

railway.exe

Developer Guide

Contents:

Project history

Files included with the project

 Project source files (on first publication)

 Release files

Program structure

Program operation

Diagnostic functions

Classes

The main user-interface items

Embarcadero version

Users' wish list

Project history:

The program began life back in 2001 as a simple track building program using Borland's Delphi. I had a vague idea of building Harry Beck type track diagrams, but before I got anywhere near that I decided it was time to learn C++ so I ported it over to Borland's C++Builder 3 and continued in C++. After a while (progress was very intermittent, I didn't really have a definite end in mind at that stage) I wondered whether it might be possible to set routes and run trains, so I started in that direction. I ran into a great many cul-de-sacs and had to back out again, leaving it for long periods as other projects took my attention. Along the way I bought C++Builder 4 on Ebay and continued with that up to 2017 when Embarcadero released their (very) updated C++ Builder free of charge - If there is the slightest possibility that you might wish to develop the program further or even just experiment with it sometime in the future then I urge you to download the free compiler as soon as possible because it might well not be available indefinitely. See 'Embarcadero version' later.

It was about 2005 or 2006 when I decided that the program was going somewhere so gave it a lot more attention from then on. I kept adding features more or less randomly as I thought of them, so the result is by no means a reflection of a systematic design process, though I hope it's all fairly logical, if complex.

I still count myself very much a C++ beginner, it's such a vast language with so many features that I still don't understand. I tend to learn enough to make something work, then leave that bit of code alone even after I've discovered much better ways to do the same thing. So I hope all who come after me won't be too critical of my efforts - you will see many things that will have you scratching your head and saying 'why the ** did he do it like that?' - I often find myself saying that too!

I have included quite extensive comments throughout, the .h files in particular contain descriptions of all classes, types, variables and functions, and these are believed to be accurate. The .cpp files contain comments relating to specific functions and activities within functions where I have felt that there is a need for explanation, though please take these with a health warning - they were accurate when written but I haven't always kept them up to date as later developments have taken place.

My preference is to lump all classes that are closely related together in a single unit. It seems conceptually simpler to me that way though I know it is not the preference for many C++ programmers who like to have each class in its own unit. Others may wish to separate the classes at a later development stage.

My hope is to hand over further development to others if there is sufficient interest. I've lived with this project so long that it has become something of an obsession, and I feel the need to break free and see what else is out there in the world. *Someone once told me that there are things called flowers and trees, holidays and other countries. I tried to find the source code for them but haven't succeeded yet - I wonder what operating system they run on?* I shall of course remain available to help where I can in explaining how functions work where it's not clear (if I can remember or can work it out from scratch - please don't rely too heavily on this though - especially as time passes and my memory deteriorates even further), and I shall continue to try to correct program errors when they appear - at least when they are caused by my own programming efforts.

One plea I would make is for at least some form of inbuilt diagnostic and error detection capability to be retained. The functions in now - call stack recording, and event and error logging - are my own creations and although I'm sure there are better and less cumbersome ways of doing the same thing they

really have repaid their (considerable) development and maintenance effort many times over. Without them I don't think I could have solved many (probably most) of the errors that have come to light. So far - and I'll probably regret saying this - there hasn't been a fault that I've not been able to track down and rectify, and it's all down to the inbuilt diagnostics and in particular the error file that records the state of play when the error occurred.

During the later stages of development I called the program `railway.exe`, intending to think up a snappier title before publication. However all the obvious ones already seem to have been used so I just stuck with `railway.exe`. It's not very imaginative I know but at least it gives a strong hint as to what it's about! Later developers may wish to change it. Google code (now ported over to GitHub) won't allow '.' characters in URLs so the development project is called 'railway-dot-exe'.

Note that in this document names of program items are indicated in blue.

The version of the program is stored in a variable called `ProgramVersion` defined as an `AnsiString` in the `TInterface` class and is displayed in the program's 'About' box. It is also recorded in several of the files that the program generates, notably the error file, so please remember to update it whenever a later version is produced.

The program has been available for public testing since March 2010, and I am grateful for all the feedback that I have received. The user website is at <http://www.railwayoperationsimulator.com/>

Files included with the project:

The source files are accessed for development purposes using the 'Git' version control system. Developers will need to be familiar with how this system works and to have it installed on their computer in order to use the system and especially to write to the source repository. For those that are unfamiliar with it there is plenty of information on the web and it's all open-source and therefore free. I use TortoiseGit which has a 'Windows' front end and is relatively easy to use.

