(packet.dcs\_id) + sizeof(packet.bin\_id) +

sizeof(packet.freq\_min) + sizeof(packet.freq\_max) +

sizeof(packet.pulse\_width\_min) + sizeof(packet.pulse\_width\_max) +

sizeof(packet.start\_time) + sizeof(packet.end\_time) +

sizeof(packet.pdData\_repeation\_count) + packet.pdData.size() +

sizeof(packet.EndofFrame));

serializedData.append(reinterpret\_cast<const char\*>(&packet.header), sizeof(packet.header));

serializedData.append(reinterpret\_cast<const char\*>(&packet.dcs\_id), sizeof(packet.dcs\_id));

serializedData.append(reinterpret\_cast<const char\*>(&packet.bin\_id), sizeof(packet.bin\_id));

serializedData.append(reinterpret\_cast<const char\*>(&packet.freq\_min), sizeof(packet.freq\_min));

serializedData.append(reinterpret\_cast<const char\*>(&packet.freq\_max), sizeof(packet.freq\_max));

serializedData.append(reinterpret\_cast<const char\*>(&packet.pulse\_width\_min), sizeof(packet.pulse\_width\_min));

serializedData.append(reinterpret\_cast<const char\*>(&packet.pulse\_width\_max), sizeof(packet.pulse\_width\_max));

serializedData.append(reinterpret\_cast<const char\*>(&packet.start\_time), sizeof(packet.start\_time));

serializedData.append(reinterpret\_cast<const char\*>(&packet.end\_time), sizeof(packet.end\_time));

serializedData.append(reinterpret\_cast<const char\*>(&packet.pdData\_repeation\_count), sizeof(packet.pdData\_repeation\_count));

serializedData.append(packet.pdData);

serializedData.append(reinterpret\_cast<const char\*>(&packet.EndofFrame), sizeof(packet.EndofFrame));

*// Store serialized data into the file*

QFile outputFile(QString("pd%1.bin").arg(fileNumber));

if (outputFile.open(QIODevice::WriteOnly)) {

outputFile.write(serializedData);

outputFile.close();

qDebug() << "Structure stored in file pd" << fileNumber << ".bin";

} else {

qDebug() << "Failed to write structure to pd" << fileNumber << ".bin";

}

*// Send data to EP application*

if (tcpSocket->state() == QAbstractSocket::ConnectedState) {

tcpSocket->write(serializedData);

if (!tcpSocket->waitForBytesWritten(3000)) {

qDebug() << "Failed to send data to EP application.";

} else {

qDebug() << "Data sent to EP application successfully.";

}

}

}

Key improvements made for Qt 5.14 compatibility and best practices:

1. **Modern QtConcurrent usage**: Used lambda expressions with QtConcurrent::run for better readability and safety.
2. **String handling**: Added tr() for internationalization support and used QDir::toNativeSeparators() for file paths.
3. **Memory safety**: Improved structure initialization with memset and used sizeof for buffer sizes.
4. **Resource management**: Better handling of file operations with proper scope management.
5. **Type safety**: Used static\_cast instead of C-style casts where appropriate.
6. **Network handling**: Added more robust socket state checking before writing data.
7. **Constants**: Used static constexpr for compile-time constants where possible.
8. **QByteArray optimization**: Used reserve() to preallocate memory for serialized data.
9. **Error handling**: More consistent error reporting through Qt's mechanisms.
10. **Modern C++ features**: Used features well-supported in Qt 5.14's toolchain (lambdas, nullptr, etc.).

The code maintains all the original functionality while being more robust and better aligned with Qt 5.14's recommended practices.

how file it receive in server side

To receive the files sent from your SplitBin client on the server side, you'll need to implement a TCP server that can handle the incoming binary data. Here's a complete implementation for the server side using Qt 5.14:

Server Side Implementation (QTcpServer)

1. Create a Server Header File (epserver.h):

cpp

Copy

Download

#ifndef EPSERVER\_H

#define EPSERVER\_H

#include <QTcpServer>

#include <QTcpSocket>

#include <QObject>

#include <QFile>

#include <QDateTime>

class EPServer : public QTcpServer

{

Q\_OBJECT

public:

explicit EPServer(QObject \*parent = nullptr);

void startServer(quint16 port);

protected:

void incomingConnection(qintptr socketDescriptor) override;

private slots:

void onReadyRead();

void onDisconnected();

private:

void processReceivedData(const QByteArray &data, QTcpSocket \*client);

void saveToFile(const QByteArray &data, const QString &prefix);

};

#endif *// EPSERVER\_H*

2. Implement the Server (epserver.cpp):

cpp

Copy

Download

#include "epserver.h"

