[DINST]

~~Dumb~~ Digitrax Instruction Set

Christopher Bero
Team 4A
CPE453 - Spring 2015

This document outlines train_rec, a program capable of handling communication to a digitrax HO model track via a network.

Index

aka Again, and this time in English!

To aid in getting up to speed with this Dumb Instruction Set, a collection of redundant and otherwise useless terms are specified on this page. These pertain to both train_rec's code base and Digitrax's own documentation.

Loconet

Loconet is the HO model track and supporting digitrax hardware. I prefix most of the classes in train_rec with loco- but have eschewed using loconet as a class name to keep from conflicting with this usage of the word.

Serial, USB, Loconet Buffer, PR3

These terms correspond to either the physical PR3 device which allows a computer to link to the track or to the serial connection to the PR3 via train_rec's locoSerial class.

Slot

Can be either a Qt slot, or a virtual slot used by Loconet to track trains. Virtual loconet slots are a construct which manage trains and special functions. Trains do not have to have unique addresses, but the pairing of a slot to a train is unique, and once paired trains are controlled by using the number of the slot they are in.

opcode

Digitrax fancies themselves real computer scientists and call the first byte in each packet an opcode. This byte determines the purpose of the packet. A database of opcodes is available via the locoPacket class.

Checksum

Every loconet packet is terminated with a checksum byte which can be used as parity to verify the packet was correctly received. This process is taken care of in the locoPacket class.

train_rec

aka Train What?

Train Receiver (interchangeably: train_rec) is a multi-threaded message handler written in C++ with the Qt framework. At its core is a set of classes which abstract between C++ level variables and the raw hex which comprises Digitrax's (rather disgusting) serial language. These classes are fairly portable, and can be salvaged for future projects which wish to take the track communication in a different direction. At a higher level, train_rec has four threads (existing as classes) which operate the GUI (mainwindow), USB connection (locoserial), SQL connection (locosql), and UDP socket (locoudp). This division greatly benefits the program's performance but does also add some slight programming overhead. Tangential classes serve a very small purpose and are used as a stop-gap to keep the system's architecture clean; they may be removed by future updates to the software. Because of their low-impact on the overall project, these last classes are not documented here. To aid in future development, low and high level classes come equipped with a small suite of debugging statements which can be individually toggled on and off for each class.

Low level classes:
- locobyte.cpp
- locopacket.cpp

High level classes:
- locoserial.cpp
- locosql.cpp
- locoudp.cpp

Tangential classes:
- locoblock.cpp
- locotrain.cpp

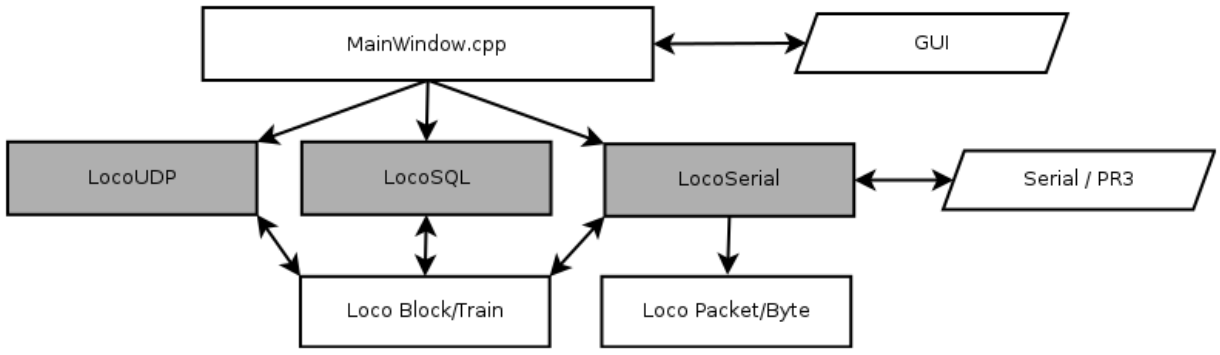

Illustration 1: High Level Design

Class - LocoByte

aka Descent into sanity

LocoByte does a lot of heavy lifting for you, it acts as a container for individual bytes inside of a loconet packet. As such, the class can self-determine if its instance is a packet's opcode. Most of the structuring for the class is built around portability and input/output; You can enter the bytes as hex, decimal, binary, or directly as qBitArrays/qByteArrays.

