The Trainset Railroad Simulation

Richard A. Brown (Editor)*
Fred B. Schneider (Editor)
Jacob Aizikowitz**
Thomas C. Bressoud
Tony Lekas***

TR 93-1329
February 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Department of Mathematics, St. Olaf College, Northfield, MN 55057
**Electronics for Imaging, 950 Elm Avenue, San Bruno, CA 94066
***Digital Equipment Corporation, 9 Northeastern Boulevard, Salem, NH 03079

The `Trainset` Railroad Simulation

Richard A. Brown[1] and Fred B. Schneider, Editors

Jacob Aizikowitz[2] Thomas C. Bressoud Tony Lekas[3]

The RR Project

Department of Computer Science
Cornell University
Ithaca, NY 14850

February 13, 1993

[1] Department of Mathematics, St. Olaf College, Northfield, MN 55057
[2] Electronics for Imaging, 950 Elm Avenue, San Bruno, CA 94066
[3] Digital Equipment Corporation, 9 Northeastern Boulevard, Salem, NH 03079

Contents

Preface

I became intrigued with real-time systems in the spring of 1986. Here was an application domain where using formal methods is justified, because the cost in life and property of programmer errors could be so great. Here also was an application domain where making assumptions about hardware failure modes is inappropriate, and so a system must be able to tolerate so-called Byzantine failures. My research efforts had been concerned with these two subjects, thus putting me in the enviable position of having discovered a "problem" for my various "solutions".

Attacking a real process-control problem seemed like a good way to get a better understanding of the area. But, which problem? The problem had to be simple enough so that Computer Science issues dominated any application-dependent details. Yet, the problem could not be too simple or else some key aspect of real-time systems might be overlooked. I was aware that various research groups (e.g., at University of Newcastle upon Tyne and at University of Waterloo) had used electric toy trains as a vehicle [sic] for such research, and so that was an obvious place to start. After studying a bit of railroading, however, it became clear that toy trains are not accurate models of reality — they change the problem too much. For example, real trains cannot accelerate or decelerate as rapidly as toy trains do. Therefore, a control program for a real train must anticipate changes to train speed; a control program for a toy train need not. Some of the inaccuracies of toy trains can be corrected by modifying the electronics used to control the trains, but dealing with this and the other custom hardware necessary for integrating a toy train set with a computer system seemed like a black hole (as only custom hardware can be) that I should avoid. Thus, I decided to build a railroad simulator along with support software for constructing railroad layouts, controlling those layouts from networks of computers, and monitoring such control experiments.

After the first version of the railroad simulator was built, it became clear that my circumstances were not unique. Other scientists were also becoming interested in studying real-time programming issues and they, too, felt that a prototype application could be a useful research tool. Courses on real-time systems would benefit from using this software in laboratory exercises, particularly because specialized hardware and software were not required. So, we cleaned-

up the code, wrote some user documentation, and put together this software distribution.

The **Trainset** system, as our railroad simulation software is now known, is the work of many people over the last 5 years. Jacob Aizikowitz defined our model of railroads and wrote the first railroad simulator in the spring of 1987; it ran under SUNOS Unix. Jacob also supervised MEng. students Ellen Blood, Anthony Pellegrini, and Jane Smidesang in producing an X10 graphics interface to the system. Michael Abbott, an undergraduate, then defined and implemented a high-level interface to the simulator for use by control programs. This software was then rewritten and ported to VMS, ULTRIX, and X11/DECWindows by Tony Lekas, a DEC engineer working with us as part of a DEC-funded research project at Cornell. Dick Brown joined the project in Fall 1989, spending that year and the following spring term on a major rewrite of the system and writing documentation for what we had. Dick was assisted by Thomas Bressoud, who rewrote and documented the layout editor and part of the graphics monitor software. Most recently, Donald Wihardja has helped us debug the installation procedure.

This software development effort would not have been possible without financial support from a number of sources. My research in concurrent and distributed systems has been funded by grants from the National Science Foundation since 1978 and from the Office of Naval Research since 1985. Dr. Andre van Tilborg, now the division director for Computer Science at ONR, was especially supportive as my ONR program manager during the initial stages of this project, and Gary Koob, his successor, has continued that tradition. Funding from Digital Equipment Corporation was also critical to the success of this project. Ed Balkovich of DEC encouraged me to apply for funding under Digital's External Research Program (ERP) and then served as our corporate liaison, helping to transfer our research results to engineers at DEC who could benefit from them. The DEC ERP funds allowed us to procure hardware for the laboratory used to develop this software and to run real-time systems experiments. John Gannon, the Software Engineering Program manager at NSF for 1988-89, alerted me to the NSF Software Capitalization Initiative and encouraged me to apply. Funding from that program supported Dick Brown's stay at Cornell and is largely responsible for transforming our research prototype into a system that could be widely distributed.

Fred B. Schneider
Ithaca, New York

Chapter 0

Introduction and Overview

Trainset is a real-time simulation of a railroad. The software consists of a simulator, an interactive graphics editor for defining railroad layouts and graphics monitor programs for displaying the state of the railroad and manually controlling it. Two communications interfaces to the simulator are provided.

- The *control program interface* (CPI) is used by computer programs that control **Trainset** railroads. The CPI consists of library routines and data structures, collectively called the ACI (Automatic Control Interface), and low-level message formats and related facilities, called the LLI (Low-Level Interface).

- The *monitor interface* is used only by the graphics monitor programs.

These interfaces can be in use simultaneously.

Trainset has been implemented in C language on Digital Equipment Corp. Ultrix, using DECwindows/X-11 and TCP/IP or DECnet.

This manual introduces **Trainset** and gives specifications for the railroads it implements. The document also discusses the mechanics of using **Trainset** and provides other information that a researcher or student should know in order to write control programs that interact with **Trainset** railroad layouts.

The manual is organized as follows.

Chapter 1 is a tutorial on using the software. It includes instructions for running a demonstration, creating and simulating a railroad layout and writing programs to control a **Trainset** railroad. A simple programming example is discussed to illustrate the use of the ACI.

Chapter 2 specifies the attributes of simulated railroads that **Trainset** supports.

Chapter 3 discusses the ACI. A high-level mechanism is presented for establishing a connection with a running **Trainset** railroad simulation and receiving an initial-state download of that railroad. Commands and queries are described

for interacting with a **Trainset** railroad. Utility routines that provide timer facilities are introduced, and a service is discussed that supports the writing of fault-tolerant control programs.

Chapter 4 documents the LLI. This information will be of interest to programmers who wish to bypass the ACI layer or reimplement it for other environments.

The appendices include reference pages for the programs that comprise **Trainset** and selected code listings.

A separate installation guide accompanies the software distribution.

Obtaining a Copy

You may obtain a copy of the **Trainset** software, installation instructions and the text of this manual from either of the Internet sites listed below.

- **ftp.cs.cornell.edu** (Cornell University)
- **ftp.stolaf.edu** (St. Olaf College)

In either case, use the file-transfer program **ftp** to open a connection to the desired host. Use the login name **anonymous**, and provide your own Internet address as the password.

After logging in, issue the **ftp** command **cd pub/trainset** to access the **Trainset** distribution directory. Among the files in that directory are:

- **README**, which briefly describes **Trainset** and the contents of the distribution directory,
- **install.dvi**, installation manual for **Trainset** (in TEX output format),
- **ts.dvi**, the text of this manual,[1] and
- **trainset.tar.Z**, the source code package for **Trainset**.

See the manual page for *ftp*(1) for file transfer instructions. Use **ftp** file type **binary** for **.dvi** and **.tar.Z** files. The source code package **trainset.tar.Z** may be unpacked on most Unix systems by issuing the following command:

```
% zcat trainset.tar.Z | tar xvf -
```

See the manual pages for *compress*(1) and *tar*(1) for more information about this unpacking procedure.

[1] To print this manual complete with figures, install **Trainset** at your local site and follow the printing instructions provided in the installation manual.

Chapter 1

Trainset **User's Guide**

1.1 Getting Started

Trainset consists of programs to support interaction with a simulated railroad. Users can control the railroad manually and watch it in action by using *graphics monitor* programs. In addition, computer programs, called *control programs*, may be written to control a **Trainset** railroad.

The rest of this section gives you a chance to get acquainted with **Trainset**. In subsequent sections we explain how to create and run your own railroad layout and how to use the ACI library.

1.1.1 Invoking the Programs

To start the **Trainset** software, enter the following command.

```
% ts
```

(The symbol % is assumed to be your shell prompt.) Three windows will open on your display, as shown in Figure 1.1. One of these, the **Simulator Window**, is a terminal window that displays messages from the simulation. The other two windows make up the graphics monitor. The **Viewer** window shows the current positions of the railroad tracks and trains in a simulation. The **Control Panel** window has controls and indicators, including pushbuttons and slide bars; it is a graphical interface for controlling trains and switch blocks manually.[1]

The sample railroad layout displayed in the **Viewer** window of Figure 1.1 has two trains and 24 blocks.[2] Each train has two ends; one is called the *head*

[1] The names of the programs that comprise **Trainset** are **tsim** for the simulator, **tsview** for the **Viewer**, **tspanel** for the **Control Panel** and **tsed** for the layout editor. **tsed** is discussed in Section 1.2 below.

[2] A careful reader may observe that the 24 block identifiers in Figure 1.1 range from 1 to 5 and 7 to 25. Identifier 6 is associated with a subblock of the cross block **cr 5**, as explained in Section 2.2.3.

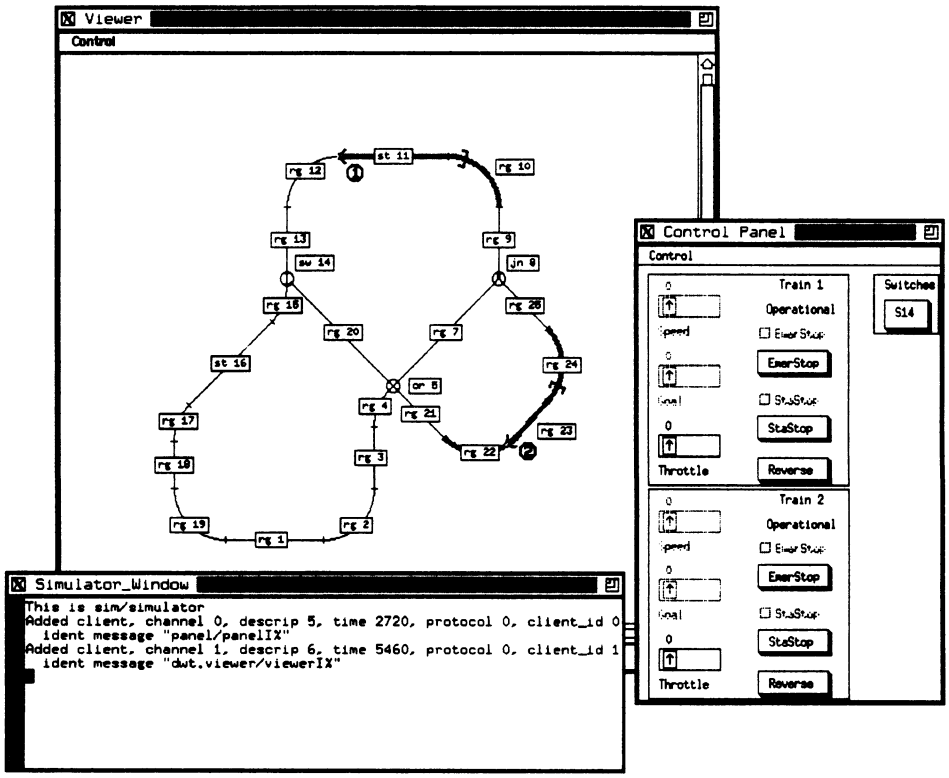


Figure 1.1: Sample railroad layout

(indicated by an angle bracket) and the other is called the *tail* (indicated by a square bracket). A block's type (see Section 2.2) is defined by its label:

`rg` for *regular blocks*,

`st` for *station blocks*,

`jn` for *join blocks*,

`cr` for *cross blocks* and

`sw` for *switch blocks*.

Most blocks in the layout of Figure 1.1 are represented by straight lines or arcs, e.g., those labelled `rg 1`, `rg 2` and `st 11`. Five such blocks are thickened to indicate that they are occupied by a train. Three blocks (`cr 5`, `jn 8` and `sw 14`) have other block types that are represented by circled symbols in the layout.

The control panel comprises one subwindow for each train and a pushbutton for each switch block. Each switch block's pushbutton can toggle that switch block's setting between the straight and turned settings. Each train's subwindow includes the following controls and indicators.

- Two slide bars labelled **Speed** and **Goal** that display the train's current speed and the goal speed desired for that train.
- A slide bar labelled **Throttle** for setting a new goal speed. An operational train accelerates or decelerates when its goal speed differs from its current speed.
- Two labels showing the name of the train (e.g., **Train 1**) and its state (**Operational**, **Derailed** or **Collided**).
- Two indicator lights, **EmerStop** and **StaStop**. The **EmerStop** light is illuminated while the train is performing an emergency stop. The **StaStop** light is illuminated when the train is capable of performing a station stop, described later.
- Three pushbuttons, **EmerStop**, **StaStop** and **Reverse**. The **EmerStop** pushbutton can be used to begin an emergency stop of the train. The **StaStop** pushbutton can be used to enable the station-stop feature for that train. The **Reverse** pushbutton changes the direction of the train's motion if the train is stopped.

1.1.2 Using the Control Panel

The best way to familiarize yourself with the features of the **Control Panel** is to try them, observing their effects as displayed in the **Viewer** window. Below are

some suggested steps for getting acquainted with the system, using the sample layout of Figure 1.1.[3]

When describing graphics manipulations, we will use the following terms for mouse-oriented input operations. The mouse button to press and release is the left mouse button (for standard workstation window managers).

- To *click* on a pushbutton or icon, move the mouse pointer into the pushbutton or icon area, then press the mouse button down and release it immediately.
- To *drag from one point to another* in a window, position the mouse cursor at the starting point, then press the mouse button and **hold it down** while moving the mouse icon to the finish point. Finally, release the mouse button.
- To *drag a slide bar to a value v*, first position the mouse pointer over the inner region that contains the indicator arrow for that slide bar graphic. Then, press the mouse button and **hold it down** while moving the mouse cursor right or left until the desired value v is displayed on the numerical display. Finally, release the mouse button. Due to variation in graphics display resolution, it may not be possible to enter arbitrary desired values using a slide bar.

Before starting, identify the controls and indicators for each train and switch block in the **Control Panel** window. The **Speed**, **Goal** and **Throttle** slide bars for each train display the initial value zero. Each train is operational, and neither of the **StaStop** and **EmerStop** indicator lights is illuminated.

Click (once) on the pushbutton for the switch block **sw 14**.

When you click on the pushbutton, the switch block toggles between the straight and turned settings. Observe that there is a delay after pressing the pushbutton before the switch block setting actually changes on the screen. This delay has two causes: communication time and the time that it actually takes for a switch block, which is a large mechanical device, to change setting.

Click on the switch block pushbutton again to toggle the switch block back to the straight setting.

Click on the **StaStop** and **EmerStop** pushbuttons for **Train 2**. Do not click on the **Reverse** pushbutton for **Train 2** nor any pushbuttons for **Train 1** at this time.

[3]Suggested actions are enclosed in boxes.

When the **StaStop** indicator light is on, we say that train is in *station-stop mode*; likewise, the **EmerStop** indicator light shows whether the train is in *emergency-stop mode*. Note that the **StaStop** indicator light can be illuminated, but the **EmerStop** indicator light cannot be illuminated yet. An operational train's station-stop mode can be enabled or disabled whether that train is moving or not. Emergency-stop mode cannot be enabled for a train unless that train is moving.

We are now ready to set a train in motion.

> Accelerate **Train 1** by dragging the **Throttle** slide bar for that train to a value near 40 (m/sec).

Notice that several things happen when you do this.

- The goal speed indicator, labelled **Goal**, now shows the throttle value.
- The current speed indicator, labelled **Speed**, begins changing from the previous speed (zero in this case) towards the goal speed.
- The train at the top of the **Viewer** window begins moving (forward towards the left, in this case).

The goal speed is not reached instantaneously. A train ordinarily changes speed at a fixed rate of acceleration, as explained in Section 2.3.2, and it takes time to accelerate from 0 to 40 m/sec.

Observe that a block is highlighted in the **Viewer** window if any part of that block is occupied by a train.

> Drag the **Throttle** slide bar to about 55 m/sec, then drag it to about 45 m/sec before the train has completed accelerating to 55 m/sec.

This exercise shows that a new goal speed value overrides a previous one when the throttle is changed, even if a previous goal speed has not yet been reached.

There is a maximum speed limit of 60 m/sec for each block in the sample layout of Figure 1.1. If a train exceeds this limit, that train derails. All minimum speed limits in the sample layout are zero. Using the editor **tsed** (see Section 1.2 below) it is possible to create new layouts having different shapes and block speed limits or to modify the features of an existing layout. However, there is no provision for changing the attributes of a layout while it is being simulated.

Next, request an emergency stop.

> Click on the **EmerStop** pushbutton for **Train 1**.

Observe that the **EmerStop** indicator light is illuminated while an emergency stop is in progress and goes off as soon as the stop has completed. Another way to stop a train is by simply setting that train's throttle to zero. An emergency stop, like a throttle change, overrides any prior motion plan. Thus, during an emergency stop a train's Goal speed is zero and its current speed decreases toward zero. There are two important differences between using the throttle to decelerate to zero and using emergency stop.

- Emergency stops use a larger deceleration rate.
- Nothing can override an emergency stop, other than derailment or collision of the train.

In particular, dragging the **Throttle** slide bar has no effect during an emergency stop. Nor does an emergency stop cause the throttle bar to move. The throttle is simply ignored until the emergency stop has completed and the throttle is dragged again.

Station blocks differ from regular blocks in that they have a *station-stop feature*. To test this feature, perform the following steps.

Click on the **StaStop** pushbutton of **Train 1** so that the **StaStop** indicator light becomes illuminated (station-stop mode).

Drag that train's **Throttle** slide bar to 30 m/sec or less.

The next time that **Train 1** enters a station block (either **st 11** or **st 16**), it will immediately begin slowing down so that it comes to a halt exactly at the opposite terminator of that block. Three conditions are required for a train to begin performing a station stop.

- That train must have station-stop mode enabled.
- That train must have speed at most the station-stop speed.
- That train must be entering a station block.

The station-stop speed (30 m/sec in the sample layout of Figure 1.1) is specified for each station block when a layout is created.

Observe that the station-stop indicator automatically goes off as soon as a station stop begins.

Allow **Train 1** to complete a station stop.

Unlike emergency stops, it is possible to abort a station stop by changing the throttle value for the train involved. If a station stop is aborted, then no station stop is performed until the three station-stop conditions are again met *on entry* into a station block.

A train's direction can only be changed when that train is stopped.

Click on **Train 1**'s **Reverse** pushbutton (once) while that train is stopped.

Then drag that train's throttle slide bar to a positive value, e.g., 30 (m/sec).

Train 1 will begin to move backwards; that is, the head-end retreats and the tail-end advances.

Up to now, both trains have remained in the **Operational** state. In order to become acquainted with another train state, try the following.

Let **Train 1** travel (backwards) around the layout until it derails at block **jn 8**.

When a train enters a join block (such as **jn 8**) from the tail terminator (the one attached to **rg 9** in the sample layout), then the train will derail. If a train derails, the state label for that train changes to **Derailed** and the train stops moving.

Trains that are not **Operational** do not respond to commands. Thus, once a train derails, there is no way to set that train in motion again, short of starting another simulation. Page 35 lists all the conditions under which a train will derail.

Click on the pushbutton for switch block **sw 14** once so that the switch block changes to the turned setting.

Drag the **Throttle** slide bar for **Train 2** to a positive value, e.g., 30 m/sec.

Observe that attempting to enter a disconnected terminator of a switch block would cause a train to derail.

Finally, observe what happens when trains collide.

Allow **Train 2** to continue until it collides with **Train 1**.

The state labels of both trains involved in a collision change to **Collided**. Henceforth, neither train will respond to any commands, so the demonstration has ended—in disaster! Fortunately, simulated trains are inexpensive to replace.

Page 36 lists all possible collision conditions.

1.1.3 Shutting the Programs Down

The programs that comprise **Trainset** may be shut down by selecting **Quit All** in the **Command** menu of the **Control Panel** or **Viewer**.

Exercises

1. Start a simulation of the default layout by issuing the `ts` command. Set both trains in motion around the track simultaneously, with `Train 1` travelling forward and `Train 2` travelling backward.

2. Perform Exercise 1. Then, begin toggling the setting of the switch block `sw 14` so that `Train 1` always passes through `sw 14` when `sw 14` is straight and `Train 2` always passes through that switch block when it is turned.

   Note: This is not as easy as it may sound! Keeping both trains travelling along different paths in this layout is a challenge, particularly if both are moving as fast as possible. Train speed adjustments and switch block setting changes must be coordinated and must be issued far enough in advance so that the trains neither collide nor occupy a switch block while it changes setting.

1.2 How to Build a Layout of Blocks and Trains

tsed is an interactive graphics editor for defining and modifying railroad layouts.

