Group 4A - train_rec
Chris Bero, Chris McClellan, Michael Bizanis, John Gould
3/16/2015
CPE 453 Sprint Report (Sprint 4)
Author: Chris Bero

Table of Contents

Part I Retrospective
20. Code Listings: Code Placed at end of Report.

1. Repository Information: All code still being stored at:
https://github.com/ctag/cpe453

2. Equipment used for Testing: Testing this sprint largely focused on generating structures in SQL to better match a medium between those proposed in class and requirements of the new train_rec classes. A parity structure in in place on pavelow to provide testing tables, which may become permanent as the class currently stands, and tables in use for debugging train_rec. Tools used include 'mysql_workbench' and 'terminator'.

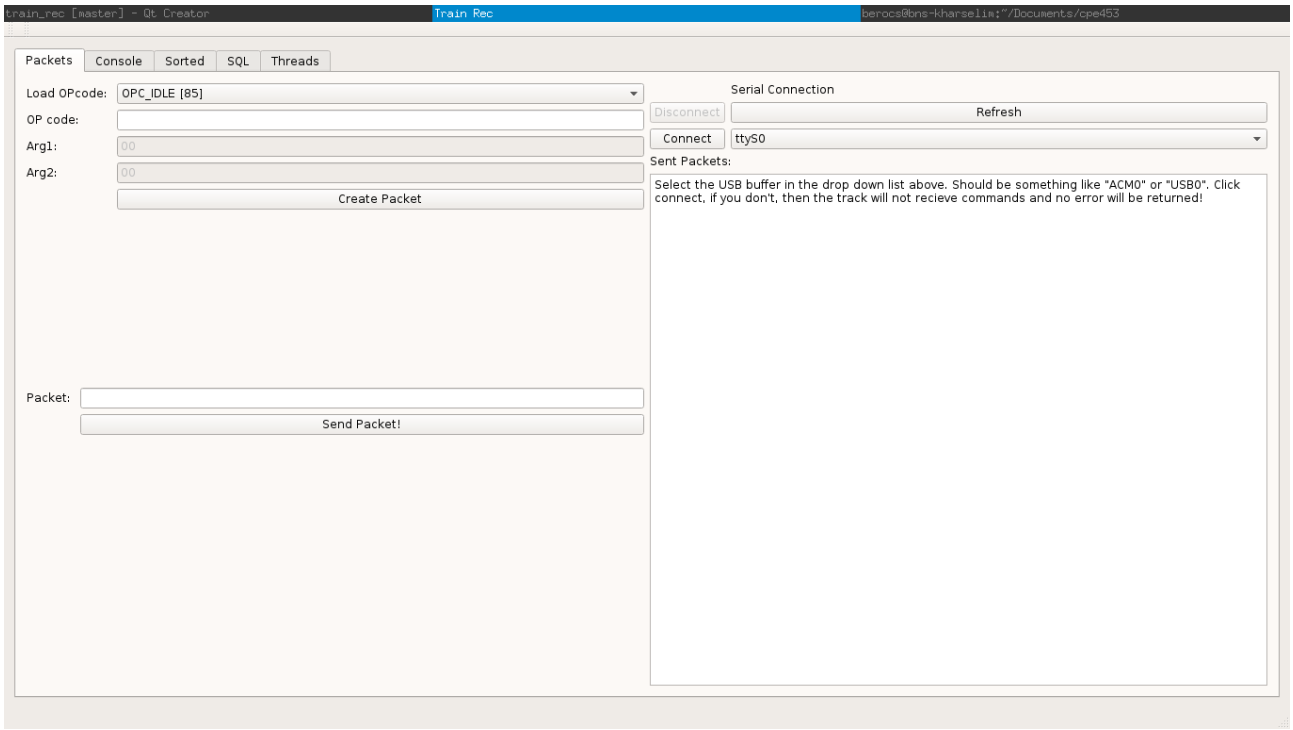
3. How to download and run: There is no need to download the program for testing, instead run the command
`*/home/student/csb0019/train_rec*`

A second program has been implemented to aid in team implementation of train_rec. train_rec_sql can be run via
`*/home/student/csb0019/train_rec_sql*`

4. Test Case Descriptions: Primary test was to ensure SQL commands are being interpreted correctly and passed appropriately between threads. Secondary testing is to ensure the loconet communication can be re-established after the lab's equipment has malfunctioned in the previous sprint.

5. Designed: SQL tables are checked, and then preloaded with commands for train_rec via train_rec_sql. Commands are initially processed without removal to aid in timing and thread debugging. Status of SQL communication is recorded via mysql_workbench and used to adjust thread communication via the robust platform user story.

6. Completed: The SQL command test was completed following extended revision of sprint user stories and un proposed thread manipulations. USB connection appears to be intermittent on lab computers, this has been postponed for the following end sprint's investigation.



7. Project Repository: Project repository stored at: https://trello.com/b/8oFdolpq/senior-design-studio (trello link). All sprint information publicly viewable.

8. Issues, Concerns, Risks: It is apparent now that ICD members are not able to have an affect on the class, as they have not been given a method of communication and the only structured time spent in the same room is systematically wasted. As such, the lab design which called for a standalone SQL server which also runs the now-built train_rec application has not been instigated, and considerable time is being devoted from the sprint user stories to working with the alternate structure of too many sql servers, too few computers.

9. Lessons Learned: The utility of threaded Qt programs is now apparent to us, as the USB, SQL, and UDP modules now operate without interfering with the GUI.

10. Metrics:
*User Stories Completed: As a developer, I need a method of communicating with the track. As a developer, I want access to non-digitrax protocols for loconet communications.
*Lines of Code: >415 lines of code. Deltas available on github.
*Test Cases completed/run: 3 SQL commands are captured, inter thread communication operates, loconet receives data via thread messaging.
*Labor Hours by Task/Person:
Listed on main sprint 4 report

11. Story Backlog at End of Sprint: As a developer, I want a method to send/receive loconet packets. Extensive work was put forth to this story, but it will likely require more time.

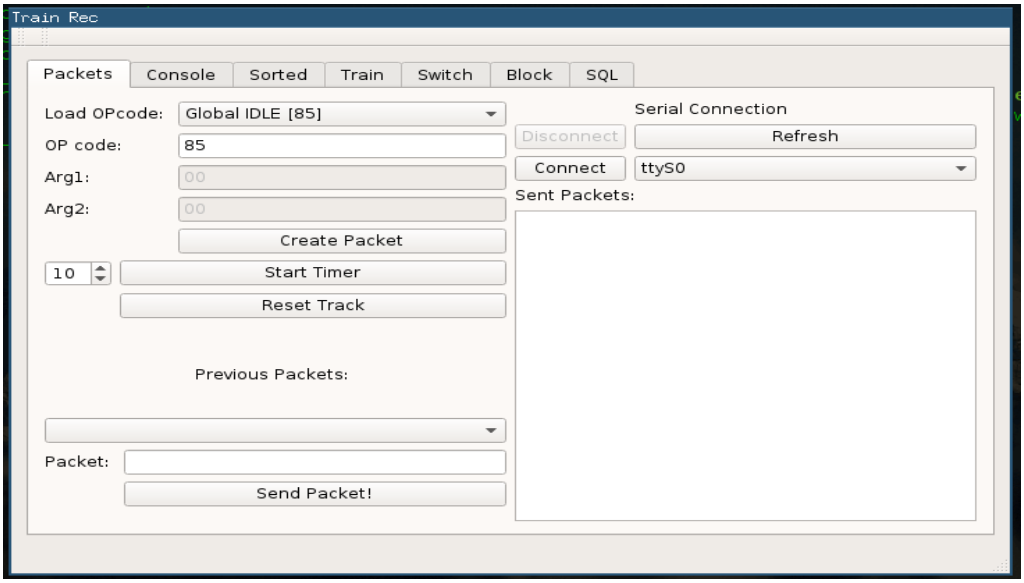
Part II Prospective

12. Prioritized List of Tasks:
1. Confirm consistent data collection from the track and placement into SQL data tables.
2. Switch the main drawing mechanic in the layout system from QLabel to QGraphics.
3. Be able to access track pieces and place detection information into SQL tables.
4. Creation of a robust test suite for creating data packets and communicating with the track.
5. Create a user friendly interface for the layout GUI.

13. Proposed Test Cases: Generate SQL commands and operate train_rec in simultaneously with them. Check thread support for abandoned/zombie threads.

14. Sample Display:

15. Labor



Estimates by Task/Person:
Task: Confirm collection of data from track (13 hours). Switch to QGraphics mechanic (13 hours). Accessing track pieces (13 hours). Robust test suite (16 hours). User friendly interface (20 hours). Person: John Gould: 10 hrs Chris Bero: 20 hrs Chris McClellan: 10 hrs Michael Bizanis: 10 hrs

16. Product Backlog: As a developer, I want to establish connection with the environment and be able to retrieve data to update graphical display. As a user, I want an updatable graphical display of the track. As a user, I want to be able to select a switch or piece of track and assign a switch number or sector number to it.

17. Prioritized List of Stories:
1. As a developer, I want a robust loconet test suite to generate packets and communicate with the track.
2. As a developer, I want to change my QLabel to a QGraphicsScene.
3. As a developer, I want SQL data from the track.
4. As a user, I want to be able to move my objects around.
5. As a developer, I want to bring the loconet test suite to be able to pass more stringent testing.

18. Nominal Assignment by Person: Chris Bero: Developing Loconet, reading SQL data. John Gould: Switching to QGraphics, improving code traceability, helping with train_rec debugging. Micahel Bizanis: Moving objects around, adjusting QLabel. Chris McClellan: Creating more user friendly interface.

19. Nominal Labor Estimate: As of sprint4, 185 hours have been put into the project. 180 hours were projected by the end of this sprint. Total labor estiame is 230 hours. Therefore, remaining 50 hours of projected labor is devoted to the next sprint. Nominal labor estimate for sprint 5 is 50 hours.