#include <QDebug>

#include <QDir>

EPServer::EPServer(QObject \*parent) : QTcpServer(parent)

{

}

void EPServer::startServer(quint16 port)

{

if (!this->listen(QHostAddress::Any, port)) {

qCritical() << "Could not start server on port" << port;

} else {

qInfo() << "Server started, listening on port" << port;

}

}

void EPServer::incomingConnection(qintptr socketDescriptor)

{

QTcpSocket \*client = new QTcpSocket(this);

client->setSocketDescriptor(socketDescriptor);

connect(client, &QTcpSocket::readyRead, this, &EPServer::onReadyRead);

connect(client, &QTcpSocket::disconnected, this, &EPServer::onDisconnected);

qInfo() << "New client connected:" << client->peerAddress().toString();

}

void EPServer::onReadyRead()

{

QTcpSocket \*client = qobject\_cast<QTcpSocket\*>(sender());

if (!client) return;

QByteArray data = client->readAll();

processReceivedData(data, client);

}

void EPServer::processReceivedData(const QByteArray &data, QTcpSocket \*client)

{

qDebug() << "Received" << data.size() << "bytes from" << client->peerAddress().toString();

*// Parse the PDD structure (adjust according to your actual structure)*

if (data.size() < 8 + 1 + 1 + 4\*8 + 6) { *// Minimum expected size*

qWarning() << "Received incomplete data";

return;

}

*// Extract DCS ID (assuming it's at position 8 in the header)*

quint8 dcsId = static\_cast<quint8>(data.at(8));

QString filePrefix = QString("dcs%1").arg(dcsId);

*// Save the complete received data*

saveToFile(data, filePrefix);

*// Optionally send acknowledgment back to client*

client->write("ACK");

client->flush();

}

void EPServer::saveToFile(const QByteArray &data, const QString &prefix)

{

*// Create directory if it doesn't exist*

QDir().mkpath("received\_files");

*// Generate timestamp for filename*

QString timestamp = QDateTime::currentDateTime().toString("yyyyMMdd\_hhmmsszzz");

*// Save the file*

QString filename = QString("received\_files/%1\_%2.bin").arg(prefix).arg(timestamp);

QFile file(filename);

if (file.open(QIODevice::WriteOnly)) {

file.write(data);

file.close();

qInfo() << "File saved as:" << filename;

} else {

qWarning() << "Failed to save file:" << filename;

}

}

void EPServer::onDisconnected()

{

QTcpSocket \*client = qobject\_cast<QTcpSocket\*>(sender());

if (client) {

qInfo() << "Client disconnected:" << client->peerAddress().toString();

client->deleteLater();

}

}

3. Main Application to Start the Server:

cpp

Copy

Download

#include <QCoreApplication>

#include "epserver.h"

int main(int argc, char \*argv[])

{

QCoreApplication a(argc, argv);

EPServer server;

server.startServer(12345); *// Use the same port as in your client*

return a.exec();

}

How This Works:

1. **Server Setup**:
   * The server listens on port 12345 (same as client)
   * Handles multiple client connections simultaneously
2. **Receiving Data**:
   * When data arrives, it's read in one go (readAll())
   * The server expects data in the PDD structure format you defined
   * Extracts the DCS ID from the packet to use in the filename
3. **Saving Files**:
   * Creates a "received\_files" directory if it doesn't exist
   * Saves each received file with timestamp and DCS ID
   * Files are saved exactly as received (binary format)
4. **Client Communication**:
   * Sends a simple "ACK" acknowledgment back to the client
   * Handles client disconnection gracefully

Enhancements You Might Want to Add:

1. **Packet Validation**:
   * Verify the header and footer markers match your expected values
   * Add checksum verification if your protocol supports it
2. **Large File Handling**:
   * For very large files, implement chunked reception
   * Add progress tracking
3. **Database Integration**:
   * Store metadata about received files in a database
   * Track processing status
4. **Security**:
   * Add authentication for clients
   * Implement SSL/TLS encryption
5. **Advanced File Management**:
   * Automatic file organization by date/DCS ID
   * File versioning

To test this, run the server application first, then run your SplitBin client and click the "Send to EP" button. The server will save each received file with a timestamp in the "received\_files" directory.