Methods

== - You can compare locobytes with the custom == operator.
= - You can assign a qBitArray to the locobyte with =.
bitsFromHex() - A Qstring with hex, such as "BA" can be loaded into the locobyte.
set_fromHex() - Public wrapper around bitsFromHex().
get_binary() - Returns a Qstring of binary.
get_hex() - Returns a Qstring of the hex.
get_oneBit() - Used to poll individual bits of the byte.
set_oneBit() - used to modify individual bits of the byte.
get_isOP() - Uses Digitrax's specification to determine if a byte is an opcode.
get_packetLength() - Uses Digitrax's specification to determine the supposed length of the rest of the packet in bytes. Only to be used on opcode or byte directly after opcode.
get_hasFollowMsg() - Uses Digitrax's specification to determine if the packet should receive a reply from the track.
do_debugBits() - Don't use this, it just spills the class contents to std_out.
do_genComplement() - Used in generating the checksum, will set the internal byte to the complement of whatever it used to be. **Doesn't return a value.**
set_fromBinary() - Sets the class contents from a Qstring of the byte's binary representation.
set_fromByteArray() - Method adapted from solution found online (referenced in the class's code). Used to set the byte's qBitArray from a qByteArray.
get_qByteArray() - Returns the calculated qByteArray.
get_qBitArray() - Returns the core qBitArray, faster than get_qByteArray().
get_decimal() - Returns an integer of the byte's decimal value.
SetDebug() - Unimplmented.

Class - LocoPacket

aka Descent into [BD0244F7]

LocoPacket is… another c++ class? Yeah, I mean its pretty self evident. Use this class to build loconet packets from locobyte instances. LocoPacket is integrated with a SQLite database that pairs opcodes with descriptions, this list determines which opcodes are legal to be used with the track.

Methods

== - You can check equality with this overloaded operator.
clear() - Purge the packet contents.
do_openDB() - Attempts to open the SQLite database of opcodes.
do_closeDB() - Closes the connection to the SQLite database.
set_allFromHex() - Sets the packet contents from a Qstring.
get_packet() - Returns the Qstring of the packet's hex.
get_finalSize() - Returns the Digitrax specification of how long the packet should be based on the opcode. Does not reflect the current number of bytes in the packet.
get_size() - Returns the current number of bytes in the packet.
get_Opcode() - Returns Qstring of the opcode byte.
get_locobyte() - Transparently escalates a locobyte out of the packet.
do_xor() - Used in generating the packet's checksum, returns a qstring of two locobytes xor.
do_genChecksum() - Generates and appends a checksum for the packet.
do_appendByte() - Adds a byte from a qstring to the end of the packet.
do_appendByteArray() - Adds a qByteArray to the end of the packet.
do_appendLocoByte() - Adds a locobyte to the end of the packet.
validChk() - Returns a boolean status for the packet's checksum.
hasOP() - Returns a boolean status for the packet's opcode.
get_isEmpty() - Transitional method to determine if a packet is empty.
validOP() - Checks the packet's opcode against the SQLite database.
get_DBopcodes() - Returns all of the opcodes in the SQLite database.
get_DBnames() - Returns all of the corresponding opcode names in the SQLite database.
is_followOnMsg() - Boolean value of whether the packet should get a response from the track.
get_QBitArray() - Returns the packet's QbitArray.
get_QbyteArray() - Returns the packet's QbyteArray.

REQUESTS

aka OMG WHAT DO?

Requests are methods of interacting with Loconet via SQL tables. Each type of request is designed to make your life as a team in CPE453 easier. To that effect, if you wish for additional features, don't hesitate to ask (of course, whether the feature is feasible will have to be determined).

Keep in mind that slots are a software-only construct. Each train has a (non-unique) ID, which can only be controlled via a "slot number" that is associated with the train. The SQL in train_rec wields slot numbers with reckless abandon and occasionally calls them trains to keep the notion easier (but can be confusing!).

Train/Slot Requests:

Set the direction and speed of a train/slot. Slot number is the ID. Speed is a scalar percent. Direction is a boolean value, with true corresponding with reverse movement.

Switch Requests:

Set the position of a switch. The number physically labeled on the switch is used as the ID.

Macro Requests:

Most everything else is a macro. Macros are utilities to make interacting with Loconet easier on a higher level. Check the demo program, train_rec_sql, for macro implementations.

MACROS

aka Trains for Normal People

Macros listed here are to provide an overview, as sql implementation may vary. Sample implementation software is available either on the EB Linux Lab computers, or from https://github.com/ctag/cpe453/tree/master/train_rec_sql.

Example Listing:

```
INSTRUCTION, arg1, arg2
Packet Format in [byte]s.

    Description of macro, how to wield it, why it matters. If no
arguments are given, then leave their fields as 'null' in sql.
```

Here INSTRUCTION would be a string entered in the “macro” field of the table, with arg1 and arg2 being placed in their respective columns as well.

```
SLOT_SCAN, slot#
[BB] [SLOT] [00] [CHK]

    Queries track for the status of the given slot#. Valid slots are
1-119. Returns with the state of a 'train' in sql, which is really
just a slot. Don't query the status of slots that you aren't
interested in; for example, what if MASTER returns several slots
with the same locomotive address? Which one do you update to control
the real train?! If you only query slots that you know have trains,
this won't happen.
```

```
SLOT_DISPATCH, slot#
[BA] [SLOT] [00] [CHK]

    Supposedly puts the given slot back into “COMMON” mode, but has
not worked consistently for me. Use when slot has mode “IN_USE”.
If the associated train does not update, then the command failed
with a LACK packet. To be safe, follow with a SLOT_SCAN.
```

```
SLOT_CLEAR, slot#
[B5] [SLOT] [03] [CHK]

    Puts the given slot into “FREE” mode. Use when slot has mode
“COMMON” or “IN_USE”. Associated train state does not automatically
update, must run SLOT_SCAN after this command to check the train
state.
```

```
SLOT_REQ, train#
[BF] [00] [ADR] [CHK]

    Requests a slot for a given train address. Returns with the
state of a 'train' in sql, which is really just a slot. This is the
initial command to go from train address to slot number that's used
in the other macros. After running this macro, check the slot/train
table for a new row corresponding to the train address.
```