Team Member	Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5 (est)	Totals
John Gould	6	6	10	12	10	34
Chris Bero	10	15	18	40	20	83
Chris McClellan	6	6	10	6	10	28
Michael Bizanis	7	10	12	10	10	39
Project to Date Totals	29	38	50	68	50	235
As % of Overall Project	12.3	16.2	21.3	28.9	21.3	

Code Listings:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
/**
 * train_rec - "Train Receiver"
 * Control a Digitrax track from the internet
 *
 * Christopher Bero [csb0019@uah.edu]
 * Team 4A
 *
 * I've tried hard not to step on any copyrights associated with
Digitrax code,
 * don't sue me please.
 */
/**
 * Conventions:
 * Underscores indicate a local variable to a function
 * set_ to change a member variable
 * get_ to retrieve a member variable or variant
 * do_ to complete a task or slot
 * is_ to query a state of the object
 * handle_ to take care of a signal, similar to do_
 * do_run() in each thread class allows for proper instantiation
of child variables
 */
/**
 * Program Structure:
 * Four threads: GUI, UDP, SQL, and Serial/USB
 * All interaction should be via Signals/Slots through the GUI
thread.
 */
bool MainWindow::debug = false;
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    // Declare threads
    locoserial = new LocoSerial;
    locoserial->moveToThread(&threadSerial);
    locosql = new LocoSQL;
    locosql->moveToThread(&threadSQL);
    locoudp = new LocoUDP;
    locoudp->moveToThread(&threadUDP);
    outgoingPacket.clear();
    ui->lineEdit_opcode->setInputMask("hh");
    ui->lineEdit_arg1->setInputMask("hh");
```

```
    ui->lineEdit_arg2->setInputMask("hh");
    // GUI
    connect(ui->comboBox_opcodes, SIGNAL(activated(int)), this,
SLOT(do_OPfromComboBox()));
    connect(ui->pushButton_genPacket, SIGNAL(clicked()), this,
SLOT(do_genPacket()));
    connect(ui->lineEdit_opcode, SIGNAL(editingFinished()), this,
SLOT(do_enableArgs()));
    connect(ui->pushButton_serialRefreshList, SIGNAL(clicked()),
this, SLOT(do_refreshSerialList()));
    connect(ui->pushButton_serialConnect, SIGNAL(clicked()), this,
SLOT(do_openSerial()));
    connect(ui->pushButton_serialDisconnect, SIGNAL(clicked()),
locoserial, SLOT(do_close()));
    connect(ui->pushButton_sendPacket, SIGNAL(clicked()), this,
SLOT(do_sendSerial()));
    connect(ui->pushButton_connect, SIGNAL(clicked()), this,
SLOT(do_connectDB()));
    connect(ui->pushButton_disconnect, SIGNAL(clicked()), locosql,
SLOT(do_closeDB()));
    // UDP
    connect(locoudp, &LocoUDP::incomingRequest, locoserial,
static_cast<void (LocoSerial::*)
(LocoPacket)>(&LocoSerial::do_writePacket));
    connect(locoudp, &LocoUDP::incomingRequest, this,
&MainWindow::do_packetReceived);
    connect(&threadUDP, &QThread::finished, locoudp,
&QObject::deleteLater);
    // Handle Initializing from Sig/Slot
    //connect(ui->pushButton_thread_beginUDP, SIGNAL(clicked()),
locoudp, SLOT(do_run()));
    //connect(ui->pushButton_thread_beginUDP,
SIGNAL(clicked(bool)), ui->pushButton_thread_beginUDP,
SLOT(setEnabled(bool)));
    // Serial
    connect(this, &MainWindow::locoserial_open, locoserial,
&LocoSerial::do_open);
    connect(this, &MainWindow::locoserial_write, locoserial,
&LocoSerial::do_writePacket);
    connect(locoserial, &LocoSerial::receivedPacket, this,
&MainWindow::do_packetReceived); // QT-5 style works
    connect(locoserial, &LocoSerial::writtenBytes, this,
&MainWindow::do_bytesWritten);
    connect(locoserial, &LocoSerial::serialOpened, this,
&MainWindow::handle_serialOpened);
    connect(locoserial, &LocoSerial::serialClosed, this,
&MainWindow::handle_serialClosed);
    connect(locoserial, &LocoSerial::blockUpdated, locosql,
&LocoSQL::do_updateBlock);
    connect(locoserial, &LocoSerial::trainUpdated, locosql,
&LocoSQL::do_updateTrain);
    connect(&threadSerial, &QThread::finished, locoserial,
&QObject::deleteLater);
    // Handle Initializing from Sig/Slot
    //connect(ui->pushButton_thread_beginSerial,
SIGNAL(clicked()), locoserial, SLOT(do_run()));
    //connect(ui->pushButton_thread_beginSerial,
SIGNAL(clicked(bool)), ui->pushButton_thread_beginSerial,
```

```
SLOT(setEnabled(bool)));
    // Macros / locoSQL
    connect(locosql, &LocoSQL::incomingRequest, locoserial,
static_cast<void (LocoSerial::*)
(LocoPacket)>(&LocoSerial::do_writePacket));
    connect(this, &MainWindow::locosql_open, locosql,
&LocoSQL::do_openDB);
    connect(locosql, &LocoSQL::DBopened, this,
&MainWindow::handle_DBopened);
    connect(locosql, &LocoSQL::DBclosed, this,
&MainWindow::handle_DBclosed);
    connect(locosql, &LocoSQL::slotScan, locoserial,
&LocoSerial::do_slotScan);
    connect(locosql, &LocoSQL::slotDispatch, locoserial,
&LocoSerial::do_slotDispatch);
    connect(locosql, &LocoSQL::slotReq, locoserial,
&LocoSerial::do_slotReq);
    connect(locosql, &LocoSQL::slotUse, locoserial,
&LocoSerial::do_slotUse);
    connect(locosql, &LocoSQL::slotClear, locoserial,
&LocoSerial::do_slotClear);
    connect(locosql, &LocoSQL::trackReset, locoserial,
&LocoSerial::do_trackReset);
    connect(locosql, &LocoSQL::trackOn, locoserial,
&LocoSerial::do_trackOn);
    connect(locosql, &LocoSQL::trackOff, locoserial,
&LocoSerial::do_trackOff);
    connect(&threadUDP, &QThread::finished, locoudp,
&QObject::deleteLater);
    // Handle Initializing from Sig/Slot
    //connect(ui->pushButton_thread_beginSQL, SIGNAL(clicked()),
locosql, SLOT(do_run()));
    //connect(ui->pushButton_thread_beginSQL,
SIGNAL(clicked(bool)), ui->pushButton_thread_beginSQL,
SLOT(setEnabled(bool)));
    // Kickstart threads
    threadSerial.start();
    locoserial->do_run(); // Auto start locoserial
    threadSQL.start();
    locosql->do_run(); // Auto start locosql
    threadUDP.start();
    locoudp->do_run(); // Auto start locoudp
    // Configure interface
    ui->comboBox_opcodes->setEditable(false);
    ui->comboBox_opcodes-
>setInsertPolicy(QComboBox::InsertAtBottom);
    do_loadOPComboBox();
    do_refreshSerialList();
    if (debug) qDebug() << timeStamp() << "Interface loaded.";
}
MainWindow::~MainWindow()
{
    // Clean up sub-threads
    threadSerial.quit();
    threadSerial.wait();
    threadSQL.quit();
    threadSQL.wait();
    threadUDP.quit();
```

```
    threadUDP.wait();
    delete ui;
}
/*
 * MAINWINDOW METHODS
 */
QString MainWindow::timeStamp()
{
    return(QTime::currentTime().toString("[HH:mm:ss:zzz] "));
}
void MainWindow::do_enableArgs()
{
    LocoPacket _packet(ui->lineEdit_opcode->text());
    int _args = _packet.get_finalSize() - 2;
    if (_args > 0) {
        ui->lineEdit_arg1->setEnabled(true);
    } else {
        ui->lineEdit_arg1->setEnabled(false);
        ui->lineEdit_arg2->setEnabled(false);
        return;
    }
    if (_args > 1) {
        ui->lineEdit_arg2->setEnabled(true);
    } else {
        ui->lineEdit_arg2->setEnabled(false);
    }
}
void MainWindow::do_genPacket()
{
    QString _hex = "";
    outgoingPacket.set_allFromHex(ui->lineEdit_opcode->text());
    int _numArgs = outgoingPacket.get_finalSize() - 2;
    LocoPacket * _packet;
    _hex.append(ui->lineEdit_opcode->text());
    if (_numArgs > 0) {
        _hex.append(ui->lineEdit_arg1->text());
    } if (_numArgs > 1) {
        _hex.append(ui->lineEdit_arg2->text());
    }
    _packet = new LocoPacket(_hex);
    _packet->do_genChecksum();
    outgoingPacket = *_packet;
    ui->textBrowser_packets->append(timeStamp() + "Generated: " +
_packet->get_packet());
    ui->lineEdit_packet->setText(_packet->get_packet());
}
void MainWindow::do_loadOPComboBox()
{
    ui->comboBox_opcodes->clear();
    LocoPacket _tmp;
    QVector<QString> _opcodes = _tmp.get_DBopcodes();
    QVector<QString> _names = _tmp.get_DBnames();
    if (_opcodes.count() != _names.count())
    {
        qDebug() << timeStamp() << "database opcodes and names
don't match.";
        return;
    }
```

```
    for (int _index = 0; _index < _opcodes.count(); ++_index)
    {
        QString _text = _names[_index];
        _text.append(" [" + _opcodes[_index] + "]");
        ui->comboBox_opcodes->insertItem(_index, _text);
    }
}
void MainWindow::do_OPfromComboBox()
{
    LocoPacket _tmp;
    QVector<QString> _opcodes = _tmp.get_DBopcodes();
    QString _hex = _opcodes[ui->comboBox_opcodes->currentIndex()];
    ui->lineEdit_opcode->setText(_hex);
    do_enableArgs();
}
void MainWindow::do_refreshSerialList()
{
    QList<QSerialPortInfo> _ports = usbPorts.availablePorts();
    int _index = _ports.count();
    ui->comboBox_serialList->clear();
    for (int i = 0; i < _index; ++i)
    {
        ui->comboBox_serialList->insertItem(i,
_ports.at(i).portName());
        ui->textBrowser_console->append(_ports.at(i).portName());
    }
}
void MainWindow::do_packetReceived(LocoPacket _packet)
{
    if (debug) qDebug() << timeStamp() << "Reading packet to text
browser. " << _packet.get_packet();
    ui->textBrowser_console->append(timeStamp() +
_packet.get_packet());
    // Sort packets for easier reading
    QString _op = _packet.get_OPcode();
    if (_op == "B2")
    {
        ui->textBrowser_sorted_b2->append(timeStamp() +
_packet.get_packet());
    } else if (_op == "A0") {
        ui->textBrowser_sorted_a0->append(timeStamp() +
_packet.get_packet());
    } else if (_op == "E7") {
        ui->textBrowser_sorted_e7->append(timeStamp() +
_packet.get_packet());
    }
}
void MainWindow::do_bytesWritten(QByteArray _bytes)
{
    QString _byteText = QString::fromLatin1(_bytes.toHex());
    if (debug) qDebug() << timeStamp() << "Writing bytes to text
browser. " << _byteText;
    ui->textBrowser_packets->append(timeStamp() + "Written: " +
_byteText);
}
void MainWindow::do_printDescriptions(QString description)
{
    ui->textBrowser_console->append(timeStamp() + description);
```