The project source code URL is <https://github.com/AlbertBall/railway-dot-exe/>

The standard folder structure has been used as follows:-

<code>railway-dot-exe/master</code>	main line of development
<code>railway-dot-exe/branches</code>	separate (possibly experimental) lines of development
<code>railway-dot-exe/tags</code>	source files associated with specific program releases

Project source files (Embarcadero version):-

In folder `/railway-dot-exe/master/`

<code>Help_Files</code>	folder
<code>CallerChecker</code>	folder (described later)
<code>AboutUnit.cpp & .h</code>	'About' box as a separate form
<code>DisplayUnit.cpp & .h</code>	screen output functions
<code>GraphicUnit.cpp & .h</code>	graphic functions
<code>InterfaceUnit.cpp & .h</code>	user inputs & application functions
<code>railway.cpp (no .h)</code>	contains the <code>tWinMain</code> function that calls the application

TextUnit.cpp & .h	text functions
TrackUnit.cpp & .h	all track, preferred direction and route functions
TrainUnit.cpp & .h	trains, train operation and timetable functions
Utilities.cpp & .h	error handling and miscellaneous functions
Interface.dfm	the railway.exe interface form
About.dfm	'About' box form
railway.cbproj) project files used by the compiler
railway.cbproj.local)
railway.res	Embarcadero generated resource file
railway.ico) contain the program icon (railway.ico probably isn't needed)
railway_Icon.ico)
railwayPCH1.h	Embarcadero generated header file
Borland.res	original resource file containing all the images
DeveloperGuide.doc	this file, in Microsoft Word 2003 format
DevHistory.txt	text file used to record development history

In folder /railway-dot-exe/master/Help_files/

HTML	folder containing all the .htm files
images	folder containing all the images
Help.chm	main help file, accessible from within the program
Help.hhc	help contents file
Help.hhk	index file
Help.chw	compiled index file
Help.hhp	help project file (for use in Microsoft's help compiler hhw.exe)
Manual.doc	'Word' version of the user manual
Manual.pdf	PDF version of the user manual (distributed with each release)
QuickStart.doc	'Word' version of the quick start guide
QuickStart.pdf	PDF version of the quick start guide (distributed with each release)

In folder /railway-dot-exe/trunk/CallerChecker/


CallerCheckerUnit.cpp & h	main unit files of this utility program (described later)
CallerCheckerUnit.dfm	form for this program
CallerChecker.cpp	contains the tWinMain function that calls the application
CallerChecker.cbproj) project files used by the compiler
CallerChecker.cbproj.local)
CallerChecker.exe	executable (this must be copied to the same folder as the railway.exe source files in order to work)
CallerChecker.res	Embarcadero generated resource file
CallerChecker.stat	compilation information

Release files:-

Files are released in 'zip' format using the folder structure as follows:

release vx.xx ('x.xx' is the release version) top level folder

Some elements have no links (platforms, concourses, parapets etc) but others have a maximum of four links (crossovers, bridges and points¹ - although points only have 3 they are treated as having 4, see below). The simpler elements have just 2 links. For example a diagonal crossover would have links 1, 3, 7 and 9. Number 5 is omitted for better symmetry - all opposites add to 10, all diagonals are odd, all horizontals and verticals are even. Points use the same link number for the leading track link, e.g. points with straight track 4 to 6 and diverging track 4 to 9 would have links 4, 6, 4, 9. The track element class contains a 4 integer array - [Link\[4\]](#), which sets out the relevant link values. Unused links take the value -1 to indicate that they are not set. The order of the links is vitally important as the position of a link is often used to find specific information. For example all continuations, buffers and gaps use [Link\[0\]](#) for the continuation/buffer/gap end, all points use [Link\[0\]](#) & [Link\[2\]](#) for the leading end, all bridges use [Link\[0\]](#) & [Link\[1\]](#) for the top track etc (see [TrackUnit.cpp](#) - function [TTrack::TFixedTrackArray::TFixedTrackArray\(\)](#) for more information).

The objects of type [TFixedTrackPiece](#), as their name implies, are fixed objects, relating to the different types of track element available on the railway. They are assembled into an array ([FixedTrackArray](#)) in class [TTrack](#). The track element that is variable and positioned on the screen in building a railway is of class [TTrackElement](#), descended from [TFixedTrackPiece](#), and this also includes two other 4-integer arrays [Conn\[4\]](#) & [ConnLink\[4\]](#), used to define the elements that are adjacent to each link position and the position of the connecting link in those elements respectively. These values are set when the 'Link all track together' button  (Program name [TrackOKButton](#)) is clicked and the track links successfully.