TRACK_ON
[83] [CHK]

Turns power to the track on. Usually results in all detection sections sending a bootup packet which lists them as “occupied”. train_rec can be configured to ignore extraneous detection sections and not let them spam the status table when the track powers on.

TRACK_OFF
[82] [CHK]

Turns power to the track off.

TRACK_RESET
[82] [CHK] then [83] [CHK]

Turns power off, waits a few (ten) seconds, turns power back on.

TRAIN_REQ, train#, speed%
[SEVERAL PACKETS]

Though no team has requested this feature, it has been added at the behest of the product owner. Has not been tested; don't use it. Train# is the address of a train or trains; speed% is an integer -100 to 100. The system will spam traffic to get a slot for the train, set it to active, and give it a throttle command; because of this, you may have to call the macro twice if the train is not already in an IN_USE slot.

Suggested workflow: Take a train address, call SLOT_REQ to get a slot, then spam this macro twice. After that, you should only need to call this once to update speed, though it will still inundate the track with traffic and make life miserable for everyone else.

Detection Section Addressing

aka Wizardry

TL;DR:
Likely the most elusive part of this system; luckily what we've gleaned will keep you from having to work with this conversion. You can totally just skip this page.

Introduction:
There are three numbers associated with a detection section. *Raw hex* is created by parsing address bits. *LS numbers* are an abstract that appears to be JMRI specific? Lastly are *BDL-DS numbers*, where each BDL168 has a number, and each DS on the BDL has a spot 1-16. We get raw hex from Loconet, and want to reach BDL-DS numbers to use in SQL.

An example:
Consider the packet [B2 6C 58 79].

Using the Digitrax documentation, we find the associated hex address for the detection section to be [46C] with aux bit of [0]. Awesome, after this point the documentation is utterly useless!

Next, we convert the hex directly into a decimal number: 1132.

Now the aux bit comes into play; if the aux bit is set we add 1 to the decimal number, then multiply by 2. Otherwise (if aux is 0) we multiply by 2, then add 1. Thus we get (1132*2)+1 = 2265.

This resulting number, 2265, is the LS number; written LS2265. Now we want to get to the BDL-DS number. To get the board, do LS# modulo 16. To get the DS, do LS# divided by 16 and round up. In our example 2265%16 = 11 and ceil(2265/16) = 142. The last digit of the BDL number corresponds to the board below the track. Thus we achieve 2-11 as the detection section of interest! This algorithm has been tested a does work for all detection sections on the inner-urban track.

Implementation

aka The Finished Puzzle

This is the easy part! (At least I really, sincerely hope).

You may download train_rec and supporting software from github.com/ctag/cpe453.

OR

As any EB Linux Lab user, run the following commands to use a pre-compiled executable of train_rec or train_rec_sql.

```
/home/student/csb0019/train_rec
```

train_rec is the main program here, it does all the magic for the track, and must be connected to one of the USB-buffers. You can run it from one of the Linux computers at the front of the train lab.

```
/home/student/csb0019/train_rec_sql
```

train_rec_sql is a sample program which allows you to manually control trains, switches, and view blocks(detection sections). Use it to get the SQL queries which work with train_rec.

As an easter-egg (I guess?) train_rec also supports direct UDP packet commands, should you so wish to implement it. Default port is 7755, send a datagram with just the QString hex for a packet minus the checksum. Packet will be checksummed and fired off to serial without much parsing.

TODO

aka Pull requests welcome.

- Debugging could use an overhaul. I feel the structure is fine, but the statements could be much clearer, and there should by run-time buttons for enabling debugging statements.
Estimated difficulty: super easy.

- Let's face it, the GUI sucks. A full overhaul may not be necessary, but at least connect the signals/slots from each thread to relevant output on the screen.
Estimated difficulty: how well do you really know Qt? Good luck.

- Tangential classes should be removed and replaced with a smarter signal/slot design between the threads.
Estimated difficulty: a good day's work.

- SQL structure is currently somewhat hard-coded to ECE's pavelow server, this should be fixed and the table structure should be better refined and defined.
Estimated difficulty: 40 hours.

- LocoSQL currently handles the conversion between speed and packet hex, this should be pushed back to LocoSerial.
Estimated difficulty: You're going to learn a lot about this code structure… 40 hours.

- UDP is really non-existant. A bolstered UDP class capable of handling macros could make the program much more robust for future projects.
Estimated difficulty: 40 hours.