```
}
/*
 * LOCOSERIAL METHODS
 */
void MainWindow::handle_serialOpened()
{
    ui->textBrowser_console->append(timeStamp() + "Serial port
opened x)");
    ui->pushButton_serialConnect->setEnabled(false);
    ui->pushButton_serialDisconnect->setEnabled(true);
    ui->comboBox_serialList->setEnabled(false);
    ui->pushButton_serialRefreshList->setEnabled(false);
}
void MainWindow::handle_serialClosed()
{
    ui->textBrowser_console->append(timeStamp() + "Serial port
closed :D");
    ui->pushButton_serialConnect->setEnabled(true);
    ui->pushButton_serialDisconnect->setEnabled(false);
    ui->comboBox_serialList->setEnabled(true);
    ui->pushButton_serialRefreshList->setEnabled(true);
}
void MainWindow::do_openSerial()
{
    int _portIndex = ui->comboBox_serialList->currentIndex();
    QSerialPortInfo _device =
usbPorts.availablePorts().at(_portIndex);
    emit locoserial_open(_device);
}
void MainWindow::do_sendSerial()
{
    outgoingPacket.set_allFromHex(ui->lineEdit_packet->text());
    if (!outgoingPacket.validChk())
    {
        qDebug() << timeStamp() << "Packet isn't right ~_~;";
        return;
    }
    ui->textBrowser_packets->append(timeStamp() + "Asking for
write: " + outgoingPacket.get_packet().toLatin1());
    // Only interact with a thread via Sig/Slots.
    emit locoserial_write(outgoingPacket);
    if (debug) qDebug() << timeStamp() << "Firing off to serial: "
<< outgoingPacket.get_packet().toLatin1();
    if (debug) qDebug() << timeStamp() <<
outgoingPacket.get_QByteArray() << outgoingPacket.get_QBitArray();
}
/*
 * SQL METHODS
 */
void MainWindow::handle_DBopened()
{
    ui->textBrowser_sql->append(timeStamp() + "Database opened.
Connection appears successful :)");
    ui->pushButton_connect->setEnabled(false);
    ui->lineEdit_database->setEnabled(false);
    ui->lineEdit_hostname->setEnabled(false);
    ui->lineEdit_password->setEnabled(false);
    ui->lineEdit_user->setEnabled(false);
```

```
    ui->spinBox_port->setEnabled(false);
    ui->pushButton_disconnect->setEnabled(true);
}
void MainWindow::handle_DBclosed()
{
    ui->textBrowser_sql->append(timeStamp() + "Database closed.");
    ui->pushButton_connect->setEnabled(true);
    ui->lineEdit_database->setEnabled(true);
    ui->lineEdit_hostname->setEnabled(true);
    ui->lineEdit_password->setEnabled(true);
    ui->lineEdit_user->setEnabled(true);
    ui->spinBox_port->setEnabled(true);
    ui->pushButton_disconnect->setEnabled(false);
}
void MainWindow::do_connectDB()
{
    // Collect information from window
    QString hostname = ui->lineEdit_hostname->text();
    int port = ui->spinBox_port->value();
    QString database = ui->lineEdit_database->text();
    QString username = ui->lineEdit_user->text();
    QString password = ui->lineEdit_password->text();
    emit locosql_open(hostname, port, database, username,
password);
}
#include "locoserial.h"
LocoSerial::LocoSerial()
{
    debug = NULL;
    incomingPacket = NULL;
    outgoingPacket = NULL;
}
LocoSerial::~LocoSerial()
{
    do_close();
    delete incomingPacket;
    delete outgoingPacket;
    delete debug;
}
QString LocoSerial::timeStamp()
{
    return(QTime::currentTime().toString("[HH:mm:ss:zzz] "));
}
void LocoSerial::do_run()
{
    incomingPacket = new LocoPacket;
    outgoingPacket = new LocoPacket;
    incomingPacket->clear();
    outgoingPacket->clear();
    debug = new bool;
    *debug = false;
    qDebug() << timeStamp() << "Serial thread initialized.";
}
void LocoSerial::do_writePacket(LocoPacket _packet)
{
    _packet.do_genChecksum();
    do_writeBytes(_packet.get_QByteArray());
    emit writtenPacket(_packet);
```

```
}
void LocoSerial::do_writeBytes(QByteArray _bytes)
{
    if (!usbBuffer) {
        return;
    }
    if (!usbBuffer->isOpen())
    {
        return;
    }
    if (*debug) qDebug() << timeStamp() << "Writing to serial: "
<< _bytes << _bytes.toInt(0, 16);
    usbBuffer->write(_bytes);
    emit writtenBytes(_bytes);
    outgoingPacket->clear();
    outgoingPacket->do_appendByteArray(_bytes);
    parse(*outgoingPacket);
}
void LocoSerial::do_querySlot(LocoByte _slot)
{
    QString _queryText = "BB" + _slot.get_hex() + "00";
    LocoPacket _querySlot(_queryText);
    do_writePacket(_querySlot); // the write() method will
generate a checksum for us
}
void LocoSerial::readTimerStart(int _msec)
{
    if (readTimer) {
        readTimerStop();
    }
    readTimer = new QTimer;
    connect(readTimer, SIGNAL(timeout()), this, SLOT(do_read()));
    readTimer->start(_msec);
}
void LocoSerial::readTimerStop()
{
    if (!readTimer) {
        return;
    }
    disconnect(readTimer, 0, 0, 0);
    readTimer->stop();
    readTimer->deleteLater();
}
void LocoSerial::do_close()
{
    readTimerStop();
    if (!usbBuffer) {
        return;
    }
    disconnect(usbBuffer, 0, 0, 0);
    usbBuffer->close();
    emit serialClosed();
    usbBuffer->deleteLater();
}
bool LocoSerial::do_open(QSerialPortInfo _port)
{
    usbBuffer = new QSerialPort;
    usbBuffer->setPort(_port);
```

```
    usbBuffer->setBaudRate(57600);
    usbBuffer->setFlowControl(QSerialPort::HardwareControl);
    //usbBuffer->open(QIODevice::ReadWrite);
    if (usbBuffer->open(QIODevice::ReadWrite)) //(usbBuffer-
>isOpen())
    {
        if (*debug) qDebug() << timeStamp() << "Serial port
appears to have opened sucessfully.";
        connect(usbBuffer, SIGNAL(readyRead()), this,
SLOT(do_read()));
        readTimerStart(20);
        emit serialOpened();
        return(true);
    }
    if (*debug) qDebug() << timeStamp() << "Serial port failed to
open.";
    usbBuffer->close();
    emit serialClosed();
    return(false);
}
void LocoSerial::do_read()
{
    if (!usbBuffer->isOpen())
    {
        return; // not open
    }
    // Read immediately if data is available
    while(usbBuffer->bytesAvailable() > 0)
    {
        if (*debug) qDebug() << timeStamp() << "Reading byte from
serial.";
        LocoByte _byte;
        _byte.set_fromByteArray(usbBuffer->read(1));
        if (!_byte.get_isOP() && incomingPacket->get_size() == 0)
        {
            if (*debug) qDebug() << timeStamp() << "Skipping non
opcode with empty incoming packet.";
            continue; // no need to assign anything but an OP to
position 0
        }
        if (_byte.get_isOP())
        {
            if (*debug) qDebug() << timeStamp() << "OP code: " <<
_byte.get_hex();
        }
        if (_byte.get_isOP() && (incomingPacket->get_size() > 0))
        {
            if (*debug) qDebug() << timeStamp() << "Dropping
packet to catch incoming op code.";
            emit droppedPacket();
            incomingPacket->clear();
        }
        incomingPacket->do_appendLocoByte(_byte);
        //incomingPacket->do_appendByteArray(usbBuffer->read(1));
// Load in a byte from the serial buffer
        if (*debug) qDebug() << timeStamp() << "packet:" <<
incomingPacket->get_size() << ":" << incomingPacket-
>get_finalSize();
```

```
        if (incomingPacket->get_size() == incomingPacket-
>get_finalSize() && incomingPacket->get_size() >= 2) // packet is
the right size
        {
            if (incomingPacket->validChk()) // packet has a valid
checksum, bingo!
            {
                if (*debug) qDebug() << timeStamp() << "Received a
full packet! " << incomingPacket->get_packet();
                emit receivedPacket(*incomingPacket);
                parse(*incomingPacket);
            } else {
                emit droppedPacket();
            }
            incomingPacket->clear(); // doesn't matter if packet
has valid checksum here, time to move to the next packet
            continue;
        }
        if (incomingPacket->get_size() > incomingPacket-
>get_finalSize() && incomingPacket->get_size() >= 2)
        {
            if (*debug) qDebug() << timeStamp() << "Dropping
packet for exceeding length stated by op code.";
            emit droppedPacket();
            incomingPacket->clear();
            continue;
        }
    }
}
/*
 * MACROS
 */
void LocoSerial::do_trackOn()
{
    LocoPacket _packet;
    _packet.set_allFromHex("837C");
    do_writePacket(_packet);
}
void LocoSerial::do_trackOff()
{
    LocoPacket _packet;
    _packet.set_allFromHex("827D");
    do_writePacket(_packet);
}
void LocoSerial::do_trackReset()
{
    do_trackOff();
    QTimer::singleShot(10000, this, SLOT(do_trackOn()));
}
void LocoSerial::do_slotScan(LocoByte _slot)
{
    if (_slot.get_decimal() < 1 || _slot.get_decimal() > 119)
    {
        // Invalid slot #
        return;
    }
    LocoPacket _packet;
    //_slot.set_fromHex(QString("%1").arg(_index, 2, 16,
```

```
QChar('0')));
    _packet.do_appendByte("BB"); // OP code
    _packet.do_appendLocoByte(_slot);
    _packet.do_appendByte("00");
    _packet.do_genChecksum();
    do_writePacket(_packet);
}
void LocoSerial::do_slotClear(LocoByte _slot)
{
    LocoPacket _packet;
    _packet.do_appendByte("B5"); // OP code
    _packet.do_appendLocoByte(_slot);
    _packet.do_appendByte("03");
    _packet.do_genChecksum();
    do_writePacket(_packet);
    //QTimer::singleShot(20, this, SLOT(do_slotScan(_slot)));
    //do_slotScan(_slot);
}
void LocoSerial::do_slotDispatch(LocoByte _slot)
{
    LocoPacket _packet;
    _packet.do_appendByte("BA"); // OP code
    _packet.do_appendLocoByte(_slot);
    _packet.do_appendByte("00");
    _packet.do_genChecksum();
    do_writePacket(_packet);
    //QTimer::singleShot(20, this, SLOT(do_slotScan(_slot)));
    //do_slotScan(_slot);
}
void LocoSerial::do_slotReq(LocoByte _adr)
{
    LocoPacket _packet;
    _packet.do_appendByte("BF"); // OP code
    _packet.do_appendByte("00");
    _packet.do_appendLocoByte(_adr);
    _packet.do_genChecksum();
    do_writePacket(_packet);
}
void LocoSerial::do_slotUse(LocoByte _slot)
{
    LocoPacket _packet;
    _packet.do_appendByte("BA"); // OP code
    _packet.do_appendLocoByte(_slot);
    _packet.do_appendLocoByte(_slot);
    _packet.do_genChecksum();
    do_writePacket(_packet);
}
/*
 * Packet related functions
 */
QString LocoSerial::parse (LocoPacket _packet)
{
    QString _opCode = (_packet.get_OPcode());
    if (!_packet.hasOP() || !_packet.validChk()) {
        return("packet is malformed >.<");
    }
    // This looks so gross >.<
    if (_opCode == "E7") {
```