1.2.1 Invoking tsed

To start **tsed**, enter the following command.

```
% tsed [filename]
```

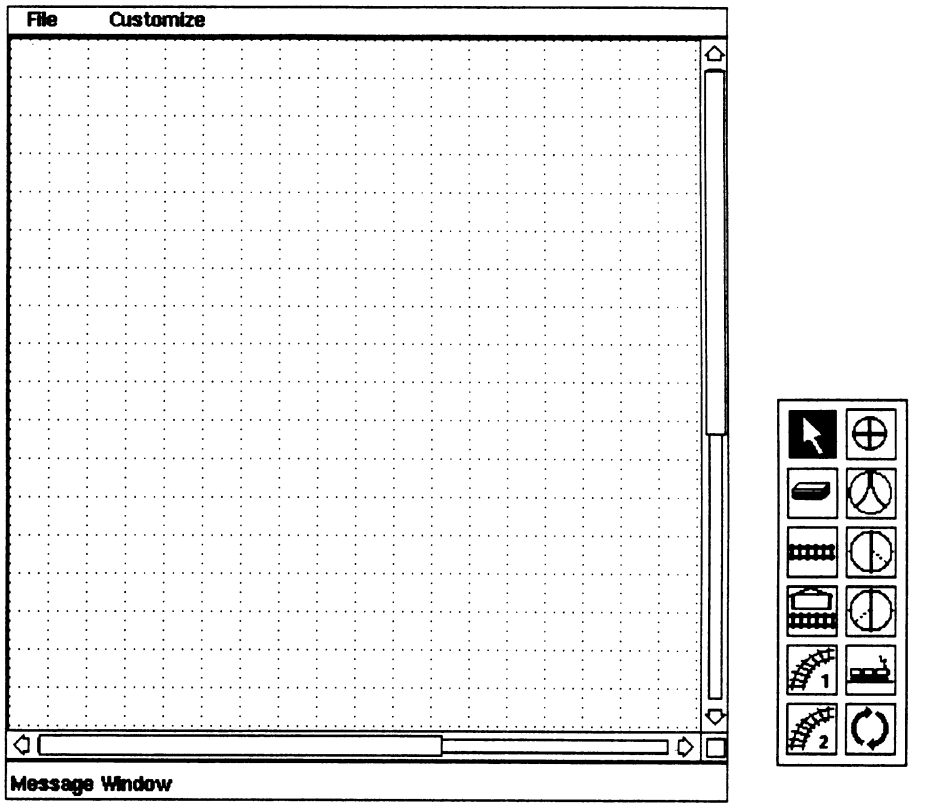
The editor may also be invoked by entering:

```
% ts -edit [filename]
```

When **tsed** is started, two windows appear on your display, as shown in Figure 1.2. The larger window that is overlaid with a grid pattern is the *canvas window* (Figure 1.2a). A railroad layout can be created in the canvas window. The distance between two adjacent parallel dotted lines in the grid is called a *grid division.*

The smaller window is called the *tools window* (Figure 1.2b). It consists of twelve icons representing operations called the *tools* that are available in **tsed** for creating and modifying blocks and trains. The tools are applied using the mouse operations described on Page 6. Figure 1.3 shows the tools window together with the names of its tools.

The tools window contains one or more tools for each of the five types of blocks in **Trainset**. Note that the tools window includes two tools for creating switch blocks. The **Switch Block 1** and **Switch Block 2** tools differ in the orientation of the switch block; each is a mirror image of the other. Also, there are three tools for creating regular blocks: **Straight Block**, **Arc Block 1** and **Arc Block 2**. A *straight block* is a regular block consisting of a line segment, and an *arc block* is a regular block consisting of a circular segment. **Arc Block 1**



(a) Canvas Window **(b) Tools Window**

Figure 1.2: `tsed` Windows

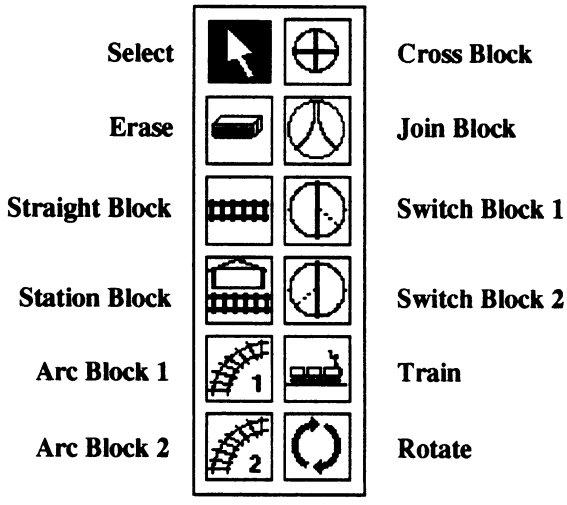


Figure 1.3: Annotated Tools Window

and **Arc Block 2** differ only in the radius of the arcs they create. The **Arc Block 1** tool creates arcs with radius two grid divisions, and the **Arc Block 2** tool creates arcs with radius four grid divisions.

The remainder of this section is a two-part tutorial on using **tsed** for creating and editing railroad layouts. Section 1.2.2 introduces you to the various **tsed** tools. A series of exercises demonstrates how to construct a layout and how to make simple editing changes. Section 1.2.3 explains how to set attributes such as block speed limits, how to save a layout in a file, and how to exit from the editor.

1.2.2 Using the tsed tools

The Current Tool

In **tsed**, one tool is enabled at any given time; it is called the *current tool.* The current tool is indicated in the tools window by being highlighted. The message area at the bottom of the canvas window also displays the name of the current tool. On startup, **Select** is the current tool.

> Click on a tool icon other than **Select** to choose a different current tool. Repeat one or more times.

Regular and Station Blocks

Straight regular blocks and station blocks are created in a layout by dragging with the mouse from one point to another in the canvas window when the corresponding straight block or station block is the current tool. This procedure creates a line segment between the starting and stopping points of the drag operation. The starting point is called the *head terminator* of the block, and the stopping point is called the *tail terminator.*

> Click on the **Straight Block** tool icon.

> Create a horizontal regular block by dragging from right to left starting near the middle of the canvas window, as shown in Figure 1.4a.

All linear blocks (**Straight** and **Station**) created by **tsed** are constrained to be vertical, horizontal or at a $\pm 45°$ diagonal. To aid in alignment, **tsed** enforces additional constraints on the placement and length of a block relative to the dimensions of the canvas window grid.

> Click on the **Station Block** tool icon.

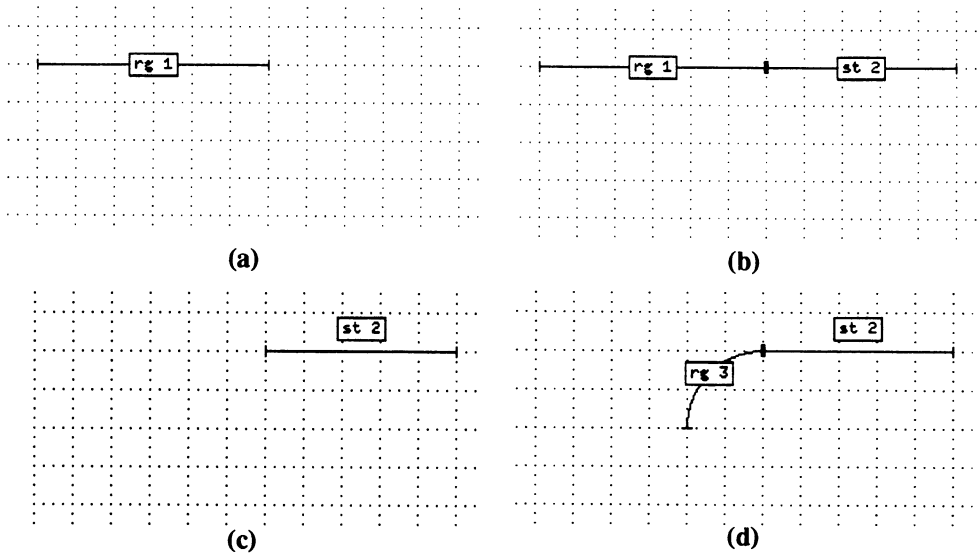


Figure 1.4: Tutorial Steps

Now create a station block in the canvas window by dragging from the right endpoint of the existing straight block to a point further to the right, as shown in Figure 1.4b.

When a new block is created, **tsed** establishes a connection with an existing block if the new head terminator is within a few pixels of an existing terminator. **tsed** also constrains the slope of the new block to match the slope of the existing block at that terminator.

Station blocks and straight regular blocks appear graphically to be identical except for their labels. Regular blocks (whether straight or arc) are labelled **rg**, and station blocks are labelled **st**.

The **Erase** and **Select** tools are used for modifying objects already on the canvas window. **Erase** enables you to remove a block or train that you have created. **Select** enables you to reposition the label for a block or to designate a block whose attributes you wish to change.

Click on the **Erase** tool icon.

Now, erase the straight block by clicking on it.

Click on the **Select** tool icon.

Now drag the label for the station block to a new position above the block.

The result of these changes are shown in Figure 1.4c.

Arc blocks are created by dragging when the current tool is **Arc Block 1** or **Arc Block 2**. The starting point of the drag operation determines the location of the arc block's head terminator. The extent of the arc is determined by the ending point of the drag operation and depends on the location of the head terminator and, if the arc block is attached to another block, the slope of that block.

Click on the **Arc Block 1** tool icon.

Create an arc block connected to the station block by dragging from the left endpoint of the station block toward a point that produces a 90° arc pointing downward, as shown in Figure 1.4d.

When using the **Straight Block**, **Station Block**, **Arc Block 1** or **Arc Block 2** tools, you can cancel creation of a block after a drag operation has already been initiated by finishing that drag operation within a few pixels of its starting point.

Start dragging to create another arc block connected to the last arc block, then cancel the creation of a new block by finishing that drag operation at its starting point.

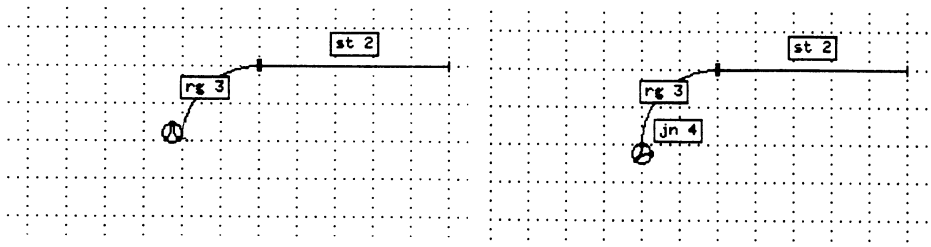
Iconic Blocks: Crosses, Joins and Switches

The **Cross Block**, **Join Block**, **Switch Block 1** and **Switch Block 2** tools always create blocks that have a fixed size and shape. In **tsed**, these blocks are referred to as *iconic blocks*. They are created by clicking with the mouse rather than by dragging.

When an iconic-block tool is current and the mouse cursor is in the canvas window, the cursor takes the form of the icon for the current tool. Along the outer circle of such a cursor are enlarged points called *hot spots* at which connections with other blocks can be made.

Click on the **Join Block** tool icon. Observe that the mouse cursor changes to the form of a join block whenever the cursor is in the canvas window.

Position the mouse cursor so that the hot spot at its right head is over the unattached terminator of the arc block. This requires slight overlapping between the arc block and the join-block cursor; see Figure 1.5a.



(a) Before Clicking **(b) After Clicking**

Figure 1.5: Join Block Creation

Now click with the mouse and create the join block of Figure 1.5b.

Note that the newly created join block is constrained so that the slope of the join block and the slope of the arc block agree at their common terminator.

tsed does not make more than one connection when an iconic block is created. Thus, during an editing session, it is probably best to create iconic blocks before creating the regular and station blocks they are attached to.

The **Rotate** tool enables you to rotate an iconic block clockwise by one hot spot.

Click on the **Rotate** tool icon.

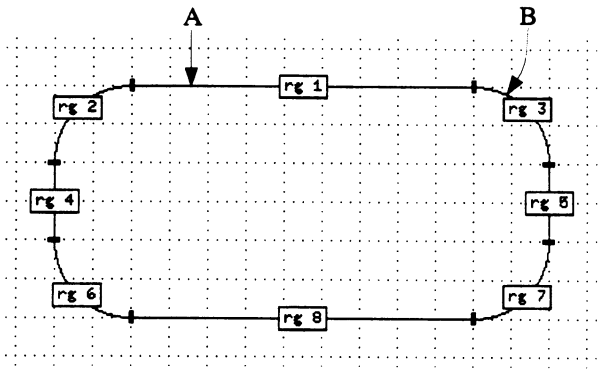
Now click on the join block created in step 15. Observe that the join block labelled **jn 4** rotates so that its tail terminator becomes the terminator attached to the arc block.

Click on the join block a second time to rotate again.

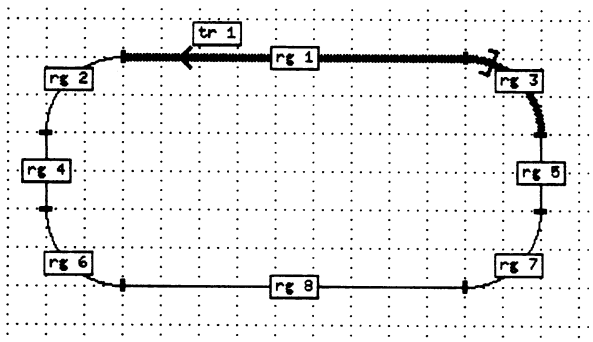
If an iconic block is not connected to any other block, the **Rotate** tool rotates it by one-eighth turns. If a switch block is rotated through a full circle using the **Rotate** tool, it changes orientation.

Trains

A train may be created in a layout by dragging when **Train** is the current tool. The drag operation begins at the position desired for the head end of a train, continues along the blocks to be occupied by that train, and stops at the position desired for that train's tail end. The head end of a train is indicated in the layout by an angle bracket, and the tail end is indicated by a square bracket. All blocks occupied by a train are highlighted. The head end of a train is constrained by **tsed** to start in a regular or station block.



(a) Before Train Creation



(b) After Train Creation

Figure 1.6: Oval Layout

Erase all blocks currently on the layout. Then, create an oval layout as in Figure 1.6a made of four straight blocks and four arc blocks, using the **Straight Block** and **Arc Block 1** tools.

It is not necessary that the block identifiers, i.e., the numbers in the labels, match those in Figure 1.6a.

Click on the **Train** tool icon.

Create a train on the oval by dragging clockwise from the point marked A to that marked B in Figure 1.6a, resulting in Figure 1.6b.

If, while creating a train, you drag across the starting point, the brackets that indicate train ends reverse their directions. Also, observe that a whole block is highlighted if any part of that block is occupied by a train.

You should now feel comfortable with the basic drag and click operations for creating blocks and trains in a layout. In the next part of this `tsed` tutorial, you will learn how to store a layout in a file, how to specify attributes such as speed limits for individual blocks, and how to exit `tsed`.

If you would prefer to return to this tutorial at a later time, select `Quit`[4] from the `File` menu now, and invoke `tsed` again when you are ready for the remainder of the tutorial. There is no need to save your present work, since it will not be used in the second part of the tutorial.

1.2.3 Creating a figure-eight layout file

Our goal is to create the figure-eight layout illustrated in Figure 1.7 and then save that layout in a file `mylayout.l`. All blocks in the layout should have minimum speed of 0 (m/sec) and maximum speed of 70, except that the station blocks at the top and bottom of the layout are to have minimum speed of 0 and maximum speed of 50.

Select `New` from the `File` menu. If you are continuing from the first part of the tutorial, `tsed` will ask if you wish to discard changes in the canvas window; click on the `Yes` button. In response to the prompt, enter `mylayout` as the name of the file to be created.

By invoking the `New` command above, `mylayout.l` becomes the *current file name*. This name is indicated in the title for the canvas window. Prior to the `New` command there was no current file name. When naming a file, the extension `.l` is automatically appended by `tsed` if you do not include it. Pathnames that

[4] `Quit` will be discussed in detail on Page 21.

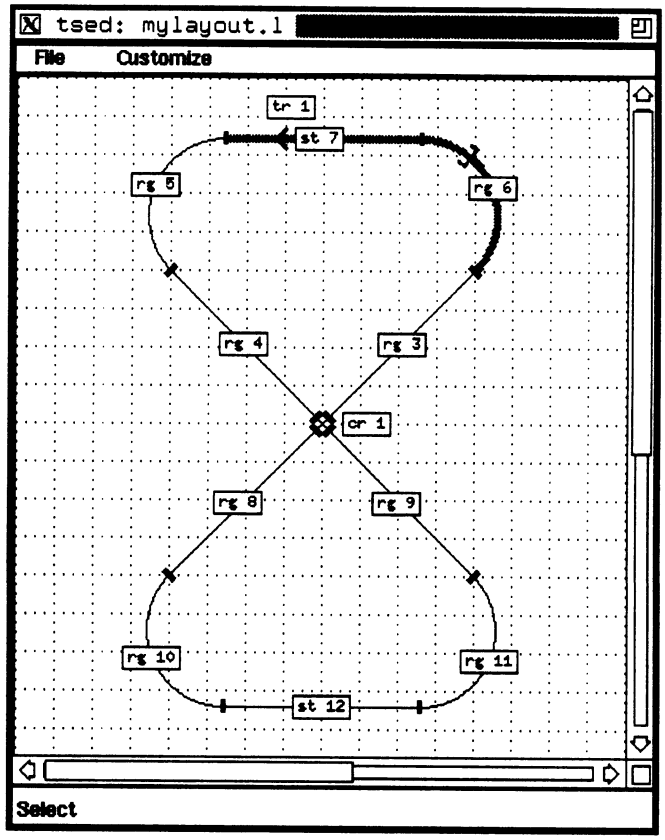


Figure 1.7: A figure-eight layout

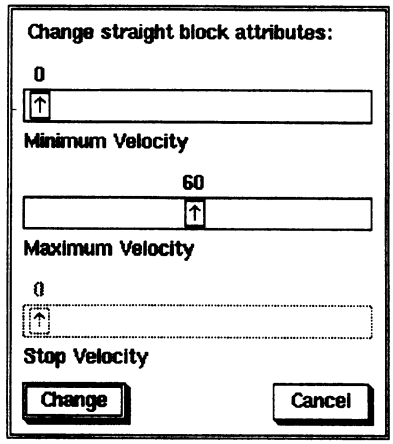


Figure 1.8: **Change Default Block Attributes** subwindow

do not begin with a slash '/' are interpreted relative to the current working directory.

The default minimum and maximum speeds for blocks can be changed by choosing the **Change Default Block Attributes** entry from the **Customize** menu. A changed default value applies to blocks created after the change; it does not affect blocks already created. **Change Default Block Attributes** is itself a menu whose subentries are the various block types.

> Change the default attributes for straight blocks by choosing **Change Default Block Attributes** from the **Customize** menu and moving the cursor to the right until the subentries appear. Select the **Straight** subentry.

After this step, a subwindow will appear that contains three slide bars as shown in Figure 1.8. The slide bar on the top determines the minimum speed for straight blocks and is normally 0. The slide bar in the middle determines the maximum speed, which is 60 at present. The slide bar on the bottom is disabled for straight blocks since it defines the station stop speed, an attribute that is only applicable to station blocks.

> Set a new maximum speed by dragging the middle slide bar to 70. Leave the minimum speed at 0.

> Click on the **Change** pushbutton to make the new default speed limits effective.

The steps above change the default maximum speed for straight blocks only. We also need arc blocks and cross blocks that have a maximum speed of 70.

Change the default maximum speed to 70 for arc blocks, using the **Arc** subentry in the **Change Default Block Attributes** menu under **Customize** and proceeding as above.

Also change the default maximum speed to 70 for cross blocks.

You are now ready to create the blocks of your layout. Refer to Figure 1.7 for illustration of the steps below.

Create a cross block near the center of the layout at an intersection point of two perpendicular dotted lines in the grid.

Rotate the new cross block once using the **Rotate** tool. The subblocks of the cross should form an '×' as opposed to a '+'.

Create two straight blocks of exactly the same length, each forming the diagonal of a square with sides about 4 grid divisions long, so that the head terminators of the straight blocks are attached to the upper hot spots of the cross block.

Note that the grid assists you in determining when blocks have exactly the same length.

Create two arc blocks with 135° extent and head terminators attached to the straight blocks that you just created.

Create a station block that connects to both unattached terminators of the arc blocks.

The attributes of an individual block may be customized by selecting that block with the **Select** tool and choosing **Change Block Attributes** from the **Customize** menu.

Click on the **Select** tool icon.

Click on station block **st 7** in the canvas window to select it for customization.

Choose **Change Block Attributes** from the **Customize** menu.

In the subwindow that appears, set a new maximum speed by dragging the slide bar in the middle to 50.

The station stop speed is another attribute that may be adjusted for station blocks using **Change Block Attributes**. (See Section 2.2.5 for a discussion of the station stop speed.) We will leave the default value at 30 for this layout.

Create the lower portion of a figure-eight layout using a procedure similar to the previous seven steps.

Place a train on the figure-eight layout.

You have now created the desired figure-eight layout. It is time to save your work and exit from the editor.

Save the figure-eight layout by choosing **Save** from the **File** menu.

Either the **Save** entry or the **Save as** ... entry in the **File** menu may be used to save your work in a file. The difference is that **Save as** ... always prompts you for a file name, while **Save** uses the current file name if there is one.

Finally, end the editing session:

Exit **tsed** by choosing **Quit** from the **File** menu.

The **Quit** command checks for any unsaved changes before exiting. If any are found, **Quit** gives you the option of writing them to a file first. The **Close** command is similar to **Quit**, except **tsed** does not exit after checking for unsaved changes. Instead, a **Close** causes **tsed** to enter a state with an empty canvas window and no current filename.

1.3 How to Run a Layout

A railroad layout **mylayout** created using **tsed** can be simulated by issuing the following command.

```
% ts -layout mylayout
```

A **Trainset** simulation will start and search for a file called **mylayout.l**, first searching in the current working directory, then in the standard location for layouts, as explained in the manual page for **ts** in Appendix D.