Track elements are stored in a vector, and each is identified to the program by its position within the vector. These positions are stored in the [Conn\[4\]](#) array so that each element knows which other elements it is linked to. Another integer member of [TTrackElement](#) is [Attribute](#). This is used to store information for elements whose form can vary - i.e. points and signals. For points [Attribute](#) can be 0 or 1 for straight or diverging respectively (or left hand fork or right hand fork respectively for 'Y' shaped points), and for signals can be 0, 1, 2 or 3 for red, yellow, double yellow or green respectively for four-aspect signals. For three-aspect attributes 2 and 3 are green, and for two-aspect signals attributes 1, 2 and 3 are green (or 'proceed' for ground signals). By means of the above information the program can navigate along the track to find preferred directions and routes.

Preferred direction elements (class [TPrefDirElement](#)) are descended from [TTrackElement](#) and contain additional information relating to the entry and exit links and their array positions, and the position of the underlying track element in the track element vector. These elements are also used for routes.

The program makes extensive use of C++ containers. The track itself, preferred directions, routes, locked routes, text, trains, timetable entries etc all use vectors, which allow railways to grow without practical limits. Also used extensively are maps and multimaps - linked lists of values that enable rapid access to individual elements using a 'key'. These are used to speed up access to things like track elements, using the location (horizontal and vertical positions) as the key. Without maps it would be necessary to search vectors one element at a time until the one with the desired location was found.

¹ Points can also be referred to as 'switches' or 'turnouts'.

With maps the location itself provides a direct means of finding the required element. Use is also made of lists and deque, though less extensively.

Trains are managed mainly by two classes, `TTrain` is the train class, with one object per train, and `TTrainController` is the handler for all the trains and timetables. All user inputs, program operating modes and the system clock are managed by the `TInterface` class, display outputs by the `TDisplay` class, graphics by the `TRailGraphics` class, and text by two classes, `TTextItem`, with one object per piece of text, and `TTextHandler` for managing all the text.

More detail is provided in comments within the source files. The header (.h) files in particular contain descriptions of classes, types, functions and variables, and the .cpp files contain explanatory comments for functions where it was felt that they were needed. Please take the .cpp comments with a health warning, as they weren't always kept up to date when changes were made to the functions. Hopefully they will help in providing a general guide to intent. The .h file comments are believed to be accurate.

Program operation:

At any time the program can be in one of many different modes of operation. The modes are set by the user interfacing with buttons and menus etc, and these modes determine what interface elements are visible and enabled, and direct events on mouse clicks and during cycling of the internal control loop - controlled by the ticking at 50ms intervals of an internal clock - `MasterClock`. The control loop handles functions that need to be accessed on a time basis such as train movements, rather than by a response to user input, and handles polling functions such as enabling or disabling the navigation buttons depending on the section of railway that is currently visible.

The major modes that affect visibility and enabling of interface elements are lumped together and handled by 'switch' and 'case' statements within `TInterface`. These are called `Level1Modes` and `Level2Modes`, the `Level2Modes` being submodes of `Level1Modes`, and enumerated in the `TInterface` class as follows:-

```
enum TLevel1Mode { BaseMode, TrackMode, PrefDirMode, OperMode, RestartSessionOperMode,
TimetableMode } Level1Mode;
```

```
enum TLevel2TrackMode { NoTrackMode, AddTrack, GapSetting, AddText, MoveText,
AddLocationName, DistanceStart, DistanceContinuing, TrackSelecting, CutMoving, CopyMoving,
Pasting, Deleting } Level2TrackMode;
```

```
enum TLevel2PrefDirMode { NoPrefDirMode, PrefDirContinuing, PrefDirSelecting }
Level2PrefDirMode;
```

```
enum TLevel2OperMode { NoOperMode, Operating, PreStart, Paused } Level2OperMode;
```

Two other modes that aren't included in the above as they don't affect visibility or enabling of other user interfacing elements are the route modes. These are enumerated as:-

```
enum { None, RouteNotStarted, RouteContinuing } RouteMode; These are used during operation of the
railway and define whether a route is awaited or in course of building.
```


On first loading the program **BaseMode** is entered, and the **File**, **Mode** and **Help** menus are available. Whenever a level 1 mode other than **BaseMode** is selected (from the **Mode** menu) the **File** and **Mode** menus are disabled, and are not available again until **BaseMode** is re-entered, by exiting the current mode. The level 2 modes are available from three of the level 1 modes - **TrackMode**, **PrefDirMode** and **OperMode**. These are described below.