```
        return(parse_E7(_packet));
    } else if (_opCode == "B2") {
        return(parse_B2(_packet));
    } else if (_opCode == "85") {
        return(parse_85(_packet));
    } else if (_opCode == "83") {
        return(parse_83(_packet));
    } else if (_opCode == "82") {
        return(parse_82(_packet));
    } else if (_opCode == "81") {
        return(parse_81(_packet));
    } else if (_opCode == "BF") {
        return(parse_BF(_packet));
    } else if (_opCode == "BD") {
        return(parse_BD(_packet));
    } else if (_opCode == "BC") {
        return(parse_BC(_packet));
    } else if (_opCode == "BB") {
        return(parse_BB(_packet));
    } else if (_opCode == "BA") {
        return(parse_BA(_packet));
    } else if (_opCode == "B9") {
        return(parse_B9(_packet));
    } else if (_opCode == "B8") {
        return(parse_B8(_packet));
    } else if (_opCode == "B6") {
        return(parse_B6(_packet));
    } else if (_opCode == "B5") {
        return(parse_B5(_packet));
    } else if (_opCode == "B4") {
        return(parse_B4(_packet));
    } else if (_opCode == "B1") {
        return(parse_B1(_packet));
    } else if (_opCode == "B0") {
        return(parse_B0(_packet));
    } else if (_opCode == "A2") {
        return(parse_A2(_packet));
    } else if (_opCode == "A1") {
        return(parse_A1(_packet));
    } else if (_opCode == "A0") {
        return(parse_A0(_packet));
    } else if (_opCode == "EF") {
        return(parse_EF(_packet));
    } else if (_opCode == "E5") {
        return(parse_E5(_packet));
    } else if (_opCode == "ED") {
        return(parse_ED(_packet));
    }
    return ("opcode [" + _opCode + "] doesn't match anything in
parser :c");
}
QString LocoSerial::parse_E7 (LocoPacket _packet)
{
    QString _description = "E7:";
    // Parse packet into usable variables
    bool _busy = _packet.get_locobyte(3).get_oneBit(2);
    bool _active = _packet.get_locobyte(3).get_oneBit(3);
    LocoByte _slot = _packet.get_locobyte(2);
```

```
    LocoByte _adr = _packet.get_locobyte(4);
    LocoByte _speed = _packet.get_locobyte(5);
    bool _dir = _packet.get_locobyte(6).get_oneBit(2);
    LocoTrain _newTrain;
    _newTrain.set_adr(_adr);
    _newTrain.set_reverse(_dir);
    _newTrain.set_slot(_slot);
    _newTrain.set_speed(_speed);
    QString _state = "NULL";
    if (_busy && _active) {
        _state = "IN_USE";
    } else if (_busy && !_active) {
        _state = "IDLE";
    } else if (!_busy && _active) {
        _state = "COMMON";
    } else {
        _state = "FREE";
    }
    _newTrain.set_state(_state);
    emit trainUpdated(_newTrain);
    _description.append(" Speed: " + _speed.get_hex() + " Slot: "
+ _slot.get_hex() + " Direction: " + QString::number(_dir)+"
State: "+_state);
    if (_description == "E7:") {
        _description.append(" No action taken?");
    }
    return (_description);
}
QString LocoSerial::parse_EF (LocoPacket _packet) {
    QString _description = "EF: Write slot data.";
    return(_description);
}
QString LocoSerial::parse_E5 (LocoPacket _packet) {
    QString _description = "E5: Move 8 bytes Peer to Peer.";
    return(_description);
}
QString LocoSerial::parse_ED (LocoPacket _packet) {
    QString _description = "ED: Send n-byte packet.";
    return(_description);
}
QString LocoSerial::parse_85 (LocoPacket _packet) {
    QString _description = "85: Requesting track state IDLE / EMG
STOP.";
    return(_description);
}
QString LocoSerial::parse_83 (LocoPacket _packet) {
    QString _description = "83: Requesting track state ON.";
    return(_description);
}
QString LocoSerial::parse_82 (LocoPacket _packet) {
    QString _description = "82: Requesting track state OFF.";
    return(_description);
}
QString LocoSerial::parse_81 (LocoPacket _packet) {
    QString _description = "81: MASTER sent BUSY code.";
    return(_description);
}
QString LocoSerial::parse_BF (LocoPacket _packet) {
```

```
    QString _description = "BF: Requesting locomotive address.";
    return(_description);
}
QString LocoSerial::parse_BD (LocoPacket _packet) {
    QString _description = "BD: Requesting switch with LACK
function."; // LACK - Long ACKnowledge
    return(_description);
}
QString LocoSerial::parse_BC (LocoPacket _packet) {
    QString _description = "BC: Requesting state of switch.";
    return(_description);
}
QString LocoSerial::parse_BB (LocoPacket _packet) {
    QString _description = "BB: Requesting SLOT data/status
block.";
    return(_description);
}
QString LocoSerial::parse_BA (LocoPacket _packet) {
    QString _description = "BA: Move slot SRC to DEST.";
    return(_description);
}
QString LocoSerial::parse_B9 (LocoPacket _packet) {
    QString _description = "B9: Link slot ARG1 to slot ARG2.";
    return(_description);
}
QString LocoSerial::parse_B8 (LocoPacket _packet) {
    QString _description = "B8: Unlink slot ARG1 from slot ARG2.";
    return(_description);
}
QString LocoSerial::parse_B6 (LocoPacket _packet) {
    QString _description = "B6: Set FUNC bits in a CONSIST uplink
element.";
    return(_description);
}
QString LocoSerial::parse_B5 (LocoPacket _packet) {
    QString _description = "B5: Write slot stat1.";
    return(_description);
}
QString LocoSerial::parse_B4 (LocoPacket _packet) {
    QString _description = "B4: Long Acknowledge - LACK.";
    return(_description);
}
QString LocoSerial::parse_B2 (LocoPacket _packet)
{
    QString _description = "B2:";
    LocoByte _arg1 = _packet.get_locobyte(1);
    LocoByte _arg2 = _packet.get_locobyte(2);
    bool _aux = _arg2.get_qBitArray()[2];
    bool _occupied = _arg2.get_qBitArray()[3];
    LocoByte _adr1 = QBitArray(8, 0);
    LocoByte _adr2 = QBitArray(8, 0);
    QBitArray _arg1Bits = _arg1.get_qBitArray();
    QBitArray _arg2Bits = _arg2.get_qBitArray();
    for (int _index = 4; _index < 7; ++_index)
    {
        if (_arg2Bits.at(_index) == 1)
        {
            _adr1.set_oneBit(_index+1, true); // Load MS nyble of
```

```
address
        } else {
            _adr1.set_oneBit(_index+1, false); // Load MS nyble of
address
        }
    }
    _adr2.set_oneBit(0, _arg2Bits.at(7));
    for (int _index = 1; _index < 8; ++_index)
    {
        if (_arg1Bits.at(_index) == 1)
        {
            _adr2.set_oneBit(_index, true); // Load LS byte of
address
        } else {
            _adr2.set_oneBit(_index, false); // Load LS byte of
address
        }
    }
    QString _address = _adr1.get_hex().mid(1,1) + _adr2.get_hex();
    LocoBlock _newBlock(_address, _aux, _occupied);
    emit blockUpdated(_newBlock);
    _description.append(" AUX: " + QString::number(_aux) + " OCC:
" + QString::number(_occupied) + ".");
    return(_description);
}
QString LocoSerial::parse_B1 (LocoPacket _packet) {
    QString _description = "B1: Turnout sensor state report.";
    return(_description);
}
QString LocoSerial::parse_B0 (LocoPacket _packet) {
    QString _description = "B0:";
    LocoByte _arg1 = _packet.get_locobyte(1);
    LocoByte _arg2 = _packet.get_locobyte(2);
    bool _state = _arg2.get_qBitArray()[2];
    QByteArray _adr;
    _adr.append(_arg2.get_hex().mid(1,1)); // Load MS byte of
address
    _adr.append(_arg1.get_hex()); // Load LS 2 bytes of address
    emit switchUpdated(_adr.toInt(0, 10), _state);
    return(_description);
}
QString LocoSerial::parse_A2 (LocoPacket _packet) {
    QString _description = "A2: Setting slot sound functions.";
    return(_description);
}
QString LocoSerial::parse_A1 (LocoPacket _packet) {
    QString _description = "A1: Setting slot direction.";
    return(_description);
}
QString LocoSerial::parse_A0 (LocoPacket _packet) {
    QString _description = "A0: Setting slot speed.";
    LocoByte _arg1 = _packet.get_locobyte(1);
    LocoByte _arg2 = _packet.get_locobyte(2);
    do_querySlot(_arg1); // Ask for E7 slot data
    return(_description);
}
```

```
#include "locosql.h"
/*
 * SQL Constants
 */
const QString schema = "`cpe453`"; // Schema for all tables.
const QString reqMacro = "`req_macro`"; // Table name for macro
requests.
const QString reqTrain = "`req_train`"; // Table name for
train/slot requests.
const QString reqSwitch = "`req_switch`"; // Table name for switch
position requests.
const QString reqPacket = "`req_packet`"; // Table name for raw
packet requests.
const QString trackTrain = "`track_train`"; // Table name for list
of scanned trains/slots.
const QString trackBlock = "`track_ds`"; // Table name for list of
Detection Sections
const QString trackSwitch = "`track_switch`"; // Table name for
list of Detection Sections
LocoSQL::LocoSQL()
{
    debug = NULL;
    doDelete = NULL;
    mainDB = NULL;
    mainQuery = NULL;
    reqIndex = NULL;
    reqTimer = NULL;
}
LocoSQL::~LocoSQL()
{
    reqTimerStop();
    if (mainDB != NULL)
    {
        mainDB->close();
        delete mainDB;
        delete mainQuery;
    }
    delete debug;
    delete doDelete;
}
QString LocoSQL::timeStamp()
{
    return(QTime::currentTime().toString("[HH:mm:ss:zzz] "));
}
void LocoSQL::do_run()
{
    mainDB = new QSqlDatabase;
    *mainDB = QSqlDatabase::addDatabase("QMYSQL", "main");
    reqIndex = new int;
    debug = new bool;
    *debug = true;
    doDelete = new bool;
    *doDelete = true;
    qDebug() << timeStamp() << "SQL thread initialized.";
}
bool LocoSQL::do_openDB(QString hostname, int port, QString
database, QString username, QString password)
{
```