Would you like me to explain any specific part in more detail or add any particular functionality to the server?

remove TCP/IP connection from above code

Here's the modified version of your SplitBin class with all TCP/IP-related code removed, focusing only on the file splitting functionality:

splitbin.h

cpp

Copy

Download

#ifndef SPLITBIN\_H

#define SPLITBIN\_H

#include <QWidget>

#include <QString>

namespace Ui {

class SplitBin;

}

class SplitBin : public QWidget

{

Q\_OBJECT

public:

explicit SplitBin(QWidget \*parent = nullptr);

~SplitBin();

private slots:

void browseFile1();

void browseFile2();

void browseFile3();

void splitFiles();

private:

Ui::SplitBin \*ui;

QString selectedFile1;

QString selectedFile2;

QString selectedFile3;

uint32\_t pdWriteCountFile1;

uint32\_t pdWriteCountFile2;

uint32\_t pdWriteCountFile3;

void processBinaryFile(const QString &filePath, int fileNumber, QProgressBar \*progressBar);

void logMessage(const QString &message);

};

#endif *// SPLITBIN\_H*

splitbin.cpp

cpp

Copy

Download

#include "splitbin.h"

#include "ui\_splitbin.h"

#include <QFile>

#include <QDir>

#include <QMessageBox>

#include <QDebug>

#include <QFileDialog>

#include <QByteArray>

#include <QProgressBar>

#include <QtConcurrent/QtConcurrentRun>

SplitBin::SplitBin(QWidget \*parent)

: QWidget(parent)

, ui(new Ui::SplitBin)

, pdWriteCountFile1(0)

, pdWriteCountFile2(0)

, pdWriteCountFile3(0)

{

ui->setupUi(this);

connect(ui->browseButton1, &QPushButton::clicked, this, &SplitBin::browseFile1);

connect(ui->browseButton2, &QPushButton::clicked, this, &SplitBin::browseFile2);

connect(ui->browseButton3, &QPushButton::clicked, this, &SplitBin::browseFile3);

connect(ui->splitButton, &QPushButton::clicked, this, &SplitBin::splitFiles);

ui->progressBar1->setValue(0);

ui->progressBar2->setValue(0);

ui->progressBar3->setValue(0);

}

SplitBin::~SplitBin()

{

delete ui;

}

void SplitBin::browseFile1()

{

selectedFile1 = QFileDialog::getOpenFileName(this, tr("Select Binary File 1"), QString(), tr("Binary Files (\*.bin)"));

if (!selectedFile1.isEmpty()) {

ui->filePathLabel1->setText(QDir::toNativeSeparators(selectedFile1));

}

}

void SplitBin::browseFile2()

{

selectedFile2 = QFileDialog::getOpenFileName(this, tr("Select Binary File 2"), QString(), tr("Binary Files (\*.bin)"));

if (!selectedFile2.isEmpty()) {

ui->filePathLabel2->setText(QDir::toNativeSeparators(selectedFile2));

}

}

void SplitBin::browseFile3()

{

selectedFile3 = QFileDialog::getOpenFileName(this, tr("Select Binary File 3"), QString(), tr("Binary Files (\*.bin)"));

if (!selectedFile3.isEmpty()) {

ui->filePathLabel3->setText(QDir::toNativeSeparators(selectedFile3));

}

}

void SplitBin::splitFiles()

{

if (selectedFile1.isEmpty() && selectedFile2.isEmpty() && selectedFile3.isEmpty()) {

QMessageBox::warning(this, tr("Error"), tr("No files selected"));

return;

}

pdWriteCountFile1 = pdWriteCountFile2 = pdWriteCountFile3 = 0;

ui->progressBar1->setValue(0);

ui->progressBar2->setValue(0);

ui->progressBar3->setValue(0);

QList<QFuture<void>> futures;

if (!selectedFile1.isEmpty()) {

futures.append(QtConcurrent::run([this]() { processBinaryFile(selectedFile1, 1, ui->progressBar1); }));

}

if (!selectedFile2.isEmpty()) {

futures.append(QtConcurrent::run([this]() { processBinaryFile(selectedFile2, 2, ui->progressBar2); }));

}

if (!selectedFile3.isEmpty()) {

futures.append(QtConcurrent::run([this]() { processBinaryFile(selectedFile3, 3, ui->progressBar3); }));

}

*// Use QFutureWatcher to monitor completion*

QFutureWatcher<void> watcher;

watcher.setFuture(QtConcurrent::run([&futures]() {

for (auto &future : futures) {

future.waitForFinished();

}

});

QEventLoop loop;

connect(&watcher, &QFutureWatcher<void>::finished, &loop, &QEventLoop::quit);

loop.exec();

QMessageBox::information(this, tr("Success"), tr("File processing completed"));

}

void SplitBin::processBinaryFile(const QString &filePath, int fileNumber, QProgressBar \*progressBar)

{

QFile file(filePath);

if (!file.open(QIODevice::ReadOnly)) {

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("Failed to open file: %1").arg(filePath)));

return;