Level 1 Modes	Level 2 Modes
---------------	---------------

BaseMode - *no mode selected (a railway may or may not be loaded)*

TrackMode - *Build/modify*

Level2TrackMode

NoTrackMode - *default for TrackMode not selected*

AddTrack - *add/remove track & other elements*

GapSetting - *set gaps*

AddText - *add/remove text*

MoveText - *move existing text*

AddLocationName - *name locations*

DistanceStart - *select start element for distance/speed setting*

DistanceContinuing - *select next element for distance/speed setting*

TrackSelecting - *make a rectangular selection (via Edit menu)*

CutMoving - *cut and move a selection*

CopyMoving - *copy and move a selection*

Pasting - *paste a selection*

Deleting - *delete a selection*

PrefDirMode - *Set preferred directions (select start element for distance/speed setting)*

Level2PrefDirMode

NoPrefDirMode - *default for PrefDirMode not selected*

PrefDirContinuing - *select next element for PrefDir setting*

PrefDirSelecting - *make a rectangular selection (via Edit menu)*

OperMode - *Operate*

Level2OperMode

NoOperMode - *default for OperMode not selected*

Operating - *railway & timetable clock running*

PreStart - *paused prior to first start*

Paused - *paused other than prior to first start*

RestartSessionOperMode - *Restart after a session load (no submodes)*

TimetableMode - *Create or edit a timetable (no submodes)*

Diagnostic functions:

To help in diagnosing faults it has been arranged that the current state of the railway is saved in detail to an error file (errorlog.err) whenever an error is caught. This file also saves two other important items - a call stack and an event log - both stored as dequeues. Almost all functions begin by pushing onto the **CallLog** stack information about the call, including a unique caller number, which identifies the calling function. When the function returns the information that was pushed is popped, so that at any time **CallLog** contains the list of nested functions at the time of the error. Although the system took considerable time to set up it has proved invaluable in allowing faults to be diagnosed. In effect it is an internal debugger that operates continually and provides a detailed output whenever a fault occurs.

The event log works similarly, but is a list rather than a stack, and stores the last thousand significant events such as button clicks and mouse moves etc. This provides a history of the actions that preceded the error.

It is possible to extract railways, sessions and timetables from the error log - instructions for doing this are included as comments in function `TInterface::SaveErrorFile()`.

I strongly urge developers to either keep these functions, or replace them with something at least as effective, because I can't overstate their value.

In order to ensure that all the caller numbers are unique (so that functions that call other functions can be identified from the call stack) another program - `CallerChecker.exe` - has been written to examine all the functions that have caller numbers, and create a file containing duplicates. In order to work this executable must be located in the folder where the source files are located. The program and its source code (also written in C++Builder 4) is provided. Note that in order to work effectively the program needs up to date arrays of all the source file names and all the functions that use callers. Please keep these up to date when new functions or files are added. There is a box to include 'missing' caller numbers if required, but I stopped using that a long time ago as there are now very many missing numbers. All that matters is that numbers are unique, so that calls can be traced to the calling functions in the error log, missing numbers are of no concern. I normally select the 'print output only' option, and examine the file 'CallerOutput.txt' afterwards.

Classes:

In `AboutUnit`:

TAboutForm *contains the 'About' box as a small form*

In `DisplayUnit`:

TDisplay *contains all functions and data relating to screen output*

In `GraphicUnit`:

TRailGraphics *contains all graphic data and functions*

In `InterfaceUnit`:

TInterface *contains all user input and associated functions, internal clock functions, program operating mode control and general railway data*

In railway.cpp:
none

In TextUnit:

TTextItem *contains an individual piece of text and its characteristics, and low-level text functions*

TTextHandler *contains the text vectors and higher-level text functions*

In TrackUnit:

TMapComp *contains the map comparator based on horizontal and vertical element position*

TFixedTrackPiece *contains basic track element data and functions for each type of element*

TTrackElement *derived from TFixedTrackPiece and contains data and functions for each track element that is placed on the railway*

TTrack *contains the track vectors, maps and multimaps (which contain and point to the individual track elements), distances and line speeds, location names and associated functions.*

TFixedTrackArray *a class that holds an array of TTrackPieces, only accessible to TTrack*

TSigElement *(structure) used as basic elements in tables of signals*

TPrefDirElement *derived from TTrackElement and contains data and functions for each preferred direction or route element*

TGraphicElement *a special class to handle graphic elements that change state during operation such as point and route flashing elements*

TOnePrefDir *contains collections of TPrefDirElements as vectors and their functions*

TOneRoute *derived from TOnePrefDir and contains similar data and functions as TOnePrefDir but for routes*

TRouteFlashElement *a single flashing element of a route that flashes during setting*