```
    mainDB->setHostName(hostname);
    mainDB->setPort(port);
    mainDB->setDatabaseName(database);
    mainDB->setUserName(username);
    mainDB->setPassword(password);
    // Attempt to open database
    if (!mainDB->open())
    {
        qDebug() << timeStamp() << "Opening postgresql database
failed D:";
        qDebug() << timeStamp() << mainDB->lastError();
        return(false);
    }
    // Setup query
    mainQuery = new QSqlQuery;
    *mainQuery = QSqlQuery(*mainDB);
    // Clear status tables on startup
    if (*doDelete) {
        do_clearAllTables();
    }
    // At 80ms, generates ~15KB/s of traffic to SQL server.
    // This value can be deprecated by hosting train_rec and SQL
on the same machine.
    reqTimerStart(140); // Wait Xms between SQL scans
    emit DBopened();
    return(true);
}
void LocoSQL::do_closeDB()
{
    reqTimerStop();
    if (mainDB == NULL || mainQuery == NULL)
    {
        return;
    }
    if (mainDB->isOpen())
    {
        if (mainQuery != NULL)
        {
            delete mainQuery;
            mainQuery = NULL;
        }
        mainDB->close();
    }
    emit DBclosed();
}
void LocoSQL::reqTimerStart(int _msec)
{
    if (reqTimer)
    {
        reqTimerStop();
    }
    reqTimer = new QTimer;
    connect(reqTimer, SIGNAL(timeout()), this,
SLOT(do_cycleReqs()));
    reqTimer->start(_msec);
}
void LocoSQL::reqTimerStop()
{
```

```
    if (!reqTimer) {
        return;
    }
    disconnect(reqTimer, 0, 0, 0);
    reqTimer->stop();
    reqTimer->deleteLater();
}
void LocoSQL::do_clearAllTables()
{
    //do_clearTable("track_ds"); // Don't clear if DS are entered
beforehand
    do_clearTable("track_train");
}
void LocoSQL::do_clearTable(QString _table)
{
    if (mainDB == NULL || mainQuery == NULL) {
        return;
    }
    if (mainDB->isOpen()) {
        _table.prepend("DELETE FROM cpe453.");
        mainQuery->prepare(_table);
        mainQuery->bindValue(":table", _table);
        if (*debug) qDebug() << timeStamp() << "Deleting all rows
in SQL cpe453." << _table;
        if (!mainQuery->exec())
        {
            qDebug() << timeStamp() << mainQuery->lastError();
        }
    }
}
int LocoSQL::get_percentFromHex(QString _hex) {
    if (*debug) qDebug() << timeStamp() << "percentfromhex: " <<
_hex;
    int _percent = _hex.toInt(0, 16);
    _percent = _percent*0.8;
    if (_percent < 0) {
        _percent = 0;
    } else if (_percent > 100) {
        _percent = 100;
    }
    if (*debug) qDebug() << timeStamp() << "get_percentFromHex():
" << _percent;
    return(_percent);
}
QString LocoSQL::get_hexFromPercent(int _percent) {
    if (_percent < 2) {
        return("00");
    }
    _percent = ceil(_percent*1.25);
    if (*debug) qDebug() << timeStamp() << "Percent " << _percent;
    QString _hex =  QString("%1").arg(_percent, 2, 16,
QChar('0')); //QString::number(_percent, 16);
    if (*debug) qDebug() << timeStamp() << "get_hexFromPercent():
" << _hex;
    get_percentFromHex(_hex);
    return(_hex);
}
QString LocoSQL::get_hexFromInt(int _adr) {
```

```
    QString _hex =  QString("%1").arg(_adr, 2, 16,
QChar('0'));//QString::number(_adr, 16);
    if (*debug) qDebug() << timeStamp() << "get_hexFromInt(): " <<
_hex;
    return(_hex);
}
void LocoSQL::do_cycleReqs()
{
    switch(*reqIndex)
    {
    case 0:
        do_reqTrain();
        *reqIndex = 1;
        break;
    case 1:
        do_reqPacket();
        *reqIndex = 2;
        break;
    case 2:
        do_reqSwitch();
        *reqIndex = 3;
        break;
    case 3:
        do_reqMacro();
        *reqIndex = 0;
        break;
    default:
        *reqIndex = 0;
        break;
    }
}
void LocoSQL::do_reqMacro()
{
    if (*debug) qDebug() << timeStamp() << "Querying for macro
requests.";
    if (mainDB == NULL || mainQuery == NULL) {
        return;
    }
    if (!mainDB->isOpen())
    {
        return;
    }
    mainQuery->prepare("SELECT * FROM "+schema+"."+reqMacro+";");
    mainQuery->exec();
    if (mainQuery->next())
    {
        QString _macro = mainQuery->value("macro").toString();
        int _id = mainQuery->value("id").toInt();
        if (_macro == "SLOT_SCAN") { // arg1 = slot #
            if (debug) qDebug() << timeStamp() << "Scanning
slot.";
            LocoByte _arg1 = get_hexFromInt(mainQuery-
>value("arg1").toInt());
            emit slotScan(_arg1);
        } else if (_macro == "SLOT_SCAN_ALL") { // no args
            if (debug) qDebug() << timeStamp() << "Not
Implemented";
        } else if (_macro == "SLOT_DISPATCH") { // arg1 = slot #
```

```
            if (debug) qDebug() << timeStamp() << "Dispatching
slot.";
            LocoByte _arg1 = get_hexFromInt(mainQuery-
>value("arg1").toInt());
            emit slotDispatch(_arg1);
        } else if (_macro == "SLOT_DISPATCH_ALL") {
            if (debug) qDebug() << timeStamp() << "Not
Implemented";
        } else if (_macro == "SLOT_CLEAR") { // arg1 = slot
            if (debug) qDebug() << timeStamp() << "Calling for
track reset.";
            LocoByte _arg1 = get_hexFromInt(mainQuery-
>value("arg1").toInt());
            emit slotClear(_arg1);
        } else if (_macro == "SLOT_CLEAR_ALL") {
            if (debug) qDebug() << timeStamp() << "Not
Implemented";
        } else if (_macro == "SLOT_REQ") { // arg1 = train
            if (debug) qDebug() << timeStamp() << "Requesting
slot";
            LocoByte _arg1 = get_hexFromInt(mainQuery-
>value("arg1").toInt());
            emit slotReq(_arg1);
        } else if (_macro == "SLOT_USE") { // arg1 = slot
            if (debug) qDebug() << timeStamp() << "Setting slot to
IN_USE.";
            LocoByte _arg1 = get_hexFromInt(mainQuery-
>value("arg1").toInt());
            emit slotUse(_arg1);
        } else if (_macro == "TRACK_RESET") {
            if (debug) qDebug() << timeStamp() << "Calling for
track reset.";
            emit trackReset();
        } else if (_macro == "TRACK_ON") {
            if (debug) qDebug() << timeStamp() << "Calling for
track on.";
            emit trackOn();
        } else if (_macro == "TRACK_OFF") {
            if (debug) qDebug() << timeStamp() << "Calling for
track off.";
            emit trackOff();
        } else if (_macro == "TRAIN_REQ") {
            if (debug) qDebug() << timeStamp() << "Calling for
train req.";
            // Local variables
            int _adr = mainQuery->value("arg1").toInt();
            int _speed = mainQuery->value("arg2").toInt();
            int _dir = 0;
            QVector<int> _slots;
            // Set dir
            if (_speed < 0) {
                _dir = 1;
                _speed = abs(_speed);
            }
            // Request a slot for the given train address
            mainQuery->prepare("INSERT INTO
"+schema+"."+reqMacro+" (`macro`, `arg1`)"
                               "VALUES ('SLOT_REQ',
```

```
'"+_adr+"');");
            mainQuery->exec();
            // Find slots associated with train address
            mainQuery->prepare("SELECT * FROM
"+schema+"."+trackTrain+" WHERE adr=:_adr;");
            mainQuery->bindValue(":_adr", _adr);
            mainQuery->exec();
            // Build list of associated slots
            while (mainQuery->next())
            {
                int _slot = mainQuery->value("slot").toInt();
                _slots.push_back(_slot);
            }
            // Iterate to scan slots and push requests
            while (!_slots.isEmpty())
            {
                mainQuery->prepare("INSERT INTO
"+schema+"."+reqMacro+" (`macro`, `arg1`)"
                                   "VALUES ('SLOT_USE',
'"+QString::number(_slots.first())+"');");
                mainQuery->exec();
                mainQuery->prepare("INSERT INTO
"+schema+"."+reqMacro+" (`macro`, `arg1`)"
                                   "VALUES ('SLOT_SCAN',
'"+QString::number(_slots.first())+"');");
                mainQuery->exec();
                mainQuery->prepare("INSERT INTO
"+schema+"."+reqTrain+" (`slot`, `speed`, `dir`)"
                                   "VALUES (:slot, :speed,
:dir);");
                mainQuery->bindValue(":slot",
QString::number(_slots.takeFirst()));
                mainQuery->bindValue(":speed", _speed);
                mainQuery->bindValue(":dir", _dir);
                mainQuery->exec();
            }
        }
        if (*doDelete)
        {
            if (debug) qDebug() << timeStamp() << "deleting macro
id: " << _id;
            mainQuery->prepare("DELETE FROM
"+schema+"."+reqMacro+" WHERE id=:_id LIMIT 1;");
            mainQuery->bindValue(":_id", _id);
            mainQuery->exec();
        }
    }
}
void LocoSQL::do_reqSwitch()
{
    if (*debug) qDebug() << timeStamp() << "Querying for switch
requests.";
    if (mainDB == NULL || mainQuery == NULL) {
        return;
    }
    if (!mainDB->isOpen())
    {
        return;
```

```
    }
    mainQuery->prepare("SELECT * FROM "+schema+"."+reqSwitch+";");
    mainQuery->exec();
    if (mainQuery->next())
    {
        LocoPacket _packet;
        LocoByte _command;
        LocoByte _id;
        int _idInt;
        LocoByte _position;
        int _positionInt;
        _command.set_fromHex("B0");
        _idInt = mainQuery->value("id").toInt();
        _id.set_fromHex(get_hexFromInt(_idInt-1));
        _positionInt = mainQuery->value("position").toInt();