1.4 Control Programs

The Automatic Control Interface (ACI) is a library of procedures and data structure definitions for writing programs that control a **Trainset** railroad.

The principal data type associated with the ACI, **LayoutData**, represents various attributes of the blocks and trains in a railroad layout. **LayoutData** is described in Section 3.2 and defined in Appendix B.1.

The ACI routines may be classified into five categories.

1. The ACI **GetDownload** procedure establishes a network connection with a running railroad simulator and receives a report of the state of the railroad being simulated. **GetDownload** returns a pointer to an internally allocated data structure of type **LayoutData** holding the state information that has been received. See Section 3.2 for further details.

2. ACI commands enable a program to change certain attributes of blocks and trains. Seven command types are available.

   - **SetSwitch** initiates a setting change for a switch block.
   - **Accelerate** and **Decelerate** initiate changing the speed of a train at a constant rate for a specified time period.
   - **SetSpeed** initiates changing the speed of a train towards a specified goal speed.
   - **SetDirection** changes the direction of a train that is stopped.
   - **EmergencyStop** halts a train using a deceleration rate that is quicker than the rate for **Decelerate**.
   - **StationStop** enables a train to come to a complete stop at a known location in a station block, provided that the train enters that station block slowly enough.

   See Section 3.3 for more details about ACI commands.

3. ACI queries enable a control program to obtain state information about a railroad after a download has been received. Four query types are available.

   - **GetBlockOccupancy** indicates whether a specified block is occupied.
   - **GetSwitchPosit** returns the setting of a switch block.
   - **GetTrainStatus** indicates whether a train is operational.
   - **GetTrainMotion** indicates whether a train is moving.

   See Section 3.4 for more information.

4. ACI voting service procedures, **SetSeqNumber** and **NewSeqNumber**, interact with a command arbitration facility in the simulator. This service is useful when implementing fault-tolerant control programs. See Section 3.5.

5. ACI utility procedures, **InitTimer**, **GetTimer**, **AwaitTimer**, **CancelTimer** and **Sleep**, provide high-level general purpose timing facilities. See Section 3.6.

```
/* demo.c -- simple demonstration of the ACI */

[1] #include "aci.h"  /* ACI definitions and declarations */

#define MAX_HOSTNAME  100
[2] #define POLLING_INTERVAL   0.100  /* Seconds */
#define POLLING_TIMEOUT    200.0  /* Seconds */
#define NULL (void *) 0   /* generic null pointer */

int poll_timeout_flag = 0;  /* flag for terminating polling loops */

/*****************************************************************************
 * set_poll_timeout_flag is executed by a timer that is used to prevent
[3]  * infinite polling loops.
 *****************************************************************************/

void
set_poll_timeout_flag()
{
  poll_timeout_flag++;
}


/*****************************************************************************
 *
 * This program shows the mechanics of the ACI layer of the control
 * program interface, by initiating a connection with the simulator,
 * receiving the state of the railroad, moving a train, and making some
 * queries.
 *
 *****************************************************************************/

main(argc, argv)
     int argc;
     char *argv[];
{
  char *progname;  /* name of this program */
  char *hostname = "";  /* host that is running simulator */
  int simnum = 0;  /* distinguishes between simulators running on same host */
  Seconds timeout = 20.0;  /* maximum second to connect to simulator */
  LayoutData *datap;  /* pointer to entire received railroad state info */
```

Figure 1.9: `demo.c`, an example using the ACI (beginning).

```
4   TrainData *tp;  /* pointers to parts of received layout data */
    BlockData *bp;
    int T = 1, B = 1;  /* identifiers of a train and a block */
    enum Occupancy occ;  /* return values from queries */
    enum TrainMotion tm;
    int n;  /* loop counter */


    /*
     * Collect command line args
     */

    progname = *argv++;
    if (argc)
      hostname = *argv++;
    if (argc)
      simnum = atoi(*argv++);
    if (argc)
      timeout = atof(*argv++);

    /*
     * Connect and get the state of the railroad.
     * Then, use received values to check the identifiers T and B.
     */

5   datap = GetDownload(hostname, simnum, progname, timeout);
    if (datap == (LayoutData *) 0) {
      printf("couldn't get initial download!\n");
      exit (1);
    }

6   if (T <= 1 || T >= datap->train_ct) {
      printf("sample train ID %d out of range [1..%d]!\n", T, datap->train_ct);
      exit (1);
    }
    if (B <= 1 || B >= datap->block_ct) {
      printf("sample block ID %d out of range [1..%d]!\n", B, datap->block_ct);
      exit (1);
    }
```

Figure 1.9: **demo.c**, an example using the ACI (continued).

```
/*
 * Print some sample values.
 */

printf("Number of trains:  %d.  Number of blocks:  %d\n",
       datap->train_ct, datap->block_ct);

tp = &datap->trains[T];
printf("Train %d has length %.2f, with front at block %d offset %.2f\n",
[7]    tp->length, tp->front.block, tp->front.offset);

bp = &datap->blocks[B];
printf("Block %d has length %.2f, max speed %.2f and min speed %.2f\n",
       bp->length, bp->max_speed, bp->min_speed);
if (bp->type == BT_REGULAR)
  printf("This is a regular block, connected to blocks %d and %d\n\n",
[8]      bp->t.rg.tail, bp->t.rg.head);

/*
 * Perform some accelerations and decelerations of train T.
 */

printf("beginning acceleration\n");
[9]  Accelerate(T, 20.0);
printf("pausing...\n");
[10] Sleep(25.0);

printf("beginning deceleration\n");
[11] Decelerate(T, 5.0);

/*
 * Travel until block B is occupied (or timeout occurs)
 */

printf("polling until block %d is reached...\n", B);
poll_timeout_flag = 0;
InitTimer(POLLING_TIMEOUT, 0, set_poll_timeout_flag);

[12,13] while ((occ = GetBlockOccupancy(B)) == OC_FREE && !poll_timeout_flag)
  Sleep(POLLING_INTERVAL);
```

Figure 1.9: `demo.c`, an example using the ACI (continued).

```
14      if (occ == OC_ERROR) {
          printf("error getting block occupancy!\n");
          exit (1);
        }
15      if (poll_timeout_flag) {
          printf("polling timed out after %f seconds\n", POLLING_TIMEOUT);
          exit (1);
        }
        CancelTimer();
16,17   /* assertion:  occ == OC_OCCUPIED */

        /*
         * Perform an emergency stop, then poll until train stops
         */

        printf("performing emergency stop\n");
18      EmergencyStop(T);

        printf("polling until train stops...\n");
        poll_timeout_flag = 0;
        InitTimer(POLLING_TIMEOUT, 0, set_poll_timeout_flag);

19      while ((tm = GetTrainMotion(T)) == TM_MOVING && !poll_timeout_flag)
          Sleep(POLLING_INTERVAL);

        if (tm == TM_ERROR) {
          printf("error when querying about train motion!\n");
          exit (1);
        }
        if (poll_timeout_flag) {
          printf("polling timed out after %f seconds\n", POLLING_TIMEOUT);
          exit (1);
        }
        CancelTimer();
        /* assertion:  tm == TM_STOPPED */

        printf("train has stopped.\n");

        exit (0);
      }
```

Figure 1.9: `demo.c`, an example using the ACI (concluded).

The example program **demo.c** (see Figure 1.9) illustrates the use of the ACI. This program connects to a simulation and receives the state of a railroad, then attempts to move a train in that railroad to a certain block and perform an emergency stop. Some key points about the code are indicated by numbers [1], [2], etc.

[1] The include file **aci.h** contains declarations and definitions required for compiling a source file that uses the ACI. In order to create an executable, one must link with the ACI library **libaci.a**.

[2] Time values passed to the ACI timer procedures are always floating-point quantities representing seconds. An ACI type **Seconds** is defined for such quantities.

[3] **set_polling_timeout** is the procedure that will be invoked if a **tsuser** timer expires.[5] Such a procedure cannot be invoked with arguments, so it changes a global variable **polling_timeout** in order to inform the main program about timer expiration.

[4] The file **aci.h** defines a number of data types besides **LayoutData**. The type **TrainData** is used to represent download information received for a train; likewise, **BlockData** represents download information for a block. Both **TrainData** and **BlockData** are component types used in the definition of **LayoutData**. Enumerated types, including **Occupancy** and **TrainMotion**, are used for command arguments and query return values. Trains and blocks have unique positive integer identifiers.

[5] **GetDownload** takes four arguments, returns a null pointer on failure, etc. Chapter 3 is the reference for this and the other ACI procedures.

[6,7,8] Several examples of references to a **LayoutData** structure follow the invocation of **GetDownload**. More direct references such as

```
datap->trains[T-1].length
datap->blocks[B-1].t.rg.tail
```

could be used in place of those that involve **tp** and **bp** in **demo.c**. Observe that the index of a train in **datap->trains** is always one less than that train's identifier. A similar remark holds for blocks.

The function **PrintDownload** in Appendix B.2 includes examples of references to every component in a **LayoutData** structure.

Checking the values of **T** and **B** at [6] is not strictly necessary in this program, since their values are known to be valid in this case. It is a good defensive programming practice to include such checks anyway, as protection against future changes.

[5]Note: As explained in Section 3.6, a call to **InitTimer** overrides any prior calls. Thus, we speak of "the ACI timer," because only one such timer can be in effect at any time.

[9] ACI commands such as **Accelerate** are non-blocking and return no values. They print no warnings about unreasonable arguments. The command

```
Accelerate(T, 20.0);
```

requests that train T accelerate for a duration of 20 seconds, increasing its speed during that period at the predefined fixed acceleration.

[10,11] The instruction

```
Sleep(25.0);
```

causes **demo.c** (*not* the **Trainset** simulator!) to suspend execution for 25 sec. This is long enough that the subsequent **Decelerate** command [11] is unlikely to interfere with the prior **Accelerate** command [9].

The train would begin to reduce its speed as soon as the **Decelerate** command is received, even if the prior acceleration had not been in effect for the acceleration's entire duration. For example, if the **Sleep** had been for 10 seconds instead of 25, then train T would accelerate for about 10 seconds, then decelerate for 5 seconds.

[12] The query **GetBlockOccupancy** is used so that train T may reach block B. That query is issued frequently until block B is found to be occupied or until an error or timeout occurs. This technique of frequent querying is called *polling*. The ACI timer is set up before the **while** statement to provide timeout mechanism for leaving that loop.

[13] The loop guard condition shows that there are two ways to leave the loop.

- If occ differs from **OC_FREE**, i.e., occ has value **OC_OCCUPIED** or **OC_ERROR**, then the loop will be exited. (Query errors can be caused by invalid arguments, loss of communication with the railroad, etc.)
- If **polling_timeout** has a non-zero value, then the loop will be exited. **polling_timeout** changes from zero to a non-zero value if the timer expires (see [3] and the invocation of **InitTimer**).

[14,15,16] Before concluding that `occ = OC_OCCUPIED`, which would indicate occupancy of block B, it is necessary to eliminate the other events that can cause the loop to exit.

[17] A period of time elapses between the moment that block B becomes occupied and the time when the control program **demo.c** can act on that information. This time period arises from the following causes.

- **Delay from polling.** Some time necessarily elapses between the moment that block B becomes occupied and the time when that sensor is checked. The sensor is checked by each successful call to the

query `GetBlockOccupancy`. Thus, if `GetBlockOccupancy` succeeds this period is bounded by the execution time of one iteration of the polling loop, except possibly when block `B` is found to be occupied on the first `GetBlockOccupancy` query.

- **Network delay.** This is the time required for a sensor value to be communicated to the control program.
- **Local processing delay.** Once the process that is running `demo.c` receives a communication that block `B` was occupied, several further steps occur: the query function returns; the polling loop exits; checks are performed in order to deduce that block occupancy was in fact the reason for loop exit; and the timer is cancelled. Each of these steps requires local processing time.

Such delays can affect the correctness of control programs. For example, the occupancy of a block might change by the time that a control program could take action on that occupancy information. In particular, it is not correct to conclude that block `B` is *currently* occupied based solely on the fact that `occ` has the value `OC_OCCUPIED` at [17].

[18] The command **`EmergencyStop`** causes a train to reduce its speed to zero quickly. Unlike **`Accelerate`** and **`Decelerate`**, the effects of **`EmergencyStop`** cannot be interrupted by another command.

[19] Polling with the query **`GetTrainMotion`** is used in **`demo.c`** to determine when the train has come to a complete stop.

Exercises

1. Write a program that uses the ACI to cause both trains in the sample layout (see Figure 1.1) to move around the track for five minutes, then stop. (Compare to Exercise 1 of Section 1.1.2.) The trains *need not* travel on different loops.

2. As in the loop [12], a timer is set up just before the **`while`** statement [19] in order to guard against an infinite loop. Is this timer necessary, or is that loop certain to exit in a reasonable amount of time without a timer? (Hint: Consider the specifications of **`GetTrainMotion`** in Section 3.4.4.)

Chapter 2

Trainset **Railroads**

2.1 Introduction

Trainset railroads are idealized versions of real railroads. This chapter discusses the attributes and operation of **Trainset** railroads. You will see that while **Trainset** railroads are simpler than their real-life counterparts, the simplifications are ones that do not make it appreciably easier to write programs to control the railroad.

2.2 Blocks

In a **Trainset** railroad, a *layout* consists of an assembly of *blocks* together with movable *trains* that occupy some of those blocks. See Figure 1.1 for an example.

Every block is assigned a unique *block ID* B, a *length* L_B, a *maximum speed limit* MX_B and a *minimum speed limit* MN_B. We expect[1]

$$0 \leq MN_B \leq MX_B \leq MAXFLOAT.$$

Each block has a set of *terminators* that delimit the track implementing that block.

As in real railroads, each block has an associated sensor called a *track circuit*, that indicates whether that block is occupied by a train. Polling a track circuit only returns one bit of information signifying whether the associated track block is occupied. Note that it is not possible to learn the exact location of a moving train from the information returned by a track circuit.

A block may have between two and four terminators, depending on its type. There are five types of train blocks: regular, join, switch, cross and station blocks. These are illustrated in Figures 2.1–2.5.

[1] *MAXFLOAT* is the largest floating point number on the computer system. The standard *mks* units of measure are used throughout this document.

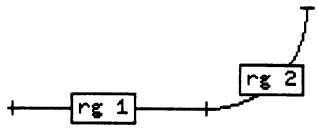


Figure 2.1: Regular Blocks

Two blocks are *attached* to each other if each of the blocks has a terminator that is *associated* with the other block. **Trainset** block layouts are subject to the following restrictions.

- A terminator may be associated with at most one block.
- A block may not be attached to itself.
- Blocks may not be attached to each other more than once, e.g., circular track configurations that consist only of two half-circle blocks are prohibited.

These restrictions make it simpler to describe attachments between blocks, but do not limit the topology of **Trainset** layouts. For example, we could easily construct a circular track configuration using two quarter circles and a semicircle.

It is not necessary that all terminators of a block be attached to other blocks. Of course, a train will derail if it attempts to exit from a block at an unattached terminator.

2.2.1 Regular Blocks

Each regular block has two terminators. Such a block can have the shape of a line segment or a circular arc. See Figure 2.1.

2.2.2 Join Blocks

A join block has three terminators and allows two train routes to merge. Two of the terminators, called the *heads*, are each connected to the third terminator, called the *tail*.

A train can occupy the track between either of the heads and the tail. Trains can enter a join block at either head and exit through the tail. A train derails when it either enters a join block at the tail or leaves from one of the heads.

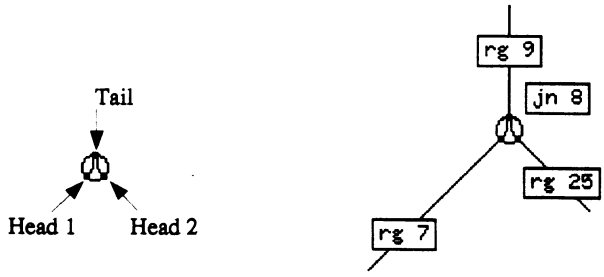


Figure 2.2: Join Block

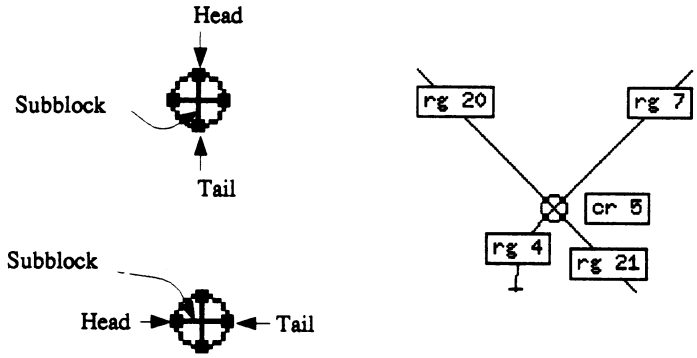


Figure 2.3: Cross Block

Join blocks are assumed to be LEN_{join} meters long measured from the tail to either head terminator.[2] At most one train may occupy a join block at any time. If a second train enters a join block that is already occupied, then a collision occurs.

2.2.3 Cross Blocks

A cross block (see Figure 2.3) allows a track to intersect itself, so that "figure eights" and related designs may be built. A cross has four terminators, and a train may only travel between any opposing pair—turns are not allowed. Thus, cross blocks behave like two short regular blocks, called *subblocks*, with the

[2]Constants such as LEN_{join} are determined at the time of installation. See the installation manual for details.

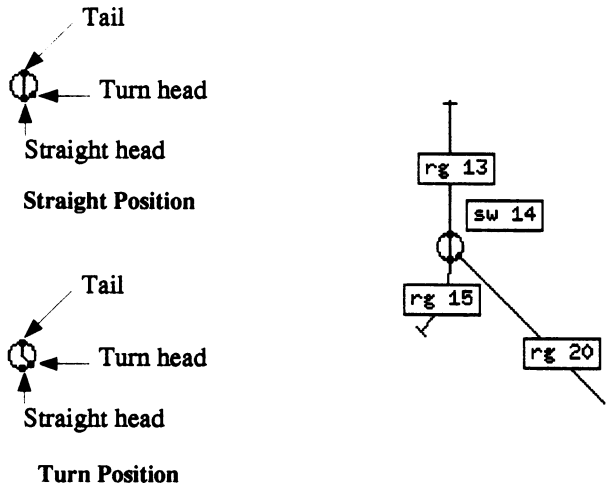


Figure 2.4: Switch Block

same length and speed limits, that have been overlaid at their midpoints. A cross block and both of its subblocks are occupied whenever either of those subblocks is occupied. Cross subblocks are LEN_{cross} meters long. A collision occurs when a cross block becomes occupied by two or more trains.

In order to distinguish between the subblocks of a cross block, each has its own block ID. One of these also serves as the cross block's ID.

2.2.4 Switch Blocks

A switch block (see Figure 2.4) has three terminators, labelled *tail*, *straight head* and *turn head*, and consists of a movable track segment that has two settings and a mechanism to move this track between those settings. Depending on the setting of the movable track, either the train may travel between the tail and straight head (in either direction) or between the tail and turn head. Thus, a switch block can be part of two different train routes, according to that switch block's setting.

A *switching time* T_{switch} must elapse for a switch block to change from one setting to the other. The switch setting is undefined during this time period. If a switch is commanded to change to one setting (e.g., straight) while it is already in that setting or in the process of changing to that setting, then that setting-change command has no effect. If the setting-change command was to switch to the *other* setting, then the switch begins changing to the new setting. It takes a full T_{switch} seconds to change to a new setting, even if the switch

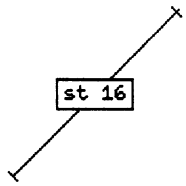


Figure 2.5: Station Block

setting was undefined at the time of the setting change command.

A train attempting to traverse a switch block will derail under the following circumstances.

- The switch block has undefined setting.
- A switch change is attempted on the switch block.
- The train enters the turn head terminator of a straight switch, or the straight head terminator of a turned switch.

Switch blocks are LEN_{switch} meters long measured from the tail to either of the other terminators. At most one train may occupy a switch block at a time or a collision will occur.

A switch block that is occupied by a derailed or collided train will not respond to setting change commands.

2.2.5 Station Blocks

Station blocks are like regular blocks (see Figure 2.1) except that they allow a train to stop at a fixed specified location. Using a station block is the only way to be sure of the exact position of a train after it has begun moving, because track circuits only reveal whether a block was occupied sometime in the past.

Each station block B has a *station-stop-speed* VSS_B. A station stop begins whenever a train that is in station-stop mode (see Section 2.3) enters a station block while travelling with speed at most VSS_B. Under these conditions, the train will automatically begin to slow down so that it will come to a complete stop exactly at the opposite terminator of the station block.

A station stop by a train can be aborted by commanding that train to change velocity or acceleration.

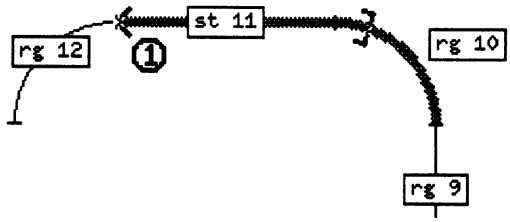


Figure 2.6: Block occupancy.

2.3 Trains

Every train is assigned a unique *train ID* T and a *length* and has the following attributes that may change during a simulation.

- A *speed* v_T and an *acceleration* a_T.
- Two *ends*, labelled *head* and *tail*, that are points within blocks.
- A *direction* dir_T.
- An *emergency-stop mode* es_T and a *station-stop mode* ss_T, which may be enabled or disabled.