TRouteFlash *the flashing route*

TAllRoutes *contains the collection of all current and locked routes on the railway as vectors, and associated functions*

TLockedRouteClass *handles routes that are locked because of approaching trains*

IDInt *a special class that holds route identifier information*

In TrainUnit:

TTrain *contains all data and functions for a single train*

TTrainController *contains data and functions for trains and timetables*

TContinuationAutoSigEntry *class for turning signals back to green in stages after a train has exited an autosig route at a continuation*

TContinuationTrainExpectationEntry

class that stores data for trains expected at continuation entries (kept in a multimap), used to display information in the floating window when mouse hovers over a continuation

TActionVectorEntry contains a single train timetable action

TTrainOperatingData contains timetabled train status information

TTrainDataEntry contains all timetable data for a single train. A vector, defined as TTrainDataVector contains the whole timetable as a collection of TTrainDataEntry elements

TOneTrainFormattedEntry contains formatted timetable data for a specific train's timetable action

TOneCompleteFormattedTrain contains formatted timetable data for all timetable actions for a specific train excluding repeats

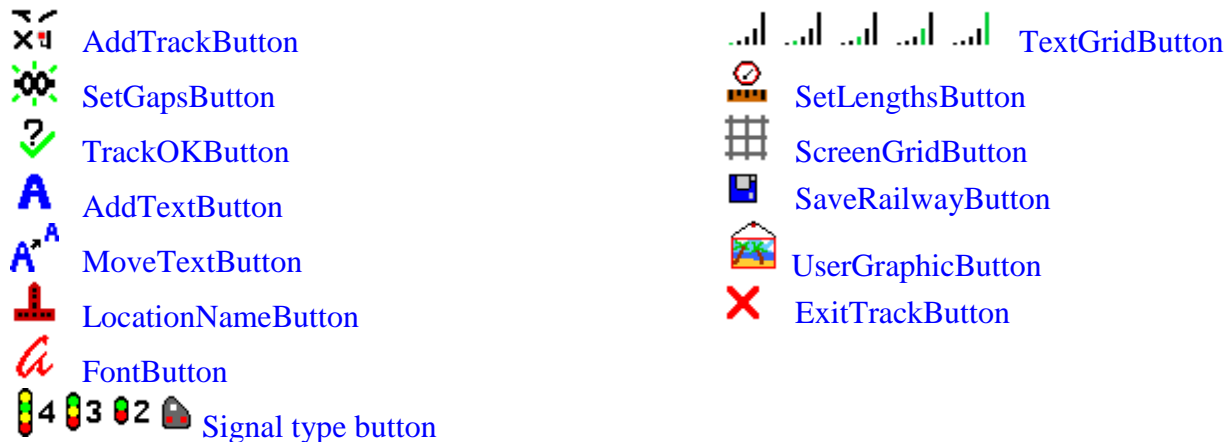
TTrainFormattedInformation contains all information for a single timetable entry (including repeats). A vector - TAllFormattedTrains - is defined that contains the whole timetable in formatted form as a collection of TTrainFormattedInformation entries

In Utilities:

TUtilities contains error and diagnostic functions, and general purpose data and functions used by other units

The main user-interface items:

'Build/modify railway' buttons (Menu item = BuildTrack1)



'AddTrack' SpeedButtons (names: 'SpeedButtonX' where 'X' = the number given)





 ONE

 ALL

SHOW HIDE

X



13

Embarcadero version:

The latest C++ Builder compiler (Version 10.3) is now available as a free starter edition, known as the Community Edition. It is available at this address

<https://www.embarcadero.com/products/cbuilder/starter>

Railway program development up to version 1.3.2 used the Borland C++ Builder 4 compiler, released in 1999 and incredibly still working many years later. Borland was taken over by Embarcadero Technologies in 2008, and they continued developing the product, the latest version being 10.3 released in November 2018.

Users' wish list:

This is a list of features recognised as being desirable, many of which have been suggested by users as useful enhancements. They aren't in any particular order and there are bound to be very many more that aren't listed.

- Three-way points
- Automatic route setting
- Option to use & display imperial units as well as metric units
- Ability to record and replay sessions
- Signalbox mode where individual signals and points are operated directly, perhaps via a graphical lever frame, with user-defined interlocking
- Sound effects, e.g. enter a track ID & hear trains & station announcements at that location
- Random failures of signals & points
- Variable train lengths
- Restricted routes - DC 3 & 4 rail; AC; tube stock etc
- Multi-player facility over the internet

When you've programmed in and thoroughly tested all those little extras you can take the rest of the day off!