_position.set_fromHex(get_hexFromInt((((_positionInt)*2)+1)*16));
        _packet.do_appendLocoByte(_command);
        _packet.do_appendLocoByte(_id);
        _packet.do_appendLocoByte(_position);
        _packet.do_genChecksum();
        if (*debug) qDebug() << timeStamp() << "Found switch
request: " << _packet.get_packet();
        emit incomingRequest(_packet);
        if (*doDelete)
        {
            mainQuery->prepare("DELETE FROM
"+schema+"."+reqSwitch+" WHERE id=:_id LIMIT 1;");
            mainQuery->bindValue(":_id", _idInt);
            mainQuery->exec();
        }
    }
}
void LocoSQL::do_reqTrain()
{
    if (*debug) qDebug() << timeStamp() << "Querying for train
requests.";
    if (mainDB == NULL || mainQuery == NULL) {
        return;
    }
    if (!mainDB->isOpen())
    {
        // open
        return;
    }
    mainQuery->prepare("SELECT * FROM "+schema+"."+reqTrain+";");
    mainQuery->exec();
    if (mainQuery->next())
    {
        LocoByte _command;
        LocoPacket _speedPacket;
        LocoPacket _dirPacket;
        LocoByte _slot;
        QString _slotString;
        int _slotInt;
        LocoByte _speed;
        int _speedInt;
        int _dir;
```

```
        _command.set_fromHex("A0");
        _speed.set_fromHex(get_hexFromPercent(mainQuery-
>value("speed").toInt()));
        _speedInt = mainQuery->value("speed").toInt();
        _slotString = mainQuery->value("slot").toString();
        _slotInt = mainQuery->value("slot").toInt();
        _slot.set_fromHex(get_hexFromInt(_slotInt));
        _dir = mainQuery->value("dir").toBool();
        _speedPacket.do_appendLocoByte(_command);
        _speedPacket.do_appendLocoByte(_slot);
        _speedPacket.do_appendLocoByte(_speed);
        _speedPacket.do_genChecksum();
        emit incomingRequest(_speedPacket);
        if (*debug) qDebug() << timeStamp() << "Found speed
request. " << _speedPacket.get_packet();
        _command.set_fromHex("A1");
        _dirPacket.do_appendLocoByte(_command);
        _dirPacket.do_appendLocoByte(_slot);
        _dirPacket.do_appendByte(QString::number(((_dir)?
2:0)+1)+"0");
        emit incomingRequest(_dirPacket);
        if (*debug) qDebug() << timeStamp() << "Found dir request.
" << _dirPacket.get_packet();
        if (*doDelete)
        {
            mainQuery->prepare("DELETE FROM
"+schema+"."+reqTrain+" WHERE slot=:_slot LIMIT 1;");
            mainQuery->bindValue(":_slot", _slotString);
            mainQuery->exec();
        }
        // Request from teams, update train status table
immediately on output
        mainQuery->prepare("INSERT INTO "+schema+"."+reqMacro+"
(`macro`, `arg1`)"
                           "VALUES ('SLOT_SCAN',
'"+_slotString+"');");
        mainQuery->exec();
    }
}
void LocoSQL::do_reqPacket()
{
    if (*debug) qDebug() << timeStamp() << "Querying for packet
requests.";
    if (mainDB == NULL || mainQuery == NULL) {
        return;
    }
    if (!mainDB->isOpen())
    {
        return;
    }
    mainQuery->prepare("SELECT * FROM "+schema+"."+reqPacket+";");
    mainQuery->exec();
    if (mainQuery->next())
    {
        LocoPacket _packet(mainQuery->value("packet").toString());
        int _id = mainQuery->value("id").toInt();
        if (_packet.validOP())
        {
```

```
            if (*debug) qDebug() << timeStamp() << "Found packet
request: " << _packet.get_packet();
            emit incomingRequest(_packet);
        }
        if (*doDelete)
        {
            mainQuery->prepare("DELETE FROM
"+schema+"."+reqPacket+" WHERE id=:_id LIMIT 1;");
            mainQuery->bindValue(":_id", _id);
            mainQuery->exec();
        }
    }
}
/*
 * Status updating methods
 */
/**
 * @brief LocoSQL::do_updateBlock
 * @param _block
 */
void LocoSQL::do_updateBlock(LocoBlock _block)
{
    if (mainDB == NULL || mainQuery == NULL) {
        return;
    }
    if (mainDB->isOpen()) {
        // This query will insert a DS if it isn't already in the
table
        /*mainQuery->prepare("INSERT INTO
"+schema+"."+trackBlock+" (id, status) "
                           "VALUES (:id, :status) "
                           "ON DUPLICATE KEY "
                           "UPDATE status=:status;");*/
        // This query will ignore DS which are not listed in the
table
        mainQuery->prepare("UPDATE "+schema+"."+trackBlock+" SET
`status`=:status WHERE `id`=:id;");
        QString _id = QString::number(_block.get_board())
+"-"+QString::number(_block.get_ds());
        int _status = _block.get_occupied();
        mainQuery->bindValue(":id", _id);
        mainQuery->bindValue(":status", _status);
        if (*debug) qDebug() << timeStamp() << "Updating SQL
block: " << _id << ":" << _status;
        mainQuery->exec();
    }
}
/**
 * @brief LocoSQL::do_updateTrain
 * @param _train
 */
void LocoSQL::do_updateTrain (LocoTrain _train)
{
    if (mainDB == NULL || mainQuery == NULL) {
        return;
    }
    if (mainDB->isOpen()) {
        QString _adr = _train.get_adr().get_hex();
```

```
        QString _slot = _train.get_slot().get_hex();
        int _speed =
get_percentFromHex(_train.get_speed().get_hex());
        int _dir = _train.get_direction()?1:0;
        QString _state = _train.get_state();
        mainQuery->prepare("INSERT INTO "+schema+"."+trackTrain+"
(slot, adr, speed, dir, state) "
                           "VALUES (:slot, :adr, :speed, :dir,
:state) "
                           "ON DUPLICATE KEY "
                           "UPDATE adr=:adr, speed=:speed,
dir=:dir, state=:state;");
        mainQuery->bindValue(":slot", _slot);
        mainQuery->bindValue(":adr", _adr);
        mainQuery->bindValue(":speed", _speed);
        mainQuery->bindValue(":dir", _dir);
        mainQuery->bindValue(":state", _state);
        if (*debug) qDebug() << timeStamp() << "Updating train
SQL." << _slot << ":" << _speed;
        mainQuery->exec();
    }
}
void LocoSQL::do_updateSwitch(int _adr, bool _state)
{
    if (mainDB == NULL || mainQuery == NULL) {
        return;
    }
    if (mainDB->isOpen()) {
        //QString _address = QString::number(_adr);
        mainQuery->prepare("INSERT INTO "+schema+"."+trackSwitch+"
(adr, state) "
                           "VALUES (:adr, :state) "
                           "ON DUPLICATE KEY "
                           "UPDATE state=:state;");
        mainQuery->bindValue(":adr", _adr);
        mainQuery->bindValue(":state", _state);
        if (*debug) qDebug() << timeStamp() << "Updating switch
SQL." << _adr << ":" << _state;
        mainQuery->exec();
        //
    }
}
#include "locopacket.h"
/* Class: LocoPacket
 *
 * For creating, manipulating, and extracting LocoNet Packets
 *
 * Christopher Bero [csb0019@uah.edu]
 * Team 4A
 */
/* Static Members */
bool LocoPacket::debug = false;
QSqlDatabase LocoPacket::packetDB =
QSqlDatabase::addDatabase("QSQLITE");
/*
 * Default Constructor
 */
LocoPacket::LocoPacket()
```

```
{
    clear();
    do_openDB();
}
/*
 * Overloaded Constructor
 */
LocoPacket::LocoPacket(QString _hex)
{
    _hex = _hex.remove(" ");
    _hex = _hex.remove(":");
    if ((_hex.count()%2) != 0)
    {
        qDebug() << timeStamp() << "Hex packet is malformed! D:";
        locobyte_array.clear();
        return;
    }
    clear();
    set_allFromHex(_hex);
    do_openDB();
}
LocoPacket::LocoPacket(QByteArray _bytearray)
{
    clear();
    do_appendByteArray(_bytearray);
    do_openDB();
}
/*
 * Default Destructor
 */
LocoPacket::~LocoPacket()
{
    //
}
QString LocoPacket::timeStamp()
{
    return(QTime::currentTime().toString("[HH:mm:ss:zzz] "));
}
void LocoPacket::clear()
{
    locobyte_array.clear();
}
bool LocoPacket::do_openDB()
{
    if (packetDB.isOpen())
    {
        return(true);
    }
    packetDB.setDatabaseName("packets.sqlite");
    if (!packetDB.open())
    {
        qDebug() << timeStamp() << packetDB.lastError();
        qDebug() << timeStamp() << "Unable to open packets sqlite
database.";
        return(false);
    }
    if (debug) qDebug() << timeStamp() << "Opened packet
database.";
```

```
    return(true);
}
void LocoPacket::do_closeDB()
{
    if (debug) qDebug() << timeStamp() << "Closed packet
database.";
    packetDB.close();
}
/* set_allFromHex()
 *
 * Define binary, hex, and bits from a string
 */
void LocoPacket::set_allFromHex(QString _hex)
{
    _hex = _hex.remove(" ");
    _hex = _hex.remove(":");
    _hex = _hex.remove("-");
    _hex = _hex.remove(",");
    if ((_hex.count()%2) != 0)
    {
        if (debug) qDebug() << timeStamp() << "Hex packet is
malformed! D:";
        return;
    }
    int _length = _hex.count() / 2;
    locobyte_array.clear();
    for (int _packet = 0; _packet < _length; ++_packet)
    {
        LocoByte _tmp_locohex(_hex.mid((_packet * 2),2));
        locobyte_array.append(_tmp_locohex);
        if (debug) qDebug() << timeStamp() << "New packet: " <<
locobyte_array[_packet].get_hex() << " " <<
locobyte_array[_packet].get_binary();
    }
    if (hasOP())
    {
        if (debug) qDebug() << timeStamp() << "Valid OP code x)";
    }
    if (validChk())
    {
        if (debug) qDebug() << timeStamp() << "Valid Checksum x)";
    }
}
bool LocoPacket::operator ==(LocoPacket _arg)
{
    return(this->get_QByteArray() == _arg.get_QByteArray());
}
QString LocoPacket::get_packet()
{
    QString _result = "";
    for (int _index = 0; _index < locobyte_array.size(); ++_index)
    {
        _result.append(locobyte_array[_index].get_hex());
    }
    return (_result);
}
int LocoPacket::get_finalSize()
{
```