We expect $0 \leq v_T \leq MAXFLOAT$. The distance along the track between a train's ends is always that train's length L_T.

Each train in a layout always occupies a sequence of attached blocks. A train *occupies a block* if the interiors of that train and that block intersect. For example, suppose that the train in Figure 2.6 has just completed a station stop, so that one end of that train is at a terminator. That train occupies blocks `rg 10` and `st 11`, but does not occupy block `rg 12`. If that train moves slightly to the left (i.e., forward), it will then occupy block `rg 12`.

A train's direction dir_T determines which of the two ends advances if that train moves forward. The train end that advances is called the *front*, and the other end is called the *rear*. For example, suppose that the head of a train is the front. If that train's direction is changed, then the train's tail will become the front and the head will become the rear. A train's direction can only change when that train is not moving, that is, when $v_T = a_T = 0$.

2.3.1 Train States

A train can be in one of three states: *operational*, *derailed* or *collided*. Figure 2.7 shows possible transitions between these states.

A train's state changes from operational to derailed under any of the following conditions:

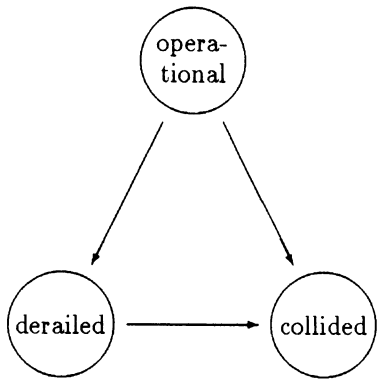


Figure 2.7: Train state transitions.

- That train occupies a block and the train's speed v_T violates one of that block's speed limits, $v_T < MN_B$ or $MX_B < v_T$.
- The front of that train exits a block at an unattached terminator.
- The front of that train enters a join block at the tail terminator or exits a join block from a head terminator.
- That train occupies a switch block that has undefined setting.
- A switch change is attempted on a switch block occupied by that train.
- That train enters the turn head terminator of a switch in the straight setting or the straight head terminator of a switch in the turned setting.

A train's state changes to collided if there is another train in the layout such that either of the following conditions holds.

- Both trains occupy the same switch, join or cross block.
- Both trains occupy the same block at the same point.

2.3.2 Laws of Motion

In **Trainset**, the acceleration of a train is one of the constant values ACC, 0, $-ACC$ and $-A_{emer}$, except during a station stop. We expect

$$0 < ACC < A_{emer} \leq MAXFLOAT.$$

If the acceleration a_T of a train is constant during a time interval $[t_0, t_1]$ and $v(t_i)$ is the speed v_T of that train at time t_i, then

$$v(t_1) = v(t_0) + a_T \cdot (t_1 - t_0).$$

For such a train, if $x(t_i)$ represents the position of an end of that train at time t_i, then

$$x(t_1) = x(t_0) + v(t_0) \cdot (t_1 - t_0) + \frac{a_T}{2} \cdot (t_1 - t_0)^2.$$

A train's station-stop mode turns on the station-stop feature for that train. A station stop begins if the station-stop mode of a train is enabled and that train enters a station block with speed v_T satisfying

$$0 \leq v_T \leq VSS_B,$$

where VSS_B is the station-stop speed of that block. The station-stop mode of that train is disabled as soon as a station stop begins.

A station stop by a train can be aborted by setting that train's speed or acceleration during that station stop. If a train performs a station stop in a station block and that station stop is not aborted, then all of the following will hold upon its completion.

- The train's speed v_T and acceleration a_T will be 0.
- The train's front will be at the opposite terminator of that station block.
- The train will occupy that station block.

A station stop's completion may be observed using the query **GetTrainMotion** discussed in Section 3.4.

The emergency-stop feature allows a train to stop at a faster rate than usual. A train performs an emergency stop when its emergency-stop mode is enabled. During an emergency stop by a train T, its acceleration satisfies $a_T = -A_{emer}$. The emergency stop finishes as soon as v_T becomes 0. At that time, the emergency-stop mode for T is disabled and its acceleration a_T becomes 0.

A train does not respond to commands during an emergency stop. Thus, emergency stops cannot be aborted.

Chapter 3

Automatic Control Interface (ACI)

3.1 Introduction

The Control Program Interface (CPI) is used for writing computer programs that interact with **Trainset** railroads. It consists of two layers, the Automated Control Interface (ACI) and the Low-Level Interface (LLI), as illustrated in Figure 3.1.

The ACI layer consists of library routines that can be used in a control program to establish communication with a simulation of a **Trainset** railroad, obtain the state of that railroad, interact with the railroad's layout using commands and queries and use the **Trainset** voting service.[1] The ACI also provides some local timer utilities. This layer is intended to meet the needs of most control program writers.

[1]The voting service facilitates writing fault-tolerant control programs. See Section 3.5.

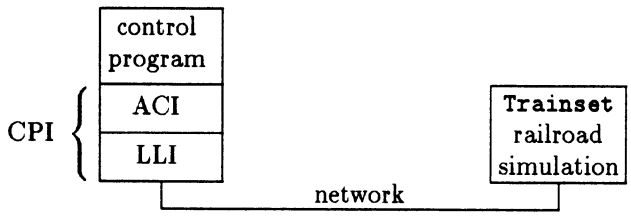


Figure 3.1: Communication interface layers.

The LLI layer has most of the same capabilities as the ACI, but at a more primitive level that requires a programmer to manage communication and message buffers directly.

This chapter describes the ACI layer. Usage details are described in a reference format using C language syntax. Chapter 4 presents the LLI layer. For programming examples that use the ACI layer, see Section 1.4 and Appendix B.2.

A **Trainset** railroad can accept three kinds of messages from CPI clients. A *state download request* asks for a railroad's state. A *command* can affect a railroad layout, e.g., by changing a switch block's setting or setting a train's acceleration. A *query* requests information about a block or train.

Trainset railroads silently ignore messages that contain syntax errors.

3.2 Initial-State Download

```
LayoutData *
GetDownload(hostname, simnum, progname, timeout)
     char *hostname;  /* machine running a simulation */
     int simnum;  /* identifies a running simulation */
     char *progname;  /* name of invoking program */
     double timeout;  /* in seconds */
```

Executing **GetDownload** causes a network connection to be established with a railroad simulation and information about the state of the railroad to be received from that simulation. The download is stored in an internal static data structure of type **LayoutData** defined in Appendix B.1.

hostname and **simnum**[2] must identify an executing **Trainset** railroad simulation, and **timeout** must be positive, or **GetDownload** will fail. If **hostname** is an empty string, then the local host is used.

The address of the **LayoutData** structure is returned if execution succeeds. A message is printed and a NULL pointer is returned on failure. Possible failure conditions are:

- **timeout** is not a positive floating-point value.
- A network connection could not be established before **timeout** seconds elapsed.
- There is no simulation on the host **hostname** with identifier **simnum**.

[2] **simnum** determines which well-known ports are to be used when initializing communication with a railroad simulation. Multiple railroad simulations may be run simultaneously on the same host if they use different **simnum** values.

- An error was detected while parsing the download.
- `GetDownload` has already been called successfully by this client at a prior time.

The download consists of the following.

General parameters:

- The number of blocks N_{blocks} and of trains N_{trains} in the layout. Block IDs range from 1 to N_{blocks} and train IDs range from 1 to N_{trains}.
- The standard acceleration rate ACC and emergency-stop acceleration rate A_{emer} for trains.
- The switching time T_{switch} for switch blocks.
- The voting window W_{voting} and voting quorum Q_{voting} for commands to be accepted. See Section 3.5.

For each block:

- The ID B, type and length of the block.
- The maximum speed limit MX_B and minimum speed limit MN_B.
- The attachments for that block. A block ID is specified for each attached terminator and 0 for each unattached terminator.
- For station blocks, the station-stop speed VSS_B.
- For subblocks of cross blocks, the block ID of the cross block.

For each train:

- The ID T and length of that train.
- The locations of that train's head and tail ends. The location of an end of a train is specified by giving the ID of a block containing that end and the offset of that end within the block. Offsets are measured from the tail terminator of a block.

3.3 Commands

A CPI client can cause a change in a `Trainset` railroad by issuing a command. Such a change is called a *layout action*. Only trains and switch blocks can be affected by commands.

Each command gives rise to at most one layout action. A command fails to cause a layout action if the preconditions of that command are not satisfied. For example, a **SetDirection** command causes a new direction to be assigned to a train only if that train is operational and not moving. Also, multiple commands may optionally be required to cause a single layout action, as described in Section 3.5.

There are seven types of command that a CPI client can send to a **Trainset** railroad: **SetSwitch**, **Accelerate**, **Decelerate**, **SetSpeed**, **SetDirection**, **EmergencyStop** and **StationStop**. These are specified as follows.

3.3.1 Set Switch

```
enum NewPosit {NP_STRAIGHT, NP_TURNED};

void
SetSwitch(block_id, new_setting)
     int block_id;
     enum NewPosit new_setting;
```

SetSwitch causes the switch block identified by **block_id** to begin changing setting to **new_setting**. A setting change requires T_{switch} seconds to complete.

block_id must identify an unoccupied switch block that is neither in the setting **new_setting** nor in the process of changing to that setting; otherwise no action is taken.

3.3.2 Accelerate and Decelerate

```
void
Accelerate(train_id, duration)
     int train_id;
     double duration;


void
Decelerate(train_id, duration)
     int train_id;
     double duration;
```

Accelerate and **Decelerate** cause the acceleration $a_{\texttt{train\_id}}$ of the train identified by **train_id** to be assigned the following value.

$$a_{\texttt{train\_id}} = \begin{cases} +ACC & \text{for } \textbf{Accelerate} \\ -ACC & \text{for } \textbf{Decelerate} \end{cases}$$

$a_{\texttt{train\_id}}$ is assigned the value 0 after either **duration** seconds elapse or the train's speed reaches 0, whichever comes first, unless $a_{\texttt{train\_id}}$ is changed again or the train collides or derails.

duration must be non-negative, and **train_id** must identify an operational train that is not performing an emergency stop; otherwise no action is taken.

3.3.3 Set Speed

```
void
SetSpeed(train_id, goal_speed)
     int train_id;
     double goal_speed;
```

SetSpeed causes the acceleration $a_{\texttt{train\_id}}$ of the train identified by **train_id** to be assigned the following value.

$$a_{\texttt{train\_id}} = \begin{cases} +ACC & \text{if } \texttt{goal\_speed} > v_{\texttt{train\_id}} \\ 0 & \text{if } \texttt{goal\_speed} = v_{\texttt{train\_id}} \\ -ACC & \text{if } \texttt{goal\_speed} < v_{\texttt{train\_id}} \end{cases}$$

The acceleration $a_{\texttt{train\_id}}$ is assigned the value 0 when that train's speed reaches **goal_speed**, unless $a_{\texttt{train\_id}}$ is changed again or the train derails or collides before the goal speed can be reached.

goal_speed must be non-negative, and **train_id** must identify an operational train that is not performing an emergency stop; otherwise no action is taken.

3.3.4 Set Direction

```
void
SetDirection(train_id, new_dir)
     int train_id;
     int new_dir;
```

SetDirection causes the direction $dir_{\texttt{train\_id}}$ train identified by **train_id** to be assigned the following value.

$$dir_{\texttt{train\_id}} = \begin{cases} \textit{tail-to-head} & \text{if } \texttt{new\_dir} > 0 \\ \textit{head-to-tail} & \text{if } \texttt{new\_dir} < 0 \\ \text{opposite of } dir_{\texttt{train\_id}} & \text{if } \texttt{new\_dir} = 0 \end{cases}$$

Here, *tail-to-head* means that the head end of that train is the front end, and *head-to-tail* means that the tail end is the front end.

The direction $dir_{\texttt{train\_id}}$ of **train_id** must identify an operational train that satisfies

$$a_{\texttt{train\_id}} = v_{\texttt{train\_id}} = 0$$

or no action is taken.

3.3.5 Emergency Stop

```
void
EmergencyStop(train_id)
     int train_id;
```

EmergencyStop causes the emergency-stop mode $es_{\texttt{train\_id}}$ of the train identified by **train_id** to be enabled. This causes an emergency stop to begin immediately. That train's acceleration $a_{\texttt{train\_id}}$ is assigned the value $-A_{emer}$ until the train's speed $v_{\texttt{train\_id}}$ becomes 0, at which time $a_{\texttt{train\_id}}$ is assigned the value 0 and $es_{\texttt{train\_id}}$ is disabled.

train_id must identify an operational train, or no action is taken.

3.3.6 Station Stop

```
enum StationStopMode {SS_DISABLED, SS_ENABLED};

void
StationStop(train_id, new_val)
     int train_id;
     enum StationStopMode new_val;
```

StationStop causes the station-stop mode $ss_{\texttt{train\_id}}$ of the train identified by **train_id** to be assigned **new_val**. A station stop (see Section 2.3.2) begins if an operational train enters a station block with station-stop mode enabled and with speed $v_{\texttt{train\_id}}$ that does not exceed the block's station-stop speed VSS_B.

train_id must identify an operational train that is not performing an emergency stop; otherwise no action is taken.

3.4 Queries

Queries are the only way that a CPI client can obtain information about a layout after the initial-state download has been received. There are four query types: **GetBlockOccupancy**, **GetSwitchPosition**, **GetTrainStatus** and **GetTrainMotion**. These queries return very limited information. Thus, facts

about the layout's state such as a train's speed and emergency-stop mode must be computed by a CPI client program from a knowledge of past values and of the properties of **Trainset** railroads. This makes writing process control programs difficult but realistic.

3.4.1 Block Occupancy

```
enum Occupancy {OC_FREE, OC_OCCUPIED, OC_ERROR = -1};

enum Occupancy
GetBlockOccupancy(block_id)
     int block_id;
```

The occupancy status of the indicated block is returned. If **block_id** is not the ID of a block or if a timeout occurs awaiting a reply from the simulation, **OC_ERROR** is returned.

3.4.2 Switch Position

```
enum SwitchPosit {SP_STRAIGHT, SP_TURNED, SP_UNDEFINED,
                  SP_ERROR = -1};

enum SwitchPosit
GetSwitchPosition(block_id)
     int block_id;
```

The setting of the indicated switch block is returned. If **block_id** is not the ID of a switch block or if a timeout occurs awaiting a reply from the simulation, **SP_ERROR** is returned.

3.4.3 Train Status

```
enum TrainStatus {TS_CRASHED, TS_RUNNING, TS_ERROR = -1};

enum TrainStatus
GetTrainStatus(train_id)
     int train_id;
```

If the indicated train has state operational, **TS_RUNNING** is returned. If that train's state is derailed or collided, **TS_CRASHED** is returned. If **train_id** is not the ID of a train or if a timeout occurs awaiting a reply from the simulation, **TS_ERROR** is returned.

3.4.4 Train Motion

```
enum TrainMotion {TM_STOPPED, TM_MOVING, TM_ERROR = -1};

enum TrainMotion
GetTrainMotion(train_id)
     int train_id;
```

If the indicated train has state operational and at least one of the train's speed $v_{\texttt{train\_id}}$ or acceleration $a_{\texttt{train\_id}}$ is non-zero, **TM_MOVING** is returned. Otherwise, **TM_STOPPED** is returned, except that **TM_ERROR** is returned if **train_id** is not the ID of a train or if a timeout occurs awaiting a reply from the simulation.

3.5 Voting

The *voting service* supports the writing of fault-tolerant control programs for a **Trainset** railroad. By default, there is a one-to-one relationship between CPI client commands (satisfying various preconditions, as described in Section 3.3) and layout actions. The voting service makes it possible to require that a simulation receive a specified number Q_{voting} of commands from different CPI clients within a certain time period W_{voting} before a layout action can occur.

The behavior of the voting service is governed by the following parameters.

- A *voting quorum* Q_{voting} (integer). When the voting service is being employed, a layout action can be generated only if Q_{voting} commands satisfying the conditions below are received from different CPI clients.
- A *voting window* W_{voting} (floating point, seconds). This is a time limit after which CPI client commands expire. If a command has not contributed to a layout action within W_{voting} seconds after receipt by a simulation, then that command will have no effect.
- A *sequence number seq* (integer) that is associated with each CPI command.

We expect $Q_{voting} > 0$, $W_{voting} \geq 0$ and $seq \geq 0$.

The voting service is enabled only for commands with positive sequence numbers. Figuratively speaking, commands with different positive sequence numbers participate in different "elections."

The following paragraphs provide a specification of the voting service.

Sequence number 0. If a command has sequence number 0, then that command alone generates a layout action immediately, as described in Section 3.3. The values of W_{voting} and Q_{voting} have no effect on such commands.

	time	command type	client	sequence number
Command A	7:22:06	`Accel`	2	61
B	7:22:13	`Accel`	1	60
C	7:22:17	`Accel`	3	61
D	7:22:19	`Accel`	1	60
E	7:22:25	`SetSpeed`	2	61
F	7:22:30	`SetSpeed`	3	61

Figure 3.2: Voting example (positive sequence numbers).

Positive sequence numbers. If a simulation receives Q_{voting} or more commands with the same positive sequence number *seq* from different CPI clients during any time period of W_{voting} seconds or less, then a single layout action is generated. By default, the layout action generated is the action caused (subject to the preconditions listed in Section 3.3) by the last of those commands.

No commands with positive sequence numbers can otherwise contribute to layout actions. Commands with the same sequence number need not have the same arguments nor even the same command type. A command can contribute to the generation of at most one layout action.

For example, suppose that $W_{voting} = 10$ sec, $Q_{voting} = 2$ and that the commands in Figure 3.2 are received by a simulation. Then only one layout action will be performed by that simulation as of the time 7:22:30. That layout action will be E, occurring at time 7:22:25. The quorum of commands consists of C and E.

The values of Q_{voting} and W_{voting} are determined when a `Trainset` railroad simulation is invoked, and cannot be changed during a simulation. The default value for Q_{voting} is 2, and the default value for W_{voting} is 5 seconds.

Sequence numbers can be modified dynamically by a control program. The ACI provides an internal sequence-number variable seq_{ACI} whose value is automatically sent with each command. The value of the variable seq_{ACI} remains constant unless explicitly modified, and the initial value is 0.

In order to employ the voting service, it is necessary to start a simulation with $Q_{voting} > 1$ and to send commands that have positive sequence numbers. The ACI provides two functions for modifying the value of a sequence number.

3.5.1 New Sequence Number

```
int
NewSeqNumber()
```

The sequence number is incremented and the new value is returned.

3.5.2 Set Sequence Number

```
int
SetSeqNumber(new_val)
     int new_val;
```

If **new_val** is non-negative, the sequence number *seq* is assigned **new_val**, and that value is returned. Otherwise, *seq* is left unchanged and -1 is returned.

3.6 Timer Facilities

The ACI layer provides procedures that implement a simple timer service in which time quantities are represented as seconds in floating point. These routines affect a program that calls them; they do not themselves interact with a **Trainset** railroad. Alternative time-management routines or direct system calls may be used instead of these timer procedures when writing programs that use the ACI. Thus, use of the ACI timer facilities is optional.

The five timer procedures are **InitTimer**, **GetTimer**, **AwaitTimer**, **CancelTimer** and **Sleep**.

3.6.1 Initialize Timer

```
enum TimerReturn {TMR_SUCCESS, TMR_ERROR = -1};
typedef double Seconds;

enum TimerReturn
InitTimer(first, period, f)
     Seconds first; /* delay to first timeout */
     Seconds period;  /* wait between successive timeouts */
     void (*f)();  /* function executed upon each timeout */
```

InitTimer initializes the timer mechanism to execute the function **f()** after **first** seconds and every **period** seconds thereafter. If **first** is zero, the timer is immediately disabled. If **period** is zero, the timer executes **f()** at most once.

If the value zero is specified instead of a function **f**, then nothing is done on timeouts except to reset the timer as appropriate. **TMR_SUCCESS** is returned if the operation succeeds. If a time argument is negative or a low-level error occurs, **TMR_ERROR** is returned.

A call to **InitTimer** overrides any prior timer setting. Thus, there is no provision for more than one timer being in effect at any time. The timers used in other ACI functions, including **GetDownload**, the ACI queries and **Sleep**, do not interfere with **InitTimer**.

3.6.2 Get Timer Values

```
enum TimerStatus {TMS_DISABLED, TMS_ONCE, TMS_INDEFINITE,
          TMS_ERROR = -1};
typedef double Seconds;

enum TimerStatus
GetTimer(nextp, periodp)
     Seconds *nextp;  /* next scheduled timeout */
     Seconds *periodp;  /* delay between successive timeouts */
```

GetTimer stores the seconds remaining until the next timer expiration in ***nextp** and stores the current period between timer expirations in ***periodp**. The return value is **TMS_DISABLED** if the timer is currently disabled, **TMS_ONCE** if the timer is scheduled to expire once only, **TMS_INDEFINITE** if the timer is scheduled to continue indefinitely and **TMS_ERROR** if a low-level error occurs.

3.6.3 Await for Timer

```
void
AwaitTimer()
```

AwaitTimer suspends the calling process until an ACI timer expiration or another operating system signal occurs.

3.6.4 Cancel Timer

```
enum TimerReturn {TMR_SUCCESS, TMR_ERROR = -1};

enum TimerReturn
CancelTimer()
```

CancelTimer cancels an ACI timer. **TMR_SUCCESS** is returned if the operation succeeds. **CancelTimer** fails if a low-level error occurs, in which case **TMR_ERROR** is returned.