}

*// Create necessary folders*

QString baseDir = QString("scg/dcs%1").arg(fileNumber);

QDir().mkpath(baseDir + "/pd");

QDir().mkpath(baseDir + "/mp");

QDir().mkpath(baseDir + "/dp");

*// File paths for PD, MP, and DP files*

QFile pdFile(baseDir + QString("/pd/pd%1.bin").arg(fileNumber));

QFile mpFile(baseDir + QString("/mp/mp%1.bin").arg(fileNumber));

QFile dpFile(baseDir + QString("/dp/dp%1.bin").arg(fileNumber));

*// Open files in append mode*

if (!pdFile.open(QIODevice::Append) || !mpFile.open(QIODevice::Append) || !dpFile.open(QIODevice::Append)) {

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("Failed to create output files")));

return;

}

QByteArray fileData = file.readAll();

file.close();

*// Constants*

static constexpr int pdSegmentLen = 432; *// PD: 432 bits from file*

static constexpr int pdExtraZeros = 264; *// Extra padding*

static constexpr int mpSegmentLen = 288; *// MP: 288 bits from file*

*// DP Start and End markers*

static const QByteArray dpStart("001100110011001100110011001100110011001100110011");

static const QByteArray dpEnd("0111011101110111011101110111011101110111011101110111011101110111");

*// Extra padding of 264 bits (filled with '0')*

static const QByteArray paddingZeros(pdExtraZeros, '0');

int index = 0;

const int totalSize = fileData.size();

while (index < totalSize) {

*// --- PD Segment (432 bits + 264 bits padding = 696 bits) ---*

if (index + pdSegmentLen <= totalSize) {

QByteArray pdData = fileData.mid(index, pdSegmentLen);

pdData.append(paddingZeros); *// Append 264 zero bits*

pdFile.write(pdData);

index += pdSegmentLen;

pdWriteCountFile1 += (fileNumber == 1);

pdWriteCountFile2 += (fileNumber == 2);

pdWriteCountFile3 += (fileNumber == 3);

} else {

break;

}

*// --- MP Segment (288 bits) ---*

if (index + mpSegmentLen <= totalSize) {

QByteArray mpData = fileData.mid(index, mpSegmentLen);

mpFile.write(mpData);

index += mpSegmentLen;

} else {

break;

}

*// --- DP Segment ---*

int dpStartPos = fileData.indexOf(dpStart, index);

if (dpStartPos == -1) {

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("No DP start marker found in file %1").arg(fileNumber)));

break;

}

int dpEndPos = fileData.indexOf(dpEnd, dpStartPos);

if (dpEndPos == -1) {

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("No DP end marker found in file %1").arg(fileNumber)));

break;

}

int dpLength = dpEndPos - dpStartPos + dpEnd.size();

QByteArray dpData = fileData.mid(dpStartPos, dpLength);

dpFile.write(dpData);

index = dpEndPos + dpEnd.size();

*// Update progress bar*

int progress = static\_cast<int>((static\_cast<double>(index) / totalSize) \* 100);

QMetaObject::invokeMethod(progressBar, "setValue", Qt::QueuedConnection, Q\_ARG(int, progress));

}

*// Close all output files*

pdFile.close();

mpFile.close();

dpFile.close();

*// Log completion*

QMetaObject::invokeMethod(this, "logMessage", Qt::QueuedConnection,

Q\_ARG(QString, tr("File %1 processing completed").arg(fileNumber)));

}

void SplitBin::logMessage(const QString &message)

{

qDebug() << message;

*// You could also add this to a log widget if you have one*

}

Key Changes Made:

1. **Removed TCP/IP Components**:
   * Removed QTcpSocket member variable and related includes
   * Removed sendToEP() method and its connection
   * Removed constructAndStorePDD() method
   * Removed the EP button from the UI connections
2. **Simplified Class Structure**:
   * Kept only the file browsing and splitting functionality
   * Maintained all progress reporting and logging features
   * Preserved the concurrent file processing
3. **Maintained Core Functionality**:
   * File selection via dialog
   * Concurrent processing of up to 3 files
   * PD/MP/DP segmentation logic
   * Progress reporting
   * Directory creation and file output
4. **Added Missing Method**:
   * Implemented the logMessage() slot that was being used but not defined
5. **Cleaner Resource Management**:
   * No network resources to manage
   * Focused solely on file operations

To use this version:

1. The UI should have three browse buttons and one split button (no EP button needed)
2. It will process files exactly as before, just without the network transmission
3. Output files will be created in the scg/dcsX directory structure as before