```
    int _result = -1;
    if (locobyte_array.count() >= 1)
    {
        _result = (locobyte_array[0].get_packetLength());
    }
    if (_result == 0 && locobyte_array.count() >= 2)
    {
        // N byte packet, check next section for count
        _result = (locobyte_array[1].get_packetLength());
    }
    return(_result);
}
int LocoPacket::get_size ()
{
    return(locobyte_array.count());
}
QString LocoPacket::get_OPcode ()
{
    if (locobyte_array.count() < 1)
    {
        return("00");
    }
    return(locobyte_array.first().get_hex());
}
LocoByte LocoPacket::get_locobyte (int _byte)
{
    if (_byte >= locobyte_array.count())
    {
        return LocoByte("FF");
    }
    return locobyte_array[_byte];
}
QString LocoPacket::do_xor(LocoByte _byte1, LocoByte _byte2)
{
    if (debug) qDebug() << timeStamp() << "doXor() ";
    LocoByte _result;
    for (short unsigned int _bit = 0; _bit < 8; ++_bit)
    {
        if (debug) qDebug() << timeStamp() << "doxor: " << _bit <<
" - " << _byte1.get_oneBit(_bit) << " : " <<
_byte2.get_oneBit(_bit);
        if (_byte1.get_oneBit(_bit) == 0 &&
_byte2.get_oneBit(_bit) == 1) {
            _result.set_oneBit(_bit, 1);
        } else if (_byte1.get_oneBit(_bit) == 1 &&
_byte2.get_oneBit(_bit) == 0) {
            _result.set_oneBit(_bit, 1);
        } else if (_byte1.get_oneBit(_bit) == 1 &&
_byte2.get_oneBit(_bit) == 1) {
            _result.set_oneBit(_bit, 0);
        } else if (_byte1.get_oneBit(_bit) == 0 &&
_byte2.get_oneBit(_bit) == 0) {
            _result.set_oneBit(_bit, 0);
        }
    }
    if (debug) qDebug() << timeStamp() << "end doXor: hex: " <<
_result.get_hex();
    return (_result.get_hex());
```

```
}
QString LocoPacket::do_genChecksum()
{
    if (validChk() && hasOP())
    {
        if (debug) qDebug() << timeStamp() << "Not generating a
checksum for an already valid packet ^-^";
        QString _chk = "";
        _chk.append(locobyte_array[locobyte_array.size()-
1].get_hex());
        return(_chk);
    }
    LocoByte _checksum;
    for (int _index = 0; _index < locobyte_array.size(); ++_index)
    {
        if (debug) qDebug() << timeStamp() << "hexIndex: " <<
_index;
        if (debug) qDebug() << timeStamp() << "genChecksum: " <<
locobyte_array[_index].get_hex() << " : " <<
locobyte_array[_index].get_binary() << " and: " <<
_checksum.get_hex() << " : " << _checksum.get_binary();
        _checksum.set_fromHex(do_xor(_checksum,
locobyte_array[_index]));
    }
    _checksum.do_genComplement();
    locobyte_array.append(_checksum);
    return(_checksum.get_hex());
}
/* do_appendByte()
 *
 * Will append a new byte to the end of the packet, replacing the
checksum if it exists
 */
void LocoPacket::do_appendByte(QString _byte)
{
    LocoByte _newByte(_byte);
    if (validChk())
    {
        locobyte_array.removeLast();
        locobyte_array.append(_newByte);
    } else {
        locobyte_array.append(_newByte);
    }
}
void LocoPacket::do_appendByteArray(QByteArray _byteArray)
{
    for (int _index = 0; _index < _byteArray.count(); ++_index)
    {
        LocoByte _newByte;
        _newByte.set_fromByteArray(_byteArray/*.mid(_index, 1)*/);
        locobyte_array.append(_newByte);
    }
}
void LocoPacket::do_appendLocoByte(LocoByte _byte)
{
    locobyte_array.append(_byte);
}
/* get_validChk()
```

```
 *
 * Returns whether the packet has a valid OP code
 */
bool LocoPacket::validChk()
{
    LocoByte _hexHolder;
    for (int _index = 0; _index < locobyte_array.size(); ++_index)
    {
        if (debug) qDebug() << timeStamp() << "hexIndex: " <<
_index;
        if (debug) qDebug() << timeStamp() << "validPacket:
working: " << locobyte_array[_index].get_hex() << " : " <<
locobyte_array[_index].get_binary() << " and: " <<
_hexHolder.get_hex() << " : " << _hexHolder.get_binary();
        LocoByte tmp = _hexHolder;
        _hexHolder.set_fromHex(do_xor(tmp,
locobyte_array[_index]));
    }
    //qDebug() << timeStamp() << "checksum: " <<
_hexHolder.get_hex();
    if (_hexHolder.get_hex() == "FF")
    {
        return(true);
    }
    return(false);
}
bool LocoPacket::hasOP()
{
    return(locobyte_array[0].get_isOP());
}
bool LocoPacket::get_isEmpty() {
    if (locobyte_array.size() > 0) {
        return false;
    }
    return true;
}
// Check for valid OP code in sqlite
bool LocoPacket::validOP()
{
    if (get_isEmpty()) {
        return false;
    }
    if (!packetDB.isOpen())
    {
        bool _status = do_openDB();
        if (!_status)
        {
            return(false); // Can't determine validity
        }
    }
    QString _op = locobyte_array[0].get_hex();
    if (debug) qDebug() << timeStamp() << "Checking opcodes
database for " << _op;
    QSqlQuery _query;
    _query.prepare("SELECT * FROM opcodes WHERE opcode=:_op;");
    _query.bindValue(":_op", _op);
    if (!_query.exec())
    {
```

```
      if (debug) qDebug() << timeStamp() << _query.lastError();
      if (debug) qDebug() << timeStamp() << "Query to find valid
OP codes failed.";
        return(false);
    } else {
      if (debug) qDebug() << timeStamp() << _query.size() <<
_query.first();
    }
    if (_query.first())
    {
        return(true);
    }
    return(false);
}
QVector<QString> LocoPacket::get_DBopcodes ()
{
    QVector<QString> _opcodes;
    if (!packetDB.isOpen())
    {
        bool _status = do_openDB();
        if (!_status)
        {
            return(_opcodes); // Can't determine validity
        }
    }
    QSqlQuery _query;
    _query.prepare("SELECT * FROM opcodes;");
    if (!_query.exec())
    {
        qDebug() << timeStamp() << _query.lastError();
        qDebug() << timeStamp() << "Query to find all OP codes
failed.";
        return(_opcodes);
    }
    while (_query.next())
    {
        _opcodes.append(_query.value("opcode").toString());
    }
    return(_opcodes);
}
QVector<QString> LocoPacket::get_DBnames ()
{
    QVector<QString> _names;
    if (!packetDB.isOpen())
    {
        bool _status = do_openDB();
        if (!_status)
        {
            return(_names); // Can't determine validity
        }
    }
    QSqlQuery _query;
    _query.prepare("SELECT * FROM opcodes;");
    if (!_query.exec())
    {
        qDebug() << timeStamp() << _query.lastError();
        qDebug() << timeStamp() << "Query to find all OP names
failed.";
```

```
        return(_names);
    }
    while (_query.next())
    {
        _names.append(_query.value("name").toString());
    }
    return(_names);
}
bool LocoPacket::is_followOnMsg ()
{
    return (locobyte_array[0].get_hasFollowMsg());
}
QBitArray LocoPacket::get_QBitArray()
{
    int _bytes = locobyte_array.count();
    QBitArray _bitArray = QBitArray(_bytes*8);
    for (int _byteIndex = 0; _byteIndex < _bytes; ++_byteIndex)
    {
        for (int _bitIndex = 0; _bitIndex < 8; ++_bitIndex)
        {
            int _pos = (_byteIndex*8)+_bitIndex;
            _bitArray[_pos] =
locobyte_array[_byteIndex].get_oneBit(_bitIndex);
        }
    }
    return _bitArray;
}
QByteArray LocoPacket::get_QByteArray()
{
    QByteArray _byteArray;
    QBitArray _bitArray = get_QBitArray();
    short int _bytes = locobyte_array.count();
    for (short int _byteIndex = 0; _byteIndex < _bytes; +
+_byteIndex)
    {
        for (int _bitIndex = 0; _bitIndex < 8; ++_bitIndex)
        {
            int _pos = (_byteIndex*8)+(7-_bitIndex); // This is
disgusting :|
            _byteArray[_byteIndex] = (_byteArray[_byteIndex] |
(_bitArray[_pos]?1:0)<<(_bitIndex));
        }
    }
    return (_byteArray);
}
/**
 * Sometimes I wonder /
 * Whether my code should wander /
 * Those tretcherous glades /
 */


#include "locobyte.h"
/* Class: LocoHex
 *
 * Store and manipulate one byte of hex
 * For use with LocoPacket class
 * Convention: Most Significant Bit/Byte/Nyble is 0 index
```

```
 *
 * Christopher Bero [csb0019@uah.edu]
 * Team 4A
 */
bool LocoByte::debug = false;
/*
 * Default Contructor
 */
LocoByte::LocoByte()
{
    createEmpty();
}
/*
 * Loaded Contstructor
 */
LocoByte::LocoByte(QString _hex)
{
    createEmpty();
    set_fromHex(_hex);
}
LocoByte::LocoByte(QBitArray _bits)
{
    createEmpty();
    if (_bits.count() == 8)
    {
        byte = _bits;
    }
}
/*
 * Default Destructor
 */
LocoByte::~LocoByte()
{
    // Let QT take care of it
}
QString LocoByte::timeStamp()
{
    return(QTime::currentTime().toString("[HH:mm:ss:zzz] "));
}
bool LocoByte::operator ==(LocoByte _arg)
{
    return(byte == _arg.get_qBitArray());
}
void LocoByte::operator =(QBitArray _arg)
{
    if (_arg.count() == 8)
    {
        byte = _arg;
    }
}
/* createEmpty()
 *
 * Initialize all member variables.
 * Do not call except on object creation.
 */
void LocoByte::createEmpty()
{
    if (debug) qDebug() << timeStamp() << "Creating new empty
```