3.6.5 Sleep

```
enum TimerReturn {TMR_SUCCESS, TMR_ERROR = -1};
typedef double Seconds;

enum TimerReturn
Sleep(sec)
     Seconds sec;  /* max number of seconds to suspend */
```

Sleep suspends the calling process for indicated number of seconds. The value **TMR_SUCCESS** is returned if the suspension was not interrupted before the time limit expired. Otherwise, **TMR_ERROR** is returned.

Sleep does not interfere with the operation of **InitTimer**. Thus, these functions may be used together, e.g., to set up a polling loop with a timeout.

Chapter 4

Low-Level Interface (LLI)

4.1 Introduction

The Low-Level Interface (LLI) consists of predefined message formats and related facilities for communicating with a **Trainset** railroad simulation. This layer is used to implement the ACI layer in the Control Program Interface (see Figure 3.1).

This chapter gives a sketch of the LLI layer. A reading knowledge of the C programming language is assumed. The rest of this chapter includes the following sections.

- Section 4.2 is external documentation for the source file `cpi.h` (see Appendix C.1). The internal structure of that file is described, and the source lines that pertain to the acceleration command are discussed as examples.
- Section 4.3 describes the steps necessary for a client to connect and identify itself to a **Trainset** railroad simulation.
- Sections 4.4 and 4.5 briefly list the message formats that are used for initial-state downloads, commands and queries. A knowledge of the ACI layer (Chapter 3) is assumed.

For a programming example that uses the LLI programming layer, see the implementation of the ACI layer in the source file `aci.c` (Appendix C.2).

4.2 Messages

In the LLI layer, a control program interacts with a **Trainset** railroad simulation by sending and receiving messages. An *LLI message* is an ASCII character string that is organized according to one of the *message formats* defined in

```
1
#define MSG_CPI_ACCEL "Xa%d,%d:%d:%lf#"
typedef struct {
2   int client_id;  /* unique identifier for this client */
    int seq_num;  /* label for this command */
    int train_id;  /* simulator train number */
    double duration;  /* time to accelerate in seconds */
} MsgCPIAccel;

#define MSG_CPI_ACCEL_ARGC 4
#define MsgCPIAccelArgs(buff, datap) (buff, MSG_CPI_ACCEL, \
  datap->client_id , datap->seq_num , datap->train_id , datap->duration )
3
#define CreateMsgCPIAccel(buff, datap) sprintf MsgCPIAccelArgs(buff, (datap))
4   #define ParseMsgCPIAccel(buff, datap) sscanf MsgCPIAccelArgs(buff, &(datap))
```

Figure 4.1: Excerpt from `cpi.h` relating to acceleration command.

`cpi.h`. Each such message format consists of *fields* for representing data, together with an identifying header and delimiter characters.

For example, consider Figure 4.1, an excerpt from `cpi.h`. Line 1 defines a message format called **MSG_CPI_ACCEL**. That message format has the following syntax.

Xa ⟨*integer*⟩ , ⟨*integer*⟩ : ⟨*integer*⟩ : ⟨*float*⟩ **#**

There are four fields. The notations ⟨*integer*⟩ and ⟨*float*⟩ indicate ASCII representations of integer and floating point numbers. The total length of a message depends on the lengths of the data in its fields.

In general, for each message format in `cpi.h` there is a type definition of an *associated data structure* whose components correspond to the fields of that message format, in the same order.[1] For **MSG_CPI_ACCEL**, the associated data structure is called **MsgCPIAccel**. See the lines near 2 in Figure 4.1.

A comparison of the message format **MSG_CPI_ACCEL** and the documented data structure **MsgCPIAccel** shows that the first ⟨*integer*⟩ field in that message format holds the client ID,[2] the second ⟨*integer*⟩ field holds the voting sequence number of an **Accelerate** command, etc.

For each message format, a macro has been defined that produces a string in that message format, using values stored in a variable of the associated data

[1] Four CPI message formats are associated with basic data types that are not explicitly defined as structures in `cpi.h`. These are **MSG_CPI_INIT** (string), **MSG_ACK** (string), **MSG_CPI_DOWNLD_DONE** (integer) and **MSG_QUIT_ALL** (string).

[2] A unique client ID is assigned to each CPI client when that client connects to a simulation, as described below.

type for the fields. Another macro has been defined for parsing such a message into a variable the associated data type. These macros are provided as a convenience. In the case of `MSG_CPI_ACCEL`, the creation macro is called `CreateMsgCPIAccel` and is defined by line [3] in Figure 4.1. The message-parsing macro is `ParseMsgCPIAccel` defined on line [4].

In general, for any such macro associated with a message format:

- There are two arguments, a character string buffer holding the message and a pointer to a variable of the associated data structure type holding the values.
- The run-time replacement value of such a macro is the length of the resulting message, not counting a null byte that is placed at the end of that message string.[3]

Thus, the macro `CreateMsgCPIAccel` behaves like a function with the following heading.

```
int
CreateMsgCPIAccel(buff, datap)
     char buff[];  /* resulting message is placed here */
     MsgCPIAccel *datap;  /* points to data values */
```

4.3 Initiating Communication

Establishing communication between a client and a `Trainset` railroad simulation requires two steps.

1. **Connection.** Create a network connection between that client and that simulation. A simulation provides well-known ports for supported communication protocol families (e.g., TCP/IP, DECnet) and is capable of socket or stream transmissions in each case. The function `GetSimPort` defined in the source file `lib/syslocal.c` returns the well-known port, given a base port (provided in `lib/syslocal.h`), a host name and an integer *simnum* that distinguishes between multiple simulations running on the same host. Use standard techniques to open one bidirectional communication line.
2. **Registration.** Send a registration message to that simulation to declare a client type (CPI or monitor) and receive a client ID. A registration message for a CPI client has the message format `MSG_CPI_INIT`

   `CPI` ⟨*one space*⟩ ⟨*string*⟩ `%`

[3] Sending a terminating null byte to a simulation as part of a message is optional.

where ⟨*string*⟩ is any sequence of characters that are not the percent sign % and is conventionally the file name of that client's executable. The simulation replies with an acknowledgement message in the format **MSG_ACK**, as follows.

ack ⟨*one space*⟩ ⟨*integer*⟩ **# %**

The ⟨*integer*⟩ field in that message represents the unique client ID number to be used in all commands and queries from that client to the simulation.

The macros **CreateMsgCPIInit** and **ParseMsgAck** may be used if desired.

We assume that each client creates only one connection with a simulation.

For the remainder of this chapter, message formats will be cited by name only, omitting mention of syntax, related macros, etc. See Appendix C.1 for syntax details.

4.4 Initial-State Download

After initiating communication with a simulation, a CPI client can receive a report of the current global state of the railroad being simulated. Such a report is called a *download*.

To request a download, a control program must send a message to a simulation using the message format **MSG_CPI_DOWNLD_REQUEST**. The acknowledgement from the simulation is the download itself. The download consists of messages in the following **DOWNLD** formats.

- **MSG_CPI_DOWNLD_GENERAL** transmits constants that describe a railroad in general, e.g., the number of blocks.
- **MSG_CPI_DOWNLD_TRAIN** describes attributes of trains. One message in this format is sent per train.
- **MSG_CPI_DOWNLD_BLOCK** describes a block. **MSG_CPI_DOWNLD_LINK** gives block attachments for a block as described in Section 3.2. One message in each of these formats is sent per block.
- **MSG_CPI_DOWNLD_DONE** marks the end of a download.

Only one download request is honored by a simulation per client connection.

4.5 Commands and Queries

- **MSG_CPI_SET_SWITCH**, **MSG_CPI_ACCEL**, **MSG_CPI_DECEL**, **MSG_CPI_SET_SPEED**, **MSG_CPI_SET_DIR**, **MSG_CPI_EMER_STOP** and **MSG_CPI_STA_STOP** request the commands described in Section 3.3. No acknowledgement message is sent in response to such a command.

- **MSG_CPI_BLOCK_OCC** requests a block's occupancy status. A simulation responds with a message[4] "**o**" if that block is occupied or "**f**" (for "free") if it is not occupied.
- **MSG_CPI_SWITCH_POSIT** asks for the current setting of a switch block. The response is "**s**" for straight, "**t**" for turned, or "**u**" for undefined.
- **MSG_CPI_TRAIN_STATUS** inquires about a train's status. A simulation responds with "**r**" for running (operational state) or "**c**" for crashed (collided or derailed states).
- **MSG_CPI_TRAIN_MOTION** queries whether a train is moving. The response is "**s**" for stopped or "**m**" for moving. A train is moving if it is operational and at least one of its speed and acceleration is non-zero. A train is stopped if it is not moving.

4.6 Quit Message From the Simulator

Just before a **Trainset** simulation exits normally, it sends a message in the format **MSG_QUIT_ALL** to all of its clients. Receipt of this message is a certain indicator that a simulation has finished.

[4]Note: One-character return strings such as "o" are sent as two-byte messages from the simulator, including the terminating null byte.

Appendix A

Constants Used by Trainset

Constant name	Description	Typical value	Page references[1]
ACC	Standard acceleration rate for trains	1.5 m/sec^2	**36**, 40, 41, 42
A_{emer}	Emergency-stop acceleration rate for trains	2.5 m/sec^2	36, **37**, 40, 43
LEN_{cross}	Length of a cross block	35 m	**33**
LEN_{join}	Length of a join block	35 m	**32**
LEN_{switch}	Length of a switch block	35 m	**34**
L_B	Length of block B	35 to 300 m	**30**
L_T	Length of train T	150 to 500 m	**35**
MN_B	Minimum speed limit on block B	0 m/sec	**30**, 36, 40
MX_B	Maximum speed limit on block B	60 m/sec	**30**, 36, 40
N_{blocks}	Number of blocks in a layout	5 to 50	**40**
N_{trains}	Number of trains in a layout	1 to 3	**40**
Q_{voting}	Voting quorum	2	40, **45**, 46
T_{switch}	Time for a switch block to change settings	2 sec	**33**, 40, 41
VSS_B	Station stop speed for a station block	30 m/sec	**34**, 37, 40, 43
W_{voting}	Voting window	5 sec	40, **45**, 46

[1] A **boldface** number indicates the primary reference for a symbol.

Appendix B

Code Listings: ACI

B.1 aci.h, Interface Header File

```
/* aci.h -- header file for ACI.
   This file defines types and constants and declares functions
   found in the ACI library libaci.a .
   Created by R. Brown, 4/91 */

/* all units are mks unless otherwise indicated */


/*
 * constant definitions
 */


#ifndef MAX_TRAINS
#define MAX_TRAINS 10    /* maximum number of trains in a layout */
#endif
#ifndef MAX_BLOCKS
#define MAX_BLOCKS 100   /* maximum number of blocks in a layout */
#endif


/*
 * type definitions for storing a state download
 */
```

```
/* BlockType specifies the type of a block */

enum BlockType {BT_REGULAR, BT_STATION, BT_SWITCH, BT_JOIN, BT_CROSS};


/* BlockTypeData represents data that is specific to a type of block */

union BlockTypeData {

  struct {  /* BT_REGULAR blocks */
    int tail, head;  /* terminators */
  } rg;

  struct {  /* BT_STATION blocks */
    int tail, head;  /* terminators */
    double sta_stop_speed;  /* station stop speed */
  } st;

  struct {  /* BT_SWITCH blocks */
    int tail, straight, turn;  /* terminators */
  } sw;

  struct {  /* BT_JOIN blocks */
    int tail, head1, head2;  /* terminators */
  } jn;

  struct {  /* BT_CROSS (sub)blocks */
    int tail, head;  /* terminators */
    int cross_id;  /* block ID of this subblock's cross block */
  } cr;
};


/* BlockData represents the download information for a specific block */

typedef struct BlockData {
  int block_id;  /* identifies the block */
  enum BlockType type;
  double length;
  double max_speed, min_speed;  /* trains derail that violate these limits */
  union BlockTypeData t;  /* additional data that depends on block type */
} BlockData;
```

```
/* Location specifies a position within a block */

struct Location {
  int block;  /* id of a block containing the location */
  double offset;  /* distance from tail of that block to the location */
} Location;


/* TrainData represents the download information for a specific train */

typedef struct TrainData {
  int train_id;  /* identifies the train */
  double length;
  struct Location head, tail;  /* init positions of train's ends */
} TrainData;


/* LayoutData represents all information received in a download */

typedef struct LayoutData {

  int block_ct;  /* number of blocks */
  int train_ct;  /* number of trains */
  double acc, emer_acc;  /* standard and emergency stop acceleration rates*/
  double switch_time;  /* time required for switch block to change posit */
  int voting_quorum;  /* number of agreeing commands required for an action */
  double voting_window;  /* time limit after which commands expire */

  BlockData blocks[MAX_BLOCKS];  /* block-specific data */
  TrainData trains[MAX_TRAINS];  /* train-specific data */

} LayoutData;


/*
 * ACI function declarations, with associated type definitions
 */


/* GetDownload, for connecting to a simulated railroad and receiving
   a report of that railroad's state.  */

LayoutData    *GetDownload();
```

```
/* ACI commands */

enum NewPosit {NP_STRAIGHT, NP_TURNED};
void SetSwitch();

void Accelerate();

void Decelerate();

void SetSpeed();

void SetDirection();

void EmergencyStop();

enum StationStopMode {SS_DISABLED, SS_ENABLED};
void StationStop();


/* ACI queries */

enum Occupancy {OC_FREE, OC_OCCUPIED, OC_ERROR = -1};
enum Occupancy GetBlockOccupancy();

enum SwitchPosit {SP_STRAIGHT, SP_TURNED, SP_UNDEFINED, SP_ERROR = -1};
enum SwitchPosit GetSwitchPosition();

enum TrainStatus {TS_CRASHED, TS_RUNNING, TS_ERROR = -1};
enum TrainStatus GetTrainStatus();

enum TrainMotion {TM_STOPPED, TM_MOVING, TM_ERROR = -1};
enum TrainMotion GetTrainMotion();


/* changes to voting service sequence number */

int SetSeqNumber();

int NewSeqNumber();


/* timer facility */
```

```
typedef double Seconds;
enum TimerReturn {TMR_SUCCESS, TMR_ERROR = -1};

enum TimerReturn InitTimer();

enum TimerStatus {TMS_DISABLED, TMS_ONCE, TMS_INDEFINITE, TMS_ERROR = -1};
enum TimerStatus GetTimer();

void AwaitTimer();

enum TimerReturn CancelTimer();

enum TimerReturn Sleep();
```

B.2 acitest.c, Example Using the ACI Interface

```
/* $RCSfile: acitest.c,v $ $Revision: 1.8 $ $Date: 92/07/17 10:58:43 $ */
/* acitest.c -- manual demonstration of the ACI interface.  */

#include <stdio.h>
#include "aci.h"

#define TIMEOUT 60.0    /* seconds to contact the simulator */
#define MAXLINE 100   /* maximum input length */
#define NORMAL 0  /* exit code for normal exit */
#define ERROR 1  /* exit code for error exit */
#define USAGE "Usage:  %s [-d] [simhost [simnum]]\n"  /* diagnostic msg */
#define QUIT_COMMAND "q"   /* user entry for quit command */

enum command_type {
  /* commands */
  SET_SWITCH = 1, ACCELERATE, DECELERATE, SET_SPEED,
  SET_DIRECTION, EMERGENCY_STOP, STATION_STOP,
  /* queries */
  GET_BLOCK_OCCUPANCY, GET_SWITCH_POSITION, GET_TRAIN_STATUS, GET_TRAIN_MOTION,
  /* other */
  SET_SEQ_NUMBER, NEW_SEQ_NUMBER,
  QUIT, HELP
  };

struct command {
```

```
  enum command_type type;  /* type of this command or query */
  char *code;  /* entry code for this command or query */
  char *name;  /* full name of this ACI command or query */
  char *args;  /* names of arguments required for this command or query */
};

/* NULL-terminated table of command types, codes and names */

struct command command_table[] = {
  /* commands */
  SET_SWITCH, "sw", "SetSwitch",   "<block_id>  <new_posit>",
  ACCELERATE, "ac", "Accelerate",   "<train_id>  <duration>",
  DECELERATE, "dc", "Decelerate",   "<train_id>  <duration>",
  SET_SPEED, "sp", "SetSpeed",   "<train_id>  <goal_speed>",
  SET_DIRECTION, "sd", "SetDirection",  "<train_id>  <new_dir>",
  EMERGENCY_STOP, "es", "EmergencyStop",  "<train_id>",
  STATION_STOP, "st", "StationStop",  "<train_id>  <new_val>",
  /* queries */
  GET_BLOCK_OCCUPANCY, "o", "GetBlockOccupancy",  "<block_id>",
  GET_SWITCH_POSITION, "p", "GetSwitchPosition",  "<block_id>",
  GET_TRAIN_STATUS, "s", "GetTrainStatus",  "<train_id>",
  GET_TRAIN_MOTION, "m", "GetTrainMotion",  "<train_id>",
  /* other */
  SET_SEQ_NUMBER, "sn", "SetSeqNumber", "<new_val>",
  NEW_SEQ_NUMBER, "nn", "NewSeqNumber", "",
  QUIT, QUIT_COMMAND, "Quit", "",
  HELP, "?", "Help",  "  h    help",
  (enum command_type) 0, (char *) 0, (char *) 0, (char *) 0
  };


/*********************************************************************/

/* Whitespace takes a char argument c.  Evaluates to non-zero if
   c is whitespace, zero otherwise */

#define Whitespace(c) (c == ' ' || c == '\t' || c == '\n')


/*********************************************************************/

/* CommandLookup determines the command corresponding to a code
   entered to identify a command.
   Any whitespace characters at the beginning of entry are ignored.
```

```
   Returns the index in the table of the indicated command, or
   -1 if the code was empty or the given entry code was not found.  */

int
CommandLookup(entry)
     char *entry;  /* entry code to be interpreted as a command */
{
  struct command *p;  /* pointer into command table */

  while (*entry && Whitespace(*entry))
    entry++;
  if (!*entry)
    return (-1);
  /* entry is non-empty string beginning with a non-whitespace char */

  if (*entry == 'h')
    entry = "?";
  /* if request is for Help then entry has value "?" */

  for (p = command_table;  p->type;  p++) {
    if (!strcmp(entry, p->code)) {
      return (p - command_table);
    }
  }
  /* entry does not match a code in the table */

  printf("  no match found\n");
  return (-1);
}


/*********************************************************************/

/* PrintDownload displays every field received in a download */

void
PrintDownload(dp, simhost, simnum)
     LayoutData *dp;  /* data received from download */
     char *simhost;  /* name of the host system running the simulator */
     int simnum;  /* identifier for a running simulation on that host */
{
  int i;  /* loop control */
  TrainData *tp;  /* utility pointer to data for a specific train */
  BlockData *bp;  /* utility pointer to data for a specific block */
```

```
 printf("Contents of download from simhost %s, simnum %d:\n",
simhost, simnum);

 printf("\nGeneral parameters:\n");
 printf("  There are %d blocks and %d trains\n",
dp->block_ct, dp->train_ct);
 printf("  Train acceleration rates (m/sec):  standard %f, emergency stop %f\n",
dp->acc, dp->emer_acc);
 printf("  Switches take %f seconds to change position\n",
dp->switch_time);
 printf("  The voting quorum is %d and the voting window is %f seconds\n",
dp->voting_quorum, dp->voting_window);

 printf("\nBlocks:\n");
 for (i=0;  i < dp->block_ct;  i++) {
   bp = &dp->blocks[i];
   printf("  Block %d has type ", bp->block_id);
   switch (bp->type) {
   case BT_REGULAR:
     printf("REGULAR");
     break;
   case BT_STATION:
     printf("STATION");
     break;
   case BT_SWITCH:
     printf("SWITCH");
     break;
   case BT_JOIN:
     printf("JOIN");
     break;
   case BT_CROSS:
     printf("CROSS");
     break;
   }
   printf(" and length %f meters\n", bp->length);
   printf("    Its speed limits are %f (minimum) and %f (maximum)\n",
  bp->min_speed, bp->max_speed);
   printf("    Its terminators are connected to blocks ");
   switch (bp->type) {
   case BT_REGULAR:
     printf("%d (head) and %d (tail)\n",
    bp->t.rg.head, bp->t.rg.tail);
     break;
```

```
    case BT_STATION:
      printf("%d (head) and %d (tail)\n",
     bp->t.st.head, bp->t.st.tail);
      printf("    Its station stop speed is %f m/sec\n",
     bp->t.st.sta_stop_speed);
      break;
    case BT_SWITCH:
      printf("%d (straight head), %d (turn head) and %d (tail)\n",
     bp->t.sw.straight, bp->t.sw.turn, bp->t.sw.tail);
      break;
    case BT_JOIN:
      printf("%d (head1), %d (head2) and %d (tail)\n",
     bp->t.jn.head1, bp->t.jn.head2, bp->t.jn.tail);
      break;
    case BT_CROSS:
      printf("%d (head) and %d (tail)\n",
     bp->t.cr.head, bp->t.cr.tail);
      printf("    This is a subblock of the cross block with ID %d\n",
     bp->t.cr.cross_id);
      break;
    }
  }

  printf("\nTrains:\n");
  for (i=0;  i < dp->train_ct;  i++) {
    tp = &dp->trains[i];
    printf("  Train %d has length %f meters\n",
   tp->train_id, tp->length);
    printf("    Its head end is in block %d, %f meters from tail terminator\n",
   tp->head.block, tp->head.offset);
    printf("    Its tail end is in block %d, %f meters from tail terminator\n",
   tp->tail.block, tp->tail.offset);
  }

  printf("\nEnd of configuration download report\n\n");
  return;
}


/*****************************************************************/

/* GetValue attempts to locate a value in the string buff.
   If buff contains only whitespace, prompt is printed and standard input
   is read until some non-whitespace characters are entered.
```

```
   If a value matching the printf format fmt is found in buff or stdin,
   then its value is placed in *valp and a pointer is returned that points
   to the first whitespace character after the value.
   If the next non-whitespace characters do not scan according to fmt,
   an error message requesting user to try again is printed and
   NULL is returned. */

char *
GetValue(buff, fmt, valp, prompt)
     char *buff;  /* buffer that may contain a value */
     char *fmt;  /* printf-style format for reading the value */
     char *valp;  /* to receive value if one is found */
     char *prompt;  /* prompt, used to obtain standard input */
{
  char *p;  /* utility pointer */

  /* invar:  buff is unexamined */
  do {
    for (p = buff;  *p && Whitespace(*p);  p++)
      ;
    /* *p is null-byte or first non-whitespace */

    if (!*p) {
      printf(" %s:  ", prompt);
      gets(buff);
      p = buff;
    }
    /* *p is first non-whitespace char, or p points to beginning of
       an unexamined (yet possibly empty) buff */
  } while (!*p || Whitespace(*p));
  /* *p is non-whitespace */

  if (sscanf(p, fmt, valp) != 1) {
    printf("Error reading value -- try again\n");
    return (NULL);
  }
  /* value was successfully read into *valp */

  while (*p && !Whitespace(*p))
    p++;
  return (p);
}
```

```
/* GetInt and GetFloat are macros for getting values of certain types
   from buff[] or standard input, via GetValue above */

#define GetInt(buff, valp, prompt) GetValue(buff, "%d", valp, prompt)
#define GetFloat(buff, valp, prompt) GetValue(buff, "%lf", valp, prompt)



/*****************************************************************/

main(argc, argv)
     int argc;  /* number of command line arguments remaining */
     char *argv[];  /* list of command line arguments remaining */
{
  char *progname;  /* name of this program */
  char simhost[MAXLINE+1];  /* name of host system running simulator */
  int simnum = 0;  /* number of this simulator */
  int print_download = 0;  /* boolean;  set non-zero to print download */
  char buff[MAXLINE+1];  /* buffer for interactive input */
  LayoutData *datap;  /* data received from download */
  int index;  /* index of a command table entry */
  char *p, **tablep;  /* utility pointers */
  struct command *commandp;  /* pointer into command table */
  char prompt[MAXLINE];  /* holds any prompts for user input */

  p = progname = *argv++;  argc--;
  /* invar:  there are no slashes in the string progname before p */
  while (*p)
    if (*p++ == '/')
      progname = p;
  /* progname is null-terminated program name after stripping directories */

  if (argc > 0 && **argv == '-') {
    /* a flag was encountered */
    if (!strcmp(*argv, "-d")) {
      print_download++;
      argc--;  argv++;
    } else {
      fprintf(stderr, USAGE, progname);
      exit (ERROR);
    }
  }
  /* all flags have been processed */
```

```
if (argc > 0) {
  /* a next argument exists, presumed to be simhost */
  if (strlen(*argv) > MAXLINE) {
    fprintf(stderr, "simulator host name too long, max = %d\n", MAXLINE);
    exit (ERROR);
  }
  /* first arg will fit in simhost */

  strcpy(simhost, *argv++);
  argc--;

} else {
  /* no more command-line arguments remain */
  printf("Enter name of host running simulator [LOCAL HOST]: ");
  if (!gets(simhost)) {
    fprintf(stderr, "error getting host name\n");
    exit (ERROR);
  }
  printf("Enter simulator number on that host [0]: ");
  if (!gets(buff)) {
    fprintf(stderr, "error getting simulator number\n");
    exit (ERROR);
  }
  simnum = (*buff) ? atoi(buff) : 0;
}
/* simhost contains (possibly empty) null-terminated string  AND
   if no argument remains then simnum holds specified or default value */

if (argc > 0) {
  /* another argument  second parameter exists, presumed to be simnum */
  sscanf(*argv++, "%d", &simnum);
  argc--;
}
/* simnum holds specified or default value */


printf("Connecting to simulator using %.1f second timeout...\n", TIMEOUT);
if (!(datap = GetDownload(simhost, simnum, progname, TIMEOUT))) {
  fprintf(stderr, "GetDownload failed\n");
  exit (ERROR);
}
/* download was successfully received */

if (print_download)
```

```
    PrintDownload(datap, simhost, simnum);

  printf("Enter ? for a list of commands\n");

  for (;;) {
    do {
      printf("%s> ", progname);  /* prompt */

      buff[0] = '\0';
      if (!gets(buff))
/* end of input encountered */
sprintf(buff, QUIT_COMMAND);
      for (p=buff;  *p && Whitespace(*p);  p++)
;
      /* p points to first non-whitespace char of buff, or null byte */
    } while (!*p);
    /* buff[] contains a new non-empty null-terminated command */

    while (*p && !Whitespace(*p))
      p++;
    /* *p is null-byte or first whitespace after command code */

    if (*p)
      *p++ = '\0';
    /* buff is null-terminated string holding a non-empty command code
       consisting of non-whitespace chars, and holding QUIT_COMMAND on EOF,
       AND string p is any remainder of the (null-terminated) input line */

    if ((index = CommandLookup(buff)) == -1) {
      printf("Unrecognized command code `%s'  (enter `?' for help)\n",
     buff);
      continue;
    }
    /* a recognized command code was entered */

    /* gather arguments for the appropriate ACI routine and call it */

    switch (command_table[index].type) {
    case SET_SWITCH:
      {
int block_id;  /* identifier of the switch block */
enum NewPosit new_posit;  /* desired position */

p = GetInt(p, &block_id, "SetSwitch.  ID of switch block");
```

```
if (!p) continue;

sprintf(prompt, "New position (%d = straight, %d = turned)",
NP_STRAIGHT, NP_TURNED);
p = GetInt(p, &new_posit, prompt);
if (!p) continue;
/* all arguments successfully obtained */

SetSwitch(block_id, new_posit);
break;
      }

    case ACCELERATE:
    case DECELERATE:
      {
int train_id;  /* identifier of the train */
double duration;  /* number of seconds to accelerate */

sprintf(prompt, "%s.  ID of train", command_table[index].name);
p = GetInt(p, &train_id, prompt);
if (!p) continue;

sprintf(prompt, "Seconds to %s", command_table[index].name);
p = GetFloat(p, &duration, prompt);
if (!p) continue;
/* all arguments successfully obtained */

if (command_table[index].type == ACCELERATE)
  Accelerate(train_id, duration);
else
  Decelerate(train_id, duration);
break;
      }

    case SET_SPEED:
      {
int train_id;  /* identifier of the train */
double goal_speed;  /* speed to accelerate or decelerate to */

p = GetInt(p, &train_id, "SetSpeed.  ID of train");
if (!p) continue;

p = GetFloat(p, &goal_speed, "New goal speed");
if (!p) continue;
```

```
/* all arguments successfully obtained */

SetSpeed(train_id, goal_speed);
break;
      }

    case SET_DIRECTION:
      {
int train_id;  /* identifier of the train */
int new_dir;  /* indicator of new direction */

p = GetInt(p, &train_id, "SetDirection.  ID of train");
if (!p) continue;

p = GetInt(p, &new_dir,
 "New direction (+1 for front=head, -1 for front=tail, 0 to toggle)");
if (!p) continue;
/* all arguments successfully obtained */

SetDirection(train_id, new_dir);
break;
      }

    case EMERGENCY_STOP:
      {
int train_id;  /* identifier of the train */

p = GetInt(p, &train_id, "EmergencyStop.  ID of train to stop");
if (!p) continue;
/* all arguments successfully obtained */

EmergencyStop(train_id);
break;
      }

    case STATION_STOP:
      {
int train_id;  /* identifier of the train */
enum StationStopMode new_val;

p = GetInt(p, &train_id, "StationStop.  ID of train");
if (!p) continue;

sprintf(prompt, "New mode (%d = disabled, %d = enabled)",
```

```
SS_DISABLED, SS_ENABLED);
p = GetInt(p, &new_val, prompt);
/* all arguments successfully obtained */

StationStop(train_id, new_val);
break;
      }

    case GET_BLOCK_OCCUPANCY:
      {
int block_id;  /* identifier of the block */
enum Occupancy occ;  /* return value from the query */

p = GetInt(p, &block_id, "GetBlockOccupancy.  ID of block");
if (!p) continue;
/* all arguments successfully obtained */

occ = GetBlockOccupancy(block_id);

if (occ == OC_FREE)
  printf("Block %d is FREE (unoccupied).\n", block_id);
else if (occ == OC_OCCUPIED)
  printf("Block %d is OCCUPIED.\n", block_id);
else /* occ == OC_ERROR */
  printf("Unable to obtain occupancy for block %d\n", block_id);
break;
      }

    case GET_SWITCH_POSITION:
      {
int block_id;  /* identifier of the switch block */
enum SwitchPosit posit;  /* return value from the query */

p = GetInt(p, &block_id, "GetSwitchPosition.  ID of switch block");
if (!p) continue;
/* all arguments successfully obtained */

posit = GetSwitchPosition(block_id);

if (posit == SP_STRAIGHT)
  printf("Switch block %d is STRAIGHT\n", block_id);
else if (posit == SP_TURNED)
  printf("Switch block %d is TURNED\n", block_id);
else if (posit == SP_UNDEFINED)
```

```
  printf("Switch block %d has UNDEFINED position\n", block_id);
else /* posit == SP_ERROR */
  printf("Unable to obtain position of switch block %d\n", block_id);

break;
      }

    case GET_TRAIN_STATUS:
      {
int train_id;  /* identifier of the train */
enum TrainStatus status;  /* return value from the query */

p = GetInt(p, &train_id, "GetTrainStatus.  ID of train");
if (!p) continue;
/* all arguments successfully obtained */

status = GetTrainStatus(train_id);

if (status == TS_CRASHED)
  printf("Train %d has CRASHED\n", train_id);
else if (status == TS_RUNNING)
  printf("Train %d is RUNNING\n", train_id);
else /* status == TS_ERROR */
  printf("Unable to obtain status of train %d\n", train_id);
break;
      }

    case GET_TRAIN_MOTION:
      {
int train_id;  /* identifier of the train */
enum TrainMotion motion;  /* return value from the query */

p = GetInt(p, &train_id, "GetTrainMotion.  ID of train");
if (!p) continue;
/* all arguments successfully obtained */

motion = GetTrainMotion(train_id);

if (motion == TM_STOPPED)
  printf("Train %d is STOPPED\n", train_id);
else if (motion == TM_MOVING)
  printf("Train %d is MOVING\n", train_id);
else /* motion == TM_ERROR */
  printf("Unable to obtain motion information for train %d\n",
```

```
 train_id);
break;
      }

    case SET_SEQ_NUMBER:
      {
int new_val;  /* new value for sequence number */
int seq_num;  /* return value from SetSeqNumber */

p = GetInt(p, &new_val, "SetSeqNumber.  New sequence number value");
if (!p) continue;
/* all arguments successfully obtained */

seq_num = SetSeqNumber(new_val);

printf("Current sequence number value:  %d\n", seq_num);
break;
      }

    case NEW_SEQ_NUMBER:
      {
int seq_num;  /* return value from NewSeqNumber */

seq_num = NewSeqNumber();

printf("Current sequence number value:  %d\n", seq_num);
break;
      }

    case QUIT:
      {
char confirm[MAXLINE+1];  /* confirmation string */

printf("Quit.  Are you sure you want to quit?\n (y/n, default y) ");
gets(confirm);
/* confirmation response has been received */

if (*confirm == '\0' || *confirm == 'y' || *confirm == 'Y')
  exit (NORMAL);
break;
      }

    case HELP:
      {
```

```
struct command *cp;  /* pointer into command_table */

printf("%-34s%s\n-------------------------------------------------\n",
       "entry", "ACI command");
for (cp = command_table;  cp->code;  cp++) {
  printf("%-4s%-30s%s\n", cp->code, cp->args, cp->name);
}
break;
      }

    default:
      printf("Unrecognized command \"%s\"\nEnter ? for help\n", buff);
      break;
  };
  }
}
```

Appendix C

Code Listings: LLI

C.1 cpi.h, Interface Header File

```
/* $RCSfile: cpi.h,v $ $Revision: 1.65 $ $Date: 92/11/24 14:41:00 $ */
/* cpi.h -- cpi message formats.  */
/* This file automatically generated by m4 from cpi.hm */



/* Messages for initial and final communication with simulator */

/* first message for simulator, identifying self as a CPI client */
#define MSG_CPI_INIT "CPI %s%%"
#define MSG_CPI_INIT_ARGC 1
#define CreateMsgCPIInit(buff, str) sprintf(buff, MSG_CPI_INIT, (str))
#define ParseMsgCPIInit(buff, str) sscanf(buff, MSG_CPI_INIT, (str))


/* standard acknowledgement message format;  client ID is return value */
#define MSG_ACK "ack %s%%"
#define MSG_ACK_ARGC 1
#define CreateMsgAck(buff, str) sprintf(buff, MSG_ACK, (str))
#define ParseMsgAck(buff, str) sscanf(buff, MSG_ACK, (str))


/* simulator sends the following message to all clients upon normal exit */
#define MSG_QUIT_ALL "QUIT_ALL %s%%"
#define MSG_QUIT_ALL_ARGC 1
#define CreateMsgQuitAll(buff, str) sprintf(buff, MSG_QUIT_ALL, (str))
```

```
#define ParseMsgQuitAll(buff, str) sscanf(buff, MSG_QUIT_ALL, (str))



/*****************************************************************************/

/* Messages for configuration download.  RAB and JG 4/90 rev 5/91 */

#define MSG_CPI_DOWNLD_REQUEST "CPI Downld request from %s%%"
#define MSG_CPI_DOWNLD_REQUEST_ARGC 1
#define CreateMsgCPIDownldRequest(buff, str) \
  sprintf(buff, MSG_CPI_DOWNLD_REQUEST, (str))
#define ParseMsgCPIDownldRequest(buff, str) \
  sscanf(buff, MSG_CPI_DOWNLD_REQUEST, (str))


#define MSG_CPI_DOWNLD_GENERAL "C %d,%d,%lf,%lf,%lf,%d,%lf#"
typedef struct {
  int block_ct, train_ct;  /* number of blocks and trains in layout */
  double acc, emer_acc;  /* standard and emergency stop acceleration rates */
  double switch_time;  /* time required for switch block to change posit */
  int voting_quorum;  /* number of agreeing commands required for an action */
  double voting_window;  /* time limit after which commands expire */
} MsgCPIDownldGeneral;

#define MSG_CPI_DOWNLD_GENERAL_ARGC 7
#define MsgCPIDownldGeneralArgs(buff, datap) (buff, MSG_CPI_DOWNLD_GENERAL, \
  datap->block_ct , datap->train_ct , datap->acc , datap->emer_acc , \
  datap->switch_time , datap->voting_quorum , datap->voting_window )

#define CreateMsgCPIDownldGeneral(buff, datap) \
  sprintf MsgCPIDownldGeneralArgs(buff, (datap))
#define ParseMsgCPIDownldGeneral(buff, datap) \
  sscanf MsgCPIDownldGeneralArgs(buff, &(datap))


#define MSG_CPI_DOWNLD_TRAIN "T%d,%lf,%d,%lf,%d,%lf#"
typedef struct {
  int train_id;  /* numerical name of this train */
  double length;  /* length of train */
  int head_block;  /* block_id for block holding train head */
  double head_offset;  /* offset within head_block of train head */
  int tail_block;  /* block_id for block holding train tail */
  double tail_offset;  /* offset within tail_block of train tail */
```

```
} MsgCPIDownldTrain;

#define MSG_CPI_DOWNLD_TRAIN_ARGC 6
#define MsgCPIDownldTrainArgs(buff, datap) (buff, MSG_CPI_DOWNLD_TRAIN, \
  datap->train_id , datap->length , datap->head_block , datap->head_offset , \
  datap->tail_block , datap->tail_offset )

#define CreateMsgCPIDownldTrain(buff, datap) \
  sprintf MsgCPIDownldTrainArgs(buff, (datap))
#define ParseMsgCPIDownldTrain(buff, datap) \
  sscanf MsgCPIDownldTrainArgs(buff, &(datap))



#define MSG_CPI_DOWNLD_BLOCK "B%d,%lf,%d,%lf,%lf,%lf,%d#"
typedef struct {
  int block_id;  /* numerical name of this block */
  double length;  /* length of the block in meters */
  int type;  /* code for type classification of this block */
  double max_speed, min_speed;  /* speed above/below which derailment occurs*/
  double sta_stop_speed;  /* highest station stop speed---sta blocks only */
  int cross_id;  /* id for the cross block---for cross subblocks only */
} MsgCPIDownldBlock;

#define MSG_CPI_DOWNLD_BLOCK_ARGC 7
#define MsgCPIDownldBlockArgs(buff, datap) (buff, MSG_CPI_DOWNLD_BLOCK, \
  datap->block_id , datap->length , datap->type , datap->max_speed , \
  datap->min_speed , datap->sta_stop_speed , datap->cross_id )

#define CreateMsgCPIDownldBlock(buff, datap) \
  sprintf MsgCPIDownldBlockArgs(buff, (datap))
#define ParseMsgCPIDownldBlock(buff, datap) \
  sscanf MsgCPIDownldBlockArgs(buff, &(datap))


#define MSG_CPI_DOWNLD_LINK "L%d,%d,%d,%d,%d#"
typedef struct {
  int block_id;  /* numerical name of this block */
  int type;  /* code for type classification of this block */
  int tail, head1, head2;  /* block_ids for adjoining blocks */
} MsgCPIDownldLink;

#define MSG_CPI_DOWNLD_LINK_ARGC 5
#define MsgCPIDownldLinkArgs(buff, datap) (buff, MSG_CPI_DOWNLD_LINK, \
```

```
  datap->block_id , datap->type , datap->tail , datap->head1 , datap->head2 )

#define CreateMsgCPIDownldLink(buff, datap) \
  sprintf MsgCPIDownldLinkArgs(buff, (datap))
#define ParseMsgCPIDownldLink(buff, datap) \
  sscanf MsgCPIDownldLinkArgs(buff, &(datap))


/* The end of an cpi config download may be recognized by searching for
   CPI_DOWNLD_TERM beginning CPI_DOWNLD_TERM_OFFSET from final char received*/

#define CPI_DOWNLD_TERM 'D'
#define CPI_DOWNLD_TERM_OFFSET -4

#define MSG_CPI_DOWNLD_DONE "D %d#"
#define MSG_CPI_DOWNLD_DONE_ARGC 1
#define CreateMsgCPIDownldDone(buff, val) \
  sprintf(buff, MSG_CPI_DOWNLD_DONE, (val))
#define ParseMsgCPIDownldDone(buff, val) \
  sscanf(buff, MSG_CPI_DOWNLD_DONE, &(val))



/*****************************************************************************/

/* Commands */

#define MSG_CPI_SET_SWITCH "Xh%d,%d:%d:%d#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int block_id;  /* simulator block number */
  int new_posit; /* new switch position--0 for straight and 1 for turned */
} MsgCPISetSwitch;

#define MSG_CPI_SET_SWITCH_ARGC 4
#define MsgCPISetSwitchArgs(buff, datap) (buff, MSG_CPI_SET_SWITCH, \
  datap->client_id , datap->seq_num , datap->block_id , datap->new_posit )

#define CreateMsgCPISetSwitch(buff, datap) \
  sprintf MsgCPISetSwitchArgs(buff, (datap))
#define ParseMsgCPISetSwitch(buff, datap) \
  sscanf MsgCPISetSwitchArgs(buff, &(datap))
```

```
#define MSG_CPI_ACCEL "Xa%d,%d:%d:%lf#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int train_id;  /* simulator train number */
  double duration;  /* time to accelerate in seconds */
} MsgCPIAccel;

#define MSG_CPI_ACCEL_ARGC 4
#define MsgCPIAccelArgs(buff, datap) (buff, MSG_CPI_ACCEL, \
  datap->client_id , datap->seq_num , datap->train_id , datap->duration )

#define CreateMsgCPIAccel(buff, datap) sprintf MsgCPIAccelArgs(buff, (datap))
#define ParseMsgCPIAccel(buff, datap) sscanf MsgCPIAccelArgs(buff, &(datap))


#define MSG_CPI_DECEL "Xd%d,%d:%d:%lf#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int train_id;  /* simulator train number */
  double duration;  /* time to decelerate in seconds */
} MsgCPIDecel;

#define MSG_CPI_DECEL_ARGC 4
#define MsgCPIDecelArgs(buff, datap) (buff, MSG_CPI_DECEL, \
  datap->client_id , datap->seq_num , datap->train_id , datap->duration )

#define CreateMsgCPIDecel(buff, datap) sprintf MsgCPIDecelArgs(buff, (datap))
#define ParseMsgCPIDecel(buff, datap) sscanf MsgCPIDecelArgs(buff, &(datap))


#define MSG_CPI_SET_SPEED "Xv%d,%d:%d:%lf#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int train_id;  /* simulator train number */
  double goal_speed;  /* new desired velocity for the train */
} MsgCPISetSpeed;

#define MSG_CPI_SET_SPEED_ARGC 4
#define MsgCPISetSpeedArgs(buff, datap) (buff, MSG_CPI_SET_SPEED, \
  datap->client_id , datap->seq_num , datap->train_id , datap->goal_speed )
```

```
#define CreateMsgCPISetSpeed(buff, datap) \
  sprintf MsgCPISetSpeedArgs(buff, (datap))
#define ParseMsgCPISetSpeed(buff, datap) \
  sscanf MsgCPISetSpeedArgs(buff, &(datap))


#define MSG_CPI_SET_DIR "Xf%d,%d:%d:%d#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int train_id;  /* simulator train number */
  int new_dir;  /* indicator of new train direction */
} MsgCPISetDir;

#define MSG_CPI_SET_DIR_ARGC 4
#define MsgCPISetDirArgs(buff, datap) (buff, MSG_CPI_SET_DIR, \
  datap->client_id , datap->seq_num , datap->train_id , datap->new_dir )

#define CreateMsgCPISetDir(buff, datap) sprintf MsgCPISetDirArgs(buff, (datap))
#define ParseMsgCPISetDir(buff, datap) sscanf MsgCPISetDirArgs(buff, &(datap))


#define MSG_CPI_EMER_STOP "Xe%d,%d:%d#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int train_id;  /* simulator train number */
} MsgCPIEmerStop;

#define MSG_CPI_EMER_STOP_ARGC 3
#define MsgCPIEmerStopArgs(buff, datap) (buff, MSG_CPI_EMER_STOP, \
  datap->client_id , datap->seq_num , datap->train_id )

#define CreateMsgCPIEmerStop(buff, datap) \
  sprintf MsgCPIEmerStopArgs(buff, (datap))
#define ParseMsgCPIEmerStop(buff, datap) \
  sscanf MsgCPIEmerStopArgs(buff, &(datap))


#define MSG_CPI_STA_STOP "Xs%d,%d:%d:%d#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
```

```
  int train_id;  /* simulator train number */
  int new_val;  /*  0 to disable and 1 to enable */
} MsgCPIStaStop;

#define MSG_CPI_STA_STOP_ARGC 4
#define MsgCPIStaStopArgs(buff, datap) (buff, MSG_CPI_STA_STOP, \
  datap->client_id , datap->seq_num , datap->train_id , datap->new_val )

#define CreateMsgCPIStaStop(buff, datap) \
  sprintf MsgCPIStaStopArgs(buff, (datap))
#define ParseMsgCPIStaStop(buff, datap) sscanf MsgCPIStaStopArgs(buff, &(datap))



/*****************************************************************************/

/* Queries */

#define MSG_CPI_BLOCK_OCC "Xo%d,%d:%d#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int block_id;  /* simulator block number */
} MsgCPIBlockOcc;

#define MSG_CPI_BLOCK_OCC_ARGC 3
#define MsgCPIBlockOccArgs(buff, datap) (buff, MSG_CPI_BLOCK_OCC, \
  datap->client_id , datap->seq_num , datap->block_id )

#define CreateMsgCPIBlockOcc(buff, datap) \
  sprintf MsgCPIBlockOccArgs(buff, (datap))
#define ParseMsgCPIBlockOcc(buff, datap) \
  sscanf MsgCPIBlockOccArgs(buff, &(datap))


#define MSG_CPI_SWITCH_POSIT "Xw%d,%d:%d#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int block_id;  /* simulator block number */
} MsgCPISwitchPosit;

#define MSG_CPI_SWITCH_POSIT_ARGC 3
#define MsgCPISwitchPositArgs(buff, datap) (buff, MSG_CPI_SWITCH_POSIT, \
```

```
  datap->client_id , datap->seq_num , datap->block_id )

#define CreateMsgCPISwitchPosit(buff, datap) \
  sprintf MsgCPISwitchPositArgs(buff, (datap))
#define ParseMsgCPISwitchPosit(buff, datap) \
  sscanf MsgCPISwitchPositArgs(buff, &(datap))


#define MSG_CPI_TRAIN_STATUS "Xt%d,%d:%d#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int train_id;  /* simulator train number */
} MsgCPITrainStatus;

#define MSG_CPI_TRAIN_STATUS_ARGC 3
#define MsgCPITrainStatusArgs(buff, datap) (buff, MSG_CPI_TRAIN_STATUS, \
  datap->client_id , datap->seq_num , datap->train_id )

#define CreateMsgCPITrainStatus(buff, datap) \
  sprintf MsgCPITrainStatusArgs(buff, (datap))
#define ParseMsgCPITrainStatus(buff, datap) \
  sscanf MsgCPITrainStatusArgs(buff, &(datap))


#define MSG_CPI_TRAIN_MOTION "Xm%d,%d:%d#"
typedef struct {
  int client_id;  /* unique identifier for this client */
  int seq_num;  /* label for this command */
  int train_id;  /* simulator train number */
} MsgCPITrainMotion;

#define MSG_CPI_TRAIN_MOTION_ARGC 3
#define MsgCPITrainMotionArgs(buff, datap) (buff, MSG_CPI_TRAIN_MOTION, \
  datap->client_id , datap->seq_num , datap->train_id )

#define CreateMsgCPITrainMotion(buff, datap) \
  sprintf MsgCPITrainMotionArgs(buff, (datap))
#define ParseMsgCPITrainMotion(buff, datap) \
  sscanf MsgCPITrainMotionArgs(buff, &(datap))
```