```
locobyte.";
    byte = QBitArray(8, 0);
    for (int16_t _bit = 0; _bit < 8; ++_bit)
    {
        byte[_bit] = 0;
    }
}
/* bitsFromHex()
 *
 * Take an initial hex character as a QString and generate the
array of bits it represents
 * Only works with one nyble (4 bits) at a time
 */
void LocoByte::bitsFromHex(QString _hex, int _nyble)
{
    if (debug) qDebug() << timeStamp() << "bitsFromHex()";
    _nyble = (_nyble*4);
    if (_hex == "1" || _hex == "3" || _hex == "5" || _hex == "7"
|| _hex == "9" || _hex == "B" || _hex == "D" || _hex == "F") {
        byte[_nyble+3] = 1;
    } else {
        byte[_nyble+3] = 0;
    }
    if (_hex == "2" || _hex == "3" || _hex == "6" || _hex == "7"
|| _hex == "A" || _hex == "B" || _hex == "E" || _hex == "F") {
        byte[_nyble+2] = 1;
    } else {
        byte[_nyble+2] = 0;
    }
    if (_hex == "4" || _hex == "5" || _hex == "6" || _hex == "7"
|| _hex == "C" || _hex == "D" || _hex == "E" || _hex == "F") {
        byte[_nyble+1] = 1;
    } else {
        byte[_nyble+1] = 0;
    }
    if (_hex == "8" || _hex == "9" || _hex == "A" || _hex == "B"
|| _hex == "C" || _hex == "D" || _hex == "E" || _hex == "F") {
        byte[_nyble+0] = 1;
    } else {
        byte[_nyble+0] = 0;
    }
    if (debug) qDebug() << timeStamp() << "end hexToBits()";
}
/* set_fromHex()
 *
 * Will re-generate an object to match the input string
 */
void LocoByte::set_fromHex (QString _hex)
{
    _hex = _hex.toUpper(); // Only deal with one case
    if (debug) qDebug() << timeStamp() << "set_fromHex() ->
nybles: 0-" << _hex.mid(0,1) << " 1-" << _hex.mid(1,1);
    bitsFromHex(_hex.mid(0,1), 0);
    bitsFromHex(_hex.mid(1,1), 1);
} /* end set_fromHex */
/* get_binary()
 *
 * Return the binary string
```

```
 */
QString LocoByte::get_binary()
{
    QString _binary = "";
    for (int16_t _bit = 0; _bit < 8; ++_bit)
    {
        if (byte[_bit] == 1)
        {
            _binary.append("1");
        } else {
            _binary.append("0");
        }
    }
    return(_binary);
} /* end get_binary() */
/* get_hex()
 *
 * Return the hex string
 */
QString LocoByte::get_hex()
{
    if (debug) qDebug() << timeStamp() << "hexFromBits()" << byte;
    int _decimal[2] = {0, 0};
    QChar _hexArray[2] = {'0', '0'};
    for (int _nyble = 0; _nyble < 2; ++_nyble)
    {
        for (int _bit = 0; _bit < 4; ++_bit)
        {
            int power = (3 - _bit);
            _decimal[_nyble] += pow(2, power) *
byte[(_nyble*4)+_bit];
        }
        if (debug) qDebug() << timeStamp() << "_decimal[" <<
_nyble << "]: " << _decimal[_nyble];
        if (_decimal[_nyble] <= 9 && _decimal[_nyble] >= 0) {
            _decimal[_nyble] += 48;
        } else {
            _decimal[_nyble] += 55;
        }
        _hexArray[_nyble] = static_cast<char>(_decimal[_nyble]);
    }
    QString _hex = QString(_hexArray, 2);
    if (debug) qDebug() << timeStamp() << "hexFromBits:" << _hex;
    if (debug) qDebug() << timeStamp() << "Generated hex from
bits: " << _hex << " integers: " << _decimal[0] << " " <<
_decimal[1];
    if (debug) qDebug() << timeStamp() << "end hexFromBits()";
    return (_hex);
} /* end get_hex() */
/* get_oneBit()
 *
 * Accepts values 0-7
 * Most Sig. Bit: 0
 */
bool LocoByte::get_oneBit(int _bit)
{
    if (_bit < 8 && _bit >= 0)
    {
```

```
        return(byte[_bit]); // Fire away
    } else {
        qDebug() << timeStamp() << "Error in get_oneBit().";
        return(-1);
    }
} /* end get_oneBit */
/* set_oneBit()
 *
 * Accepts a bit position and value, then regenerates the object
to match the new binary value
 */
void LocoByte::set_oneBit(int _bit, bool _value)
{
    if (debug) qDebug() << timeStamp() << "setBit() bit: " << _bit
<< " value: " << _value;
    if (_bit < 8 && _bit >= 0)
    {
        byte[_bit] = _value;
    } else {
        qDebug() << timeStamp() << "Error in set_oneBit().";
    }
    if (debug) qDebug() << timeStamp() << "end setBit()";
} /* end set_oneBit */
/* get_isOP()
 *
 * Returns a boolean value if the hex is an OPcode or not
 */
bool LocoByte::get_isOP()
{
    return(byte[0]);
} /* end get_isOP */
short unsigned int LocoByte::get_packetLength()
{
    if (!get_isOP()) { // Assume we want the second hex 7-bit
packet length
        return(get_decimal());
    }
    bool _bit1 = byte[1];
    bool _bit2 = byte[2];
    if (!_bit1 && !_bit2) {
        return(2);
    } else if (!_bit1 && _bit2) {
        return(4);
    } else if (_bit1 && !_bit2) {
        return(6);
    } else if (_bit1 && _bit2) {
        return(0); // Packet is N bytes, defined by next byte in
packet.
    }
    return(-1);
}
bool LocoByte::get_hasFollowMsg()
{
    if (get_isOP() && byte[4])
    {
        return (true);
    }
    return(false);
```

```
}
/* do_debugBits()
 *
 * Check the status of the bits in the raw nyble arrays
 */
void LocoByte::do_debugBits()
{
    qDebug() << timeStamp() << "Debugging bits.";
    for (int _nyble = 0; _nyble < 2; ++_nyble)
    {
        qDebug() << timeStamp() << "Nyble: " << _nyble;
        for (int _bit = 0; _bit < 4; ++_bit)
        {
            qDebug() << timeStamp() << " bit: " << _bit << "
value: " << byte[(_nyble*4)+_bit];
        }
    }
}
/* do_genComplement()
 *
 * Generate the hex's complement and then save it as the object
 */
void LocoByte::do_genComplement()
{
    if (debug) qDebug() << timeStamp() << "genComplement byte: "
<< get_hex();
    byte = ~byte;
    if (debug) qDebug() << timeStamp() << "end genComplement()";
}
/* set_fromBinary()
 *
 */
void LocoByte::set_fromBinary(QString _binary)
{
    for (int16_t _bit = 0; _bit < 8; ++_bit)
    {
        QString _test = _binary.mid(_bit, 1);
        if (_test == "1")
        {
            byte[_bit] = 1;
        } else {
            byte[_bit] = 0;
        }
    }
}
// http://qt-project.org/wiki/WorkingWithRawData
void LocoByte::set_fromByteArray(QByteArray _bytearr)
{
    byte = QBitArray(_bytearr.count()*8);
    for (int _byte = 0; _byte < _bytearr.count(); ++_byte)
    {
        for (int _bit = 0; _bit < 8; ++_bit)
        {
            short unsigned int _pos = (_byte*8)+_bit;
            bool _val = _bytearr.at(_byte)&(1<<(7-_bit));
            byte.setBit(_pos, _val);
        }
    }
```

```
}
QByteArray LocoByte::get_qByteArray()
{
    // Resulting byte array
    QByteArray _byteArray;
    // Convert from QBitArray to QByteArray
    for(int b=0; b<8/*byte.count()*/; ++b)
    {
        //_byteArray[b/8] = (_byteArray[b/8] | ((byte[b]?1:0)<<(b
%8)));
        _byteArray[0] = (_byteArray[0] | ((byte[b]?1:0)<<(b)));
    }
    return(_byteArray);
}
QBitArray LocoByte::get_qBitArray()
{
    return(byte);
}
int LocoByte::get_decimal()
{
    int _result = 0;
    for(int b=0; b<8/*byte.count()*/; ++b)
    {
        _result = (_result + ((byte[b]?1:0)<<(7-b)));
    }
    return(_result);
}
void LocoByte::setDebug(bool _debug)
{
    debug = _debug;
}


#include "locoblock.h"
bool LocoBlock::debug = false;
LocoBlock::LocoBlock ()
{
    //name = "No Name";
    //adr = QBitArray(11); //QByteArray(3,'0');
    bdl16x = 0;
    ds = 0;
    aux = false;
    occupied = 0;
}
LocoBlock::LocoBlock (int _bdl16x, int _ds, bool _aux, bool
_occupied)
{
    //adr = _adr;
    aux = _aux;
    bdl16x = _bdl16x;
    ds = _ds;
    occupied = _occupied;
}
LocoBlock::LocoBlock (QString _hex, bool _aux, bool _occupied)
{
    set_adr(_hex, _aux);
    occupied = _occupied;
}
```

```
LocoBlock::~LocoBlock ()
{
    //
}
QString LocoBlock::timeStamp()
{
    return(QTime::currentTime().toString("[HH:mm:ss:zzz] "));
}
bool LocoBlock::operator== (LocoBlock _arg)
{
    return((get_board()== _arg.get_board()) && (ds ==
_arg.get_ds()) && (aux == _arg.get_aux()));
}
/*
void LocoBlock::set_name (QString _name)
{
    name = _name;
}
*/
/*
void LocoBlock::set_adr (QByteArray _adr)
{
    adr = _adr;
}
*/
void LocoBlock::set_adr(QString _hex, bool _aux)
{
    aux = _aux;
    int _decimalAdr;
    if (aux)
    {
        _decimalAdr = (_hex.toInt(0, 16)+1)*2;
    } else {
        _decimalAdr = (_hex.toInt(0, 16)*2)+1;
    }
    ds = (_decimalAdr%16);
    if (ds == 0)
    {
        ds = 16;
    }
    bdl16x = ceil(_decimalAdr/16.0);
    if (debug) qDebug() << timeStamp() << aux << ds << bdl16x;
}
void LocoBlock::set_adr(int _bdl16x, int _ds, bool _aux)
{
    aux = _aux;
    ds = _ds;
    bdl16x = _bdl16x;
}
void LocoBlock::set_ds(int _ds)
{
    ds = _ds;
}
void LocoBlock::set_bdl16x(int _bdl16x)
{
    bdl16x = _bdl16x;
}
/*
```

```
void LocoBlock::set_adr (QBitArray _adr)
{
    adr = _adr;
}
*/
void LocoBlock::set_aux (bool _aux)
{
    aux = _aux;
}
void LocoBlock::set_occupied (bool _occupied)
{
    occupied = _occupied;
}
/*
QString LocoBlock::get_name ()
{
    return(name);
}
*/
/*
QBitArray LocoBlock::get_adr ()
{
    return(adr);
}
*/
int LocoBlock::get_bdl16x()
{
    return(bdl16x);
}
int LocoBlock::get_board()
{
    if (debug) qDebug() << timeStamp() << "Get board: " << bdl16x
%10;
    return(bdl16x%10);
}
int LocoBlock::get_ds()
{
    return(ds);
}
bool LocoBlock::get_aux ()
{
    return(aux);
}
bool LocoBlock::get_occupied ()
{
    return(occupied);
}
int LocoBlock::get_occupied_int ()
{
    return(occupied?1:0);
}
```