C.2 aci.c, Example Using the LLI Interface

```
/* $RCSfile: aci.c,v $ $Revision: 1.1 $ $Date: 92/07/17 10:58:20 $ */
/* aci.c -- implementation of the Automatic Control Interface of the CPI
   R. Brown 6/91, based on specifications and earlier versions */

#include <stdio.h>  /* for standard error and NULL */
#include <sys/types.h>  /* required for acisys.h */
#include "cpi.h"  /* low-level interface of the CPI */
#include "aci.h"  /* for data type definitions */
#include "acisys.h"  /* operating system-specific routines */

#define BUFFSIZE 200  /* maximum size of a message */
#define RETRY_INTERVAL 5.0  /* seconds between retries for ConnectToSim */
#define QUERY_TIMEOUT 5.0  /* seconds before giving up on a query response */
#define EPSILON 0.0001  /* tolerance for floating point comparisons */

LayoutData layout_data, *ldp = &layout_data;
    /* data structure for storing the information passed in the download */
int chan;  /* socket for communicating with simulator */
int client_id;  /* identifier for this client, obtained from simulator*/
int seq_num = 0;  /* sequence number, for voting purposes */


/*****************************************************************************/

/* SendInitMessage sends the simulator the initialization message for this
   application.  Returns nonzero on success, zero on failure.
   RAB 11/30/89 */

int
SendInitMessage(chan, string)
     int chan;  /* channel descriptor for communication with simulator */
     char string[];  /* string to send as part of the message to sim */
{
  char buff[BUFFSIZE]; /* buffer for init message and ack */
  int n;  /* length of received init message */
  char client_id_string[BUFFSIZE];  /* string representing client_id */

  /* send the initialization message */
  sprintf(buff, MSG_CPI_INIT, string);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("SendInitMessage:  SendMsg failed");
    return (0);
```

```
  }

  /* wait for, verify acknowledgement */
  if ((n=RecvMsg(chan, buff, BUFFSIZE)) < 0) {
    perror("SendInitMessage:  can't receive acknowledgement");
    return (0);
  }
  buff[n] = '\0';
  if (sscanf(buff, MSG_ACK, client_id_string) != 1) {
    fprintf(stderr, "SendInitMessage:  expected ack, received \"%s\"\n", buff);
    return (0);
  }

  sscanf(client_id_string, "%d", &client_id);  /* extract unique id */

  return (1);
}


/*****************************************************************************/

/*
   We trust the simulator's download... */

LayoutData *
GetDownload(hostname, simnum, progname, timeout)
     char *hostname;  /* machine running a simulation */
     int simnum;  /* identifies a running simulation */
     char *progname;  /* name of invoking program */
     double timeout;  /* in seconds */
{
  static GetDownloadSucceeded = 0;  /* set non-zero on first successful call */
  char buff[BUFFSIZE];  /* buffer for holding one message */
  union {
    MsgCPIDownldGeneral gen;  /* for unpacking message of general attributes */
    MsgCPIDownldBlock block;  /* for unpacking message of block attributes */
    MsgCPIDownldLink link;    /* for unpacking message of block attachments */
    MsgCPIDownldTrain train;  /* for unpacking message of train attributes */
    int val;  /* for receiving integer value passed with MsgCPIDownldDone */
  } msg;
  int incomplete;  /* non-zero if the download is found to be incomplete */
  int recv_msg_status;  /* return value from RecvMsg */
  int i;  /* loop control */
```

```
if (GetDownloadSucceeded) {
  fprintf(stderr, "GetDownload has already been called successfully\n");
  return (NULL);
}
/* there have been no prior successful calls */

if (timeout <= 0) {
  fprintf(stderr, "GetDownload:  Non-positive timeout %.1f\n", timeout);
  return (NULL);
}
/* valid timeout was specified */

if (!*hostname) {
  GetLocalHost(buff, sizeof(buff));
  hostname = buff;
}

while ((chan = ConnectToSim(hostname, simnum)) == -1) {
  timeout -= RETRY_INTERVAL;
  if (timeout < EPSILON) {
    fprintf(stderr, "GetDownload:  Could not connect to simulator\n");
    return (NULL);
  } else {
    fprintf(" retrying...\n");
    Sleep(RETRY_INTERVAL);
  }
}
/* a connection to the desired simulation has been established */

if (!SendInitMessage(chan, progname)) {
  fprintf(stderr, "GetDownload:  Could not send init message\n");
  CloseChan(chan);
  return (NULL);
}
/* CPI initialization message has been sent and acknowledged */

CreateMsgCPIDownldRequest(buff, progname);
if (SendMsg(chan, buff, strlen(buff)) < 0 ) {
  perror("MsgCPIDownldRequest SendMsg failed");
  fprintf(stderr, "GetDownload:  Could not send request for download\n");
  CloseChan(chan);
  return (NULL);
}
```

```
  /* Download has been requested.
     The simulator's expected response is the download itself. */


  /* receive download */

  /* invar:  all CPIDownld messages so far have been parsed */
  while ((recv_msg_status = RecvDownldMsg(chan, buff, BUFFSIZE)) > 0 &&
 ParseMsgCPIDownldDone(buff, msg.val) !=
                        MSG_CPI_DOWNLD_DONE_ARGC) {
    /* new message of length recv_msg_status that is not DownldDone
       has been received */

    buff[recv_msg_status] = '\0';

    if (ParseMsgCPIDownldGeneral(buff, &msg.gen) ==
                        MSG_CPI_DOWNLD_GENERAL_ARGC) {
      ldp->block_ct = msg.gen.block_ct;
      ldp->train_ct = msg.gen.train_ct;
      ldp->acc = msg.gen.acc;
      ldp->emer_acc = msg.gen.emer_acc;
      ldp->switch_time = msg.gen.switch_time;
      ldp->voting_quorum = msg.gen.voting_quorum;
      ldp->voting_window = msg.gen.voting_window;

      if (ldp->block_ct > MAX_BLOCKS || ldp->train_ct > MAX_TRAINS) {
fprintf(stderr,
"GetDownload:  MAX_TRAINS (%d) or MAX_BLOCKS (%d) too small\n",
MAX_TRAINS, MAX_BLOCKS);
return (NULL);
      }

    } else if (ParseMsgCPIDownldBlock(buff, &msg.block) ==
                        MSG_CPI_DOWNLD_BLOCK_ARGC) {
      int index = msg.block.block_id - 1;
        /* this block's index in ldp->blocks[] */

      ldp->blocks[index].block_id = msg.block.block_id;
      ldp->blocks[index].length = msg.block.length;
      ldp->blocks[index].type = (enum BlockType) msg.block.type;
      ldp->blocks[index].max_speed = msg.block.max_speed;
      ldp->blocks[index].min_speed = msg.block.min_speed;
      if ((enum BlockType) msg.block.type == BT_STATION)
ldp->blocks[index].t.st.sta_stop_speed = msg.block.sta_stop_speed;
```

```
      if ((enum BlockType) msg.block.type == BT_CROSS)
ldp->blocks[index].t.cr.cross_id = msg.block.cross_id;

    } else if (ParseMsgCPIDownldLink(buff, &msg.link) ==
                        MSG_CPI_DOWNLD_LINK_ARGC) {
      int index = msg.link.block_id - 1;
        /* this block's index in ldp->blocks[] */

      switch (ldp->blocks[index].type) {
      case BT_REGULAR:
      case BT_STATION:
      case BT_CROSS:
ldp->blocks[index].t.rg.tail = msg.link.tail;
ldp->blocks[index].t.rg.head = msg.link.head1;
break;
      case BT_JOIN:
ldp->blocks[index].t.jn.tail = msg.link.tail;
ldp->blocks[index].t.jn.head1 = msg.link.head1;
ldp->blocks[index].t.jn.head2 = msg.link.head2;
break;
      case BT_SWITCH:
ldp->blocks[index].t.sw.tail = msg.link.tail;
ldp->blocks[index].t.sw.straight = msg.link.head1;
ldp->blocks[index].t.sw.turn = msg.link.head2;
break;
      }

    } else if (ParseMsgCPIDownldTrain(buff, &msg.train) ==
                        MSG_CPI_DOWNLD_TRAIN_ARGC) {
      int index = msg.block.block_id - 1;
        /* this train's index in ldp->trains[] */

      ldp->trains[index].train_id = msg.train.train_id;
      ldp->trains[index].length = msg.train.length;
      ldp->trains[index].head.block = msg.train.head_block;
      ldp->trains[index].head.offset = msg.train.head_offset;
      ldp->trains[index].tail.block = msg.train.tail_block;
      ldp->trains[index].tail.offset = msg.train.tail_offset;

    } else { /* unrecognized message format */
      fprintf(stderr, "GetDownload:  Unrecognized message from simulator\n");
      fprintf(stderr, "%s\n", buff);
      return (NULL);
    }
```

```
}

if (recv_msg_status <= 0) {
  fprintf(stderr, "GetDownload:  Failed in attempt to receive a message\n");
  return (NULL);
}
/* There were no failures to receive a message, and
   all received messages have been successfully parsed and data stored in
   layout_data, and one Done message was received */


/* check to see whether download was complete */

incomplete = 0;
/* we will assume that all of the fields from MsgCPIDownldGeneral were
   received and stored properly in layout_data if block_ct is nonzero */
if (!ldp->block_ct) {
  fprintf(stderr, "GetDownload:  incomplete download (missing General)\n");
  incomplete++;
}

for (i = 0;  i < ldp->block_ct;  i++) {
  /* we will assume that all of the fields from MsgCPIDownloadBlock were
     received and stored properly in layout_data if block_id is nonzero */
  if (!ldp->blocks[i].block_id) {
    fprintf(stderr, "GetDownload:  incomplete download (missing Block %d)\n",
    i + 1);
    incomplete++;
  }
  /* we will assume that all of the fields from MsgCPIDownloadLink were
     received and stored properly in layout_data if tail is nonzero */
  if (!ldp->blocks[i].t.rg.tail) {
    fprintf(stderr, "GetDownload:  incomplete download (missing Link %d)\n",
    i + 1);
    incomplete++;
  }
}

for (i=0;  i < ldp->train_ct;  i++)
  /* we will assume that all of the fields from MsgCPIDownloadTrain were
     received and stored properly in layout_data if train_id is nonzero */
  if (!ldp->trains[i].train_id) {
    fprintf(stderr, "GetDownload:  incomplete download (missing Train %d)\n",
    i + 1);
```

```
      incomplete++;
    }

  if (incomplete)
    return (NULL);
  /* a complete download was successfully received */

  GetDownloadSucceeded++;
  return (&layout_data);
}


/*********************************************************************/

/* commands */

void
SetSwitch(block_id, new_posit)
     int block_id;
     enum NewPosit new_posit;
{
  MsgCPISetSwitch msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.block_id = block_id;
  msg.new_posit = (int) new_posit;

  CreateMsgCPISetSwitch(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("SetSwitch:  SendMsg failed");
  }
}


void
Accelerate(train_id, duration)
     int train_id;
     double duration;
{
  MsgCPIAccel msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */
```

```
  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.train_id = train_id;
  msg.duration = duration;

  CreateMsgCPIAccel(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("Accelerate:  SendMsg failed");
  }
}


void
Decelerate(train_id, duration)
     int train_id;
     double duration;
{
  MsgCPIDecel msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.train_id = train_id;
  msg.duration = duration;

  CreateMsgCPIDecel(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("Decelerate:  SendMsg failed");
  }
}


void
SetSpeed(train_id, goal_speed)
     int train_id;
     double goal_speed;
{
  MsgCPISetSpeed msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.train_id = train_id;
  msg.goal_speed = goal_speed;
```

```
  CreateMsgCPISetSpeed(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("SetSpeed:  SendMsg failed");
  }
}


void
SetDirection(train_id, new_dir)
     int train_id;
     int new_dir;
{
  MsgCPISetDir msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.train_id = train_id;
  msg.new_dir = new_dir;

  CreateMsgCPISetDir(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("SetDirection:  SendMsg failed");
  }
}


void
EmergencyStop(train_id)
     int train_id;
{
  MsgCPIEmerStop msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.train_id = train_id;

  CreateMsgCPIEmerStop(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("EmergencyStop:  SendMsg failed");
  }
}
```

```
void
StationStop(train_id, new_val)
     int train_id;
     enum StationStopMode new_val;

{
  MsgCPIStaStop msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.train_id = train_id;
  msg.new_val = (int) new_val;

  CreateMsgCPIStaStop(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("StationStop:  SendMsg failed");
  }
}


/*****************************************************************/

/* Queries */


enum Occupancy
GetBlockOccupancy(block_id)
     int block_id;
{
  MsgCPIBlockOcc msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */
  int n;  /* return status from RecvMsgTimed */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.block_id = block_id;

  CreateMsgCPIBlockOcc(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("GetBlockOccupancy:  SendMsg failed");
  }
```

```
  if ((n = RecvMsgTimed(chan, buff, BUFFSIZE, QUERY_TIMEOUT)) < 0)
    return (OC_ERROR);
  /* response successfully received from simulator and stored in buff */

  switch (buff[0]) {
  case 'o':  return (OC_OCCUPIED);
  case 'f':  return (OC_FREE);
  default:
    buff[n] = '\0';
    fprintf(stderr, "GetBlockOccupancy:  unknown simulator response \"%s\"\n",
    buff);
    return (OC_ERROR);
  }
}


enum SwitchPosit
GetSwitchPosition(block_id)
     int block_id;
{
  MsgCPISwitchPosit msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */
  int n;  /* return status from RecvMsgTimed */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.block_id = block_id;

  CreateMsgCPISwitchPosit(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("GetSwitchPosition:  SendMsg failed");
  }
  if ((n = RecvMsgTimed(chan, buff, BUFFSIZE, QUERY_TIMEOUT)) < 0)
    return (SP_ERROR);
  /* response successfully received from simulator and stored in buff */

  switch (buff[0]) {
  case 's':  return (SP_STRAIGHT);
  case 't':  return (SP_TURNED);
  case 'u':  return (SP_UNDEFINED);
  default:
    buff[n] = '\0';
    fprintf(stderr, "GetSwitchPosition:  unknown simulator response \"%s\"\n",
```

```
    buff);
    return (SP_ERROR);
  }
}


enum TrainStatus
GetTrainStatus(train_id)
     int train_id;
{
  MsgCPITrainStatus msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */
  int n;  /* return status from RecvMsgTimed */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.train_id = train_id;

  CreateMsgCPITrainStatus(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("GetTrainStatus:  SendMsg failed");
  }
  if ((n = RecvMsgTimed(chan, buff, BUFFSIZE, QUERY_TIMEOUT)) < 0)
    return (TS_ERROR);
  /* response successfully received from simulator and stored in buff */

  switch (buff[0]) {
  case 'c':  return (TS_CRASHED);
  case 'r':  return (TS_RUNNING);
  default:
    buff[n] = '\0';
    fprintf(stderr, "GetTrainStatus:  unknown simulator response \"%s\"\n",
    buff);
    return (TS_ERROR);
  }
}


enum TrainMotion
GetTrainMotion(train_id)
     int train_id;
{
  MsgCPITrainMotion msg;  /* data structure for message construction */
  char buff[BUFFSIZE];  /* buffer for outgoing message */
```

```
  int n;  /* return status from RecvMsgTimed */

  msg.client_id = client_id;  /* identifier of this client */
  msg.seq_num = seq_num;  /* voting sequence number */
  msg.train_id = train_id;

  CreateMsgCPITrainMotion(buff, &msg);
  if (SendMsg(chan, buff, strlen(buff)) < 0) {
    perror("GetTrainMotion:  SendMsg failed");
  }
  if ((n = RecvMsgTimed(chan, buff, BUFFSIZE, QUERY_TIMEOUT)) < 0)
    return (TM_ERROR);
  /* response successfully received from simulator and stored in buff */

  switch (buff[0]) {
  case 's':  return (TM_STOPPED);
  case 'm':  return (TM_MOVING);
  default:
    buff[n] = '\0';
    fprintf(stderr, "GetTrainMotion:  unknown simulator response \"%s\"\n",
    buff);
    return (TM_ERROR);
  }
}


/*********************************************************************/

/* Voting Sequence Number */

int
SetSeqNumber(new_val)
     int new_val;
{
  if (new_val < 0)
    return (-1);
  else {
    seq_num = new_val;
    return (seq_num);
  }
}


int
```

```
NewSeqNumber()
{
  return (++seq_num);
}
```

Appendix D

Reference Pages

Name

acitest - manual test of CPI interface in Trainset

Syntax

acitest [-d] [simhost [simnum]]

Description

acitest is an example program included with the Trainset software. Its source code illustrates the use of the upper layer (called the ACI) of the Control Program Interface (CPI). When invoked and connected to a running railroad simulation (see tsim(1)), it enables a user to interactively perform each ACI command and query. The effects of each action may be observed using the monitor programs tspanel(1) and tsview(1).

Options and Arguments

-d Causes the state information received from a simulation to be printed during startup.

simhost

Specifies the name of a host that is running a simulation of a railroad. If an empty string is specified, the local host is used. If the simhost argument is omitted, a value is requested interactively.

simnum

An integer that determines which invocation of the tsim simulator (at the indicated host) is to be used. The default value is 0. The monitor programs and the desired invocation of tsim should use the same value for number.

See Also

ts(1), tsim(1), tspanel(1), tsview(1).

Name

ts - launch Trainset applications

Syntax

ts [flags...]
ts -edit [file] [-layout file] [-tsroot dir] [X11-options]

Description

ts invokes (''launches'') applications related to the Trainset railroad simulation software in a computing environment that may involve a heterogeneous network of computing systems. With no options, ts typically starts a demonstration consisting of a simulation of a sample railroad, an X11 application that graphically displays the current state of the railroad, and another X11 application that provides for manual control of the railroad. ts options enable the user to select which applications are to be launched, choose the railroad layout to be simulated, specify the host to launch from, etc.

Programmers may write software that interacts with Trainset, including programs that automatically control a railroad and custom versions of the basic Trainset software. ts provides a configuration mechanism for adding launch information describing such programs to its database.

When invoked with -edit as the first command-line option, ts invokes the tsed editor for railroad layouts and launches no other applications. The options -tsroot and -layout are recognized by ts in the case of -edit, and may appear either in an environment variable TSOPTS or on the command line. file names the layout data file to be edited. file may be presented as the second command line argument if it does not begin with a dash '-'. It may also be provided using the -layout option. -tsroot specifies the tsed search paths for layout data files, as discussed in Search Paths below. In addition, any additional options on the command line, in

particular options recognized by X11, are passed on to tsed.

Applications

Trainset consists of application programs that support interaction with a simulated railroad. Railroads are represented by layouts that consist of trains and blocks of track. Users of Trainset may write programs that use the Control Program Interface (CPI) to control such a railroad. For more information, see the manuals for Trainset.

The standard applications that comprise Trainset include the following.

tsim	Simulator of railroad layouts.
tsed	Editor for interactive creation and modification of a layout.
tsview	A graphical display of the state of a layout during a simulation.
tspanel	A graphical control panel for manually operating a layout's trains and switch blocks.

In addition, programs that demonstrate the CPI, including acitest and demo, are provided in the Trainset distribution.

Control Codes

In ts, applications are referenced by control codes that are strings consisting of an uppercase letter optionally followed by any combination of lowercase letters, digits, underscores and periods. The following control codes are conventionally defined in the main configuration file for ts:

Sim, S, Tsim	Trainset railroad simulator.

Panel, P, Tspanel
: Control panel application.

Viewer, V, Tsview
: Viewer application for observing a simulation.

Acitest, A
: Acitest program that allows manual experimentation with CPI interface features.

Demo, D
: Demonstration of the CPI interface discussed in the Trainset manual.

Any of the ts options -args, -delay, -direct, -display, -host, -tsroot, -xterm and -xtermargs can be localized to apply to a specific application by prefixing that option with the appropriate control code(s). For example, either 'Sim:-host loki' or 'S:-host loki' specifies that the simulator should be launched at the site loki. Also, 'PV:-display thor:0.0' causes the panel and viewer applications to display graphics output on the X11 server thor:0.0. Arbitrary arguments can be passed to applications using the -args option prefixed by the appropriate control code, e.g., 'Myprog:-args "-level 4 -trace"'. Quoting (e.g., using '"') enables spaces to be included in the argument string.

If the colon that terminates a prefix is followed immediately by a character (e.g., dash '-') that cannot be part of a control code name, then that colon may be omitted. For example, 'Sim-host loki' is equivalent to 'Sim:-host loki'.

An alias facility is provided for associating multiple control codes with a single application. Aliases may be declared in configuration files (discussed below) or by using the -alias option. The alias feature can be used to change the associations between control codes and applications. For example, suppose that a control code Train_control_2 is associated with an application that has

been written locally. Then the ts option '-alias T=Train_control_2' assigns the control code T to represent that local application. Train_control_2 is a better identifier, but the alias T is more convenient as a prefix. Also, '-alias Demo=T' could be used to associate the control code Demo with that local application instead of the standard Trainset demonstration program.

The -dup option may be used to invoke multiple copies of an application. This feature supports the launching of replicated programs. -dup clones the launch information gathered so far for the application in question and associates a control code with the copy. The launch attributes in the two copies may thereafter be modified independently.

Search Paths

The directory tsroot identifies the location of the Trainset installation in the local file system. Tsroot may be specified as the value of an environment variable TSROOT or on the command line using the -tsroot option.

Ts searches for its own configuration files in the directories ., ./lib, tsroot and tsroot/lib (in that order).

Layout data files for a simulation are sought in the directories ., ./layouts, tsroot, tsroot/layouts and tsroot/trainset/layouts. The default extension '.l' is appended (if omitted) to a layout data file name by Trainset applications and searching is repeated if a file with the given name was not found. If no layout data file is specified, the file tsroot/layouts/sample.l is used.

Binaries for Trainset applications are sought using the environment variable PATH as usual.

Launching Applications

Unless the -edit option is specified, ts gathers information

from the following sources, in order, before launching applications.

Main configuration file

This file associates control codes with applications, contains invocation information for applications (e.g., name of executable and default arguments) and provides a default list of applications to be launched. See ts.config(5) for the file format. The name of the main configuration file may be specified with the -config option. The default main configuration file is tsroot/ts.config.

Auxiliary configuration files

One or more additional configuration files may be designated to supplement the main configuration file. These are specified by the ts options -confign where n is a single digit. Auxiliary configuration files are processed in increasing order of n. The values of n need not be contiguous. Auxiliary configuration files enable individual users and groups of users to install their own applications and customize launching information for applications that appear in the main configuration file. Attributes for the launch are either overridden or appended as appropriate. If a non-empty list of applications to be launched is included in an auxiliary configuration file, it overrides any prior list. There are no default auxiliary configuration files.

Environment variables TSROOT, TSOPTS and DISPLAY

If there is an environment variable TSROOT, it is treated as the default value for tsroot (see Search Paths above). If there is an environment variable TSOPTS, it is treated as a list of ts options that are parsed prior to the options specified on the command line. If there is an environment variable DISPLAY with value X11-display, then the option -xtermargs X11-display is implicitly processed just prior to any

options in TSOPTS.

Command line options
See Options below.

The significance of the ordering above is that later values override or are appended to earlier values. For example, a -host option in TSOPTS overrides any HOST entries in configuration files, and 'Sim:-args "-window 5"' on the command line appends to the arguments for the Sim application that were collected from configuration files and TSOPTS.

Launching at a remote host is accomplished by invoking ts at the remote site, as described in ts.remote(8). The information that is passed to a remote ts consists of an options list that is assembled by the local ts after scanning the local sources listed above. That options list consists of the following.

A -launch option listing the applications to be launched at that site.

A -display option, unless no -display attribute was given and there is no local DISPLAY environment variable.

Any applicable control-code specific -display options.

-simhost and -simnum options.

Any local command-line options that remain after removing -host and -launch (+) options.

The remote ts gets its launch information first from remote-site configuration files, then from remote-site environment variables and finally from the options list that was assembled at the local site. (Note that values for attributes such as -delay and -tsroot may be passed to the remote site only if they are specified in local command-line

options.)

ts provides two methods for launching applications.

xterm Method
: If the configuration file attribute XTERM for an application is a string of positive length or if the -xterm option is used, then the application is invoked in an xterm(1) window that is created for that purpose. That application's standard input and output will be associated with the xterm window. Arguments can be passed to xterm using the -xtermargs option discussed below.

Direct Method
: Applications that are launched without using the xterm method are invoked asynchronously by the ts process, and share that process's standard output. The standard input for all such applications is /dev/null, except the last application launched by the direct method inherits the ts process's standard input as well as standard output. Thus, an interactive application (e.g., acitest(1)) may be launched using the direct method provided that it the last such application specified. See the Examples section.

xterm is the default launch method. The direct method is currently implemented for the local host only, i.e., the host at which the ts command was entered. Thus, any -host options are ignored for applications launched by the direct method.

Options

-alias control1=control2
: Declares the control code control1 to be an alias for the control code control2. The two codes may then be used interchangeably for referring to the application launch attributes associated with control2, if there is such an application. The symbol '=' is optional.

-dup control1=control2
Make a copy control1 of the application launch attributes associated with control2. Since the two control codes represent different copies, a change in the attributes referenced by control1 will have no effect on the attributes referenced by control2. This is useful for invoking multiple copies of the same application, e.g., parallel copies of the same CPI program for controlling a railroad. The symbol '=' is optional.

-args arglist
Provides program-specific arguments to be passed to applications. This option is ordinarily prefixed by control codes, e.g., 'Sim:-args "-update 0.5"'.

-config filename
Specifies the name of the main configuration file.

-confign filename
Specifies the name of the nth auxiliary configuration file.

-delay seconds
Specifies an integer number of seconds to pause just after launching each application.

-direct
Specifies that applications should be launched using the direct method.

-display X11-display
Indicates the display for X11 output. The environment variable DISPLAY is set to X11-display for the applications that are launched, and '-display X11-display' is appended to the list of arguments for xterm. The latter effect is approximately equivalent to -xtermargs "-display X11-display"; this does not affect applications launched by the direct method, but such applica-

tions can still obtain the value X11-display by examining the environment variable DISPLAY.

-edit filename
Invoke the tsed editor to create or modify a layout data file of blocks and trains to be simulated. If filename is a relative path (does not begin with '/'), a file with that name is searched for using the search path for layouts. If such a file is found, its path is passed to tsed for modification. Otherwise, filename is passed, so that a new layout data file with that name may be created relative to the current working directory. The -edit option should not be used in combination with any other ts options.

-host hostname
ts seeks configuration files and invokes executables at the site hostname, except when specifically overridden by other ts options.

-layout filename
The railroad layout to be simulated is described in filename, which must be in the format produced by tsed. See Search Paths above.

-launch control...
Specifies that the indicated applications (only) should be invoked. The default list of applications to launch is determined by the first lines of configuration files.

-quorum n
Sets the simulator voting quorum value to n. Equivalent to 'Sim:-args "-quorum n"'.

-tsroot dirname
Specifies the value of tsroot.

-simhost hostname

Specifies the host on which to invoke the simulator. Equivalent to 'Sim:-host hostname'.

-simnum number
number determines which well-known port is to be used when initializing communication with a railroad simulation. Permissible values for number are installation-dependent and typically include the range 0 to 99. Applications that are launched with a given value for number can only connect with a simulation that was launched with that same -simnum value number; thus, multiple railroad simulations may be run simultaneously on the same host if they use different values for number. It is convenient to assign several unique values of number to each Trainset user in order to avoid collisions among users.

-update seconds
Specifies the frequency that the simulator sends state update reports to graphics monitor programs (control panel and viewer). seconds may be a decimal value, e.g., 0.5. Equivalent to 'Sim:-args "-update seconds"'.

-window seconds
Sets the simulator voting window value to seconds. Equivalent to 'Sim:-args "-window seconds"'.

-xterm
Specifies that applications should be launched using the xterm method.

-xtermargs args
Provides arguments to be passed to xterm(1), for applications to be launched using the xterm method. Enclose args in quotes if it contains embedded spaces.

+control...
Same as -launch control.

args Any options that are not listed above are passed to xterm(1) for applications launched using the xterm method. Thus, the options args is approximately equivalent to -xtermargs "args".

Examples

Start the standard programs (as listed in in configuration files) using their default methods. Use the default layout data file to define the railroad being simulated. Send all graphics output to the X11 server specified in the environment variable DISPLAY, and interpret any options listed in the environment variable TSOPTS.

```
ts
```

Seek a file oval.l as described in Search Paths above for layout data files. If it is found, then start the standard programs as before, with the simulator tsim using that file oval.l to define the railroad layout.

```
ts -layout oval.l
```

Start the standard programs as before, except send output for the Panel application to thor:0 and send all other graphics output to the X11 server loki:0. Send updated information to any running graphics monitor programs (e.g., Viewer and Panel) every 3/4 second.

```
ts -display loki:0 -update .75 Panel:"-display thor:0"
```

Launch the applications S, V and A using their default methods. Unless these control codes have been reassigned, they refer to the Simulator, Viewer and Acitest applications.

```
ts +SVA
```

Launch the applications S, V and A using the direct method. Output for the various applications will be intermingled on the screen. Since A is specified last, that application can be operated interactively. The direct method is useful for debugging new applications and for systems that do not support the program xterm(1).

```
ts +SVA -direct
```

Bugs

The remote invocation facility is not currently implemented.

Unpredictable results may occur if a control code is made a duplicate of itself.

See Also

tsed(1), tsim(1), tspanel(1), tsview(1), ts.config(5), ts.remote(8).

Name

tsed - Trainset layout editor

Syntax

tsed [file] [-layout file] [-tsroot dir] [X11-options]

Description

tsed is an interactive graphics editor for defining and modifying railroad layouts for use in the Trainset railroad simulation software. On startup, tsed displays two windows: (1) a canvas window on which the layout is created. The window contains a set of pulldown menus along the top, a message area at the bottom, and is overlaid with an alignment grid, and (2) a tools window consisting of icons representing the operations available for creating and modifying blocks and trains.

With no file or options specified, tsed starts with a blank unnamed layout. The options -tsroot and -layout are recognized by tsed and may appear either in the environment variable TSOPTS or on the command line. file names the layout data file to be edited. file may be presented as the second command line argument if it does not begin with a dash '-'. It may also be provided via the -layout option. -tsroot identifies the location of the ts installation in the file system and is used to determine the location of auxiliary files needed by tsed. These are searched for in tsroot, tsroot/lib and tsroot/trainset/lib if tsroot is specified and then according to the PATH environment variable. tsed uses tsroot to search for layout data files in the following order: ., ./layouts, tsroot, tsroot/layouts and tsroot/trainset/layouts. The default extension '.l' is appended to a layout data file name and the search repeated if the given file was not found.

tsed also recognizes the standard X11 application options.

Layout Tools

The layout tools lie in two columns of icons in the tools window and are called (top to bottom, left to right) Select, Erase, Straight Block, Station Block, Arc Block 1, Arc Block 2, Cross Block, Join Block, Switch Block 1, Switch Block 2, Train, and Rotate. Select, Erase, and Rotate manipulate existing blocks. Straight Block and Station Block create linear blocks. Arc Block 1 and Arc Block 2 create circular arc blocks. Cross Block, Join Block, Switch Block 1, and Switch Block 2 create fixed size iconic blocks. The current tool is shown highlighted by inverting its colors and is set by clicking on it. Clicking the left mouse button in the canvas area invokes the current layout tool. tsed provides the following layout tools:

Select — Move block labels by dragging within their boundaries. Select a block by clicking on it whether or not it's already selected and discard any other selection.

Erase — Remove a block or train from a layout by clicking on it.

The four tools, Straight Block, Station Block, Arc Block 1, and Arc Block 2, create regular and station blocks using a drag technique to determine the position of the head and tail terminators of the block. If the start point or end point of the drag is within a few pixels of the terminator of an existing block, tsed attempts to establish a connection. In such a case, tsed constrains the slope of the new block to match the slope of the existing block at the terminator.

Straight Block — Create a straight regular block. The block is constrained to be vertical, horizontal, or at + or - 45 deg diagonal.

Station Block — Create a station block. Created and constrained in the same way as straight block

above.

Arc Block 1 Create a circular arc regular block. The start point of a drag determines the location of the head terminator of the block. The radius of the circular arc is two grid divisions. The extent of the arc is determined by the angle given by the head terminator and the end point of the drag. The extent of the arc is constrained to be a multiple of 45 deg.

Arc Block 2 The same as Arc Block 1 but with a radius of four grid divisions.

The four tools, Cross Block, Join Block, Switch Block 1, and Switch Block 2, create iconic blocks, blocks that have a fixed size and shape. They are created by simply clicking the mouse, whose cursor takes the form of the icon for the current tool. Along the outer circle of such a cursor are enlarged points called hot spots at which connections with other blocks can be made. When a connection is established, the newly created iconic block is constrained so that the slope at the connecting hot spot and the slope at the existing terminator agree. tsed does not establish more than one connection when an iconic block is created.

Cross Block Create a cross block.

Join Block Create a join block.

Switch Block 1 Create a switch block whose turn head is 45 deg counter clockwise from its straight head.

Switch Block 2 Create a switch block whose turn head is 45 deg clockwise from its straight head.

Train Create a train. The drag operation for creating a train begins at the position

desired for the head end of a train, continues along the blocks to be occupied by that train, and stops at the position desired for that train's tail end. The head end of a train is indicated by an angle bracket, and the tail end is indicated by a square bracket. All blocks that are occupied by a train are highlighted. The head end of a train is constrained to start in a regular or station block.

Rotate — Rotate an iconic block clockwise about its center to the next hot spot.

Pulldown Menus

The pulldown menus File and Customize lie in a menu bar along the top of the canvas window. They contain commands which you execute by pulling down the menu and releasing the mouse button on the command.

The File menu contains the following commands to operate on files:

New — Clear the current layout and create a new empty layout, requesting a layout name from the user.

Open — Open an existing layout using a layout data file typed by the user. The search is the same as specified above.

Save — Save the current layout under the current file name, if one exists. If one does not, then it is the same as executing Save As ...

Save As ... — Save the current layout under a new name specified by the user.

Close — Close the current layout and clear the can-

vas.

Quit — Exit the application after first checking for any unsaved changes.

The Customize menu contains the following commands to alter the appearance of the current layout and the attributes of the blocks contained within:

Change Block Attributes — Change the attributes of a selected block (See Select above). If no block is selected, a warning beep is sounded.

Change Default Block Attributes — Change the default creation attributes of a block type. This is a hierarchical menu whose submenu are the various block types.

Labels — Change the appearance of the labels on the current layout. This is a hierarchical menu whose submenu entries are: Names -- Labels are block names; Speeds -- Labels indicate the minimum, maximum, and stop velocities associated with each block; Hide Labels -- Hide the labels in the current layout.

Hide/Show Grid — Hide (Show) the grid lines in the canvas window. Note that this does not turn the grid alignment and constraints off, it merely removes the grid from view.

Files

tsroot/lib/tsed.uid

See Also

ts(1), layout data file description

Bugs / Restrictions

Due to a bug in the DEC Xqdsg server, tsed causes the X server to crash when using the Arc Block 1 and Arc Block 2 tools with X servers from Ultrix 4.0 and Ultrix 4.1. There is no problem in Ultrix 4.2 and the Ultrix 4.2 X server can be used with Ultrix 4.0 and 4.1.

Name

tsim - Trainset railroad simulation

Syntax

tsim [flags...]

Description

tsim simulates a Trainset railroad. The Trainset software consists of this simulator, an interactive graphics editor tsed(1) for defining railroad layouts and graphics monitor programs tspanel(1) and tsview(1) for displaying the state of the railroad and manually controlling it. See The Trainset Railroad Simulation for more information.

The programs tsim, tspanel and tsview are ordinarily invoked using the launch program ts(1).

Interfaces

tsim provides three interfaces. The control program interface (CPI) is used by computer programs that control a Trainset railroad. The CPI is fully documented and enables researchers and students to write control programs for investigating issues such as real-time computing and fault-tolerance. Programs such as acitest(1) that illustrate the CPI are included in the software distribution.

The other two interfaces are used internally by Trainset. They are: the layout interface, for reading files that describe a railroad and its initial state, as produced by tsed; and the monitor interface, for communication between tsim, tspanel and tsview.

Options

-layout layoutfile

The railroad layout to be simulated is described in the layout data file layoutfile, which must be in the for-

mat produced by tsed. Specify a complete absolute or relative path for layoutfile.

-quorum n
Sets the voting quorum value for this simulation to n.

-simnum number
Number determines which well-known ports are to be used when other programs are initializing communication with this simulation. Permissible values for number are installation-dependent and typically include the range 0 to 100. Multiple instances of tsim may be run simultaneously on the same host if they use different values for number.

-update seconds
Specifies the frequency that tsim is to send state update reports to graphics monitor programs. Seconds may be a decimal value, e.g., 0.5.

-window seconds
Sets the voting window value for this simulation to seconds.

See Also
ts(1), tsed(1), tspanel(1), tsview(1), acitest(1).

Name

tspanel, tsview - Trainset railroad monitor programs

Syntax

tspanel [-simhost hostname] [-simnum number] [-tsroot dir]
tsview [-simhost hostname] [-simnum number] [-tsroot dir]

Description

tspanel is a graphical interface for manually controlling and displaying quantitative state information about a simulation of a Trainset railroad. tsview displays the state of a Trainset railroad, including location of trains and settings of switches. These programs enable a user to establish initial conditions of train motion and switch settings in a railroad and to monitor the progress of a simulation. Each program receives regular reports of the railroad layout's state from the Trainset simulator, tsim(1). See The Trainset Railroad Simulation for more information.

tspanel and tsview are ordinarily invoked using the launch program ts(1).

Options

-simhost hostname
: Specifies the host that is running the simulator tsim.

-simnum number
: number determines which invocation of the tsim simulator (at the indicated host) is to be used. The monitor programs and the desired invocation of tsim should use the same value for number.

-tsroot dir
: (DECwindows implementation only.) Specifies directory for locating uid startup file, e.g., tsroot/tsview.uid

or tsroot/lib/tsview.uid.

Limitations

Currently implemented on DECwindows only.

See Also

ts(1), tsim(1).