**Controller Design**

**CSCI 492**

**WINTER 2011**

****

by

**SCOTT SCHMALZ**

**BEN HJERRILD**

**BRIAN MUSOBO**

(Modifications by Martin Osborne)

Table of Contents

[Introduction 6](#_Toc289423508)

[ER Diagram 7](#_Toc289423509)

[Hardware Components and Context Diagram 10](#_Toc289423510)

[LocoNet 10](#_Toc289423511)

[Throttle – UT4R 11](#_Toc289423512)

[UR92 12](#_Toc289423513)

[DCS-200 13](#_Toc289423514)

[PM42 13](#_Toc289423515)

[Train 14](#_Toc289423516)

[TC64 14](#_Toc289423517)

[Breakout Board (BRK2X5-B-FT) 14](#_Toc289423518)

[Sensor 15](#_Toc289423519)

[DS64 15](#_Toc289423520)

[Switches (Tortoises) 16](#_Toc289423521)

[LocoBuffer 16](#_Toc289423522)

[LocoNet Messages 17](#_Toc289423523)

[Power On Message 17](#_Toc289423524)

[Power Off Message 17](#_Toc289423525)

[Sensor Report Message 17](#_Toc289423526)

[Switch Set Message 18](#_Toc289423527)

[Switch Report Message 18](#_Toc289423528)

[Slot Registration Message 18](#_Toc289423529)

[Speed Set Message 20](#_Toc289423530)

[Direction/Function Set Message 20](#_Toc289423531)

[Sound Set Message 20](#_Toc289423532)

[Functional Requirements 22](#_Toc289423533)

[Train Behavior 22](#_Toc289423534)

[Section Behavior 23](#_Toc289423535)

[Switch Behavior 23](#_Toc289423536)

[Sensor Behavior 23](#_Toc289423537)

[ASCII User Interface 25](#_Toc289423538)

[Start Screen 25](#_Toc289423539)

[Help Screen 26](#_Toc289423540)

[Command Screen 26](#_Toc289423541)

[Debug Screen 28](#_Toc289423542)

[Layout XML 29](#_Toc289423543)

[Tasks and Protected Types 34](#_Toc289423544)

[Controller Architecture Overview 34](#_Toc289423545)

[LocoBuffer Server 36](#_Toc289423546)

[Data Members 36](#_Toc289423547)

[Slot Lookup Table 36](#_Toc289423548)

[Functions, Procedures, and Entries 37](#_Toc289423549)

[Data Members 37](#_Toc289423550)

[Keyboard Read Task 37](#_Toc289423551)

[Data Members 38](#_Toc289423552)

[Screen Write Task 38](#_Toc289423553)

[Data Members 38](#_Toc289423554)

[iToy Read Task 39](#_Toc289423555)

[Data Members 39](#_Toc289423556)

[iToy Write Task 39](#_Toc289423557)

[Data Members 39](#_Toc289423558)

[Command Queue Manager 40](#_Toc289423559)

[Functions, Procedures, and Entries 40](#_Toc289423560)

[Data Members 41](#_Toc289423561)

[Train Task 41](#_Toc289423562)

[Data Members 41](#_Toc289423563)

[Layout Task 43](#_Toc289423564)

[Data Members 43](#_Toc289423565)

[Layout Manager 43](#_Toc289423566)

[Data Members 44](#_Toc289423567)

[Functions, Procedures, and Entries 46](#_Toc289423568)

[File I/O 50](#_Toc289423569)

[Functions, Procedures, and Entries 51](#_Toc289423570)

[Package List 52](#_Toc289423571)

[LocoBuffer Package 52](#_Toc289423572)

[Keyboard/Screen Package 52](#_Toc289423573)

[iToy Package 53](#_Toc289423574)

[Command Queue Package 53](#_Toc289423575)

[Layout Package 53](#_Toc289423576)

[Trains Package 53](#_Toc289423577)

[File I/O Package 54](#_Toc289423578)

[Communicating with the LocoBuffer 55](#_Toc289423579)

[Train States 59](#_Toc289423580)

[Switch States 64](#_Toc289423581)

[Data Flow Diagram 66](#_Toc289423582)

[Internal Messages 67](#_Toc289423583)

[Analyzing Sensor Fire Message 69](#_Toc289423584)

[Start Controller 72](#_Toc289423585)

[Register Train 74](#_Toc289423586)

[Train Speed 76](#_Toc289423587)

[Sensor Fire 78](#_Toc289423588)

[Update Switch from Throttle 81](#_Toc289423589)

[Switch Finish Move 83](#_Toc289423590)

[Save Controller State 85](#_Toc289423591)

[Restore Controller State 87](#_Toc289423592)

[Halt Controller 89](#_Toc289423593)

[Log File 91](#_Toc289423594)

[Acceptance Tests 92](#_Toc289423595)

[Head-On Collision Test 92](#_Toc289423596)

[Trailing Test 92](#_Toc289423597)

[Track Crossing Test 92](#_Toc289423598)

[Switch Test 92](#_Toc289423599)

[Train Direction Test 92](#_Toc289423600)

[Yard Test 93](#_Toc289423601)

[Update Switch Underneath Train Test 93](#_Toc289423602)

[Halting Controller Test 93](#_Toc289423603)

[Save and Restore Test 93](#_Toc289423604)

[Attempt Update Switch in Reserved Section Test 94](#_Toc289423605)

[iThrottle 95](#_Toc289423606)

[Overview 95](#_Toc289423607)

[User Interface 95](#_Toc289423608)

[Top Menu 95](#_Toc289423609)

[Throttle Screen 96](#_Toc289423610)

[Location Screen 96](#_Toc289423611)

[Switches Screen 97](#_Toc289423612)

[Place Train Screen 99](#_Toc289423613)

[Communicating with Controller 100](#_Toc289423614)

[Integration with Controller 100](#_Toc289423615)

[Notes on the iThrottle 100](#_Toc289423616)

[Glossary 101](#_Toc289423617)

[Additional Resources 104](#_Toc289423618)

Introduction

Controller is a real-time railroad system whose purpose is to provide a buffer of safety in order to prevent trains from crashing. A crash can occur when two or more trains collide, or if a train approaches a switch from a direction which will cause derailment. We say Controller is a buffer because the software sits in between train conductors and the physical railroad. This is to say that before a conductor’s commands to the railroad can be executed, they must pass through Controller and be confirmed to be a “safe action”. We’ll define “safe actions” as being commands that will not put a train at risk of crashing.

This document begins by providing you with conceptual information necessary to understand how the train system works. This includes high-level descriptions of how the pieces of the physical train system interact with each other, and how the hardware components work. As you progress through the document, our focus will switch to Controller itself and how to implement the system. By the end of the document, you should understand how the different pieces of the train system and Controller will function. During reading, if you ever come across any terminology that doesn’t make sense to you, refer to the glossary for a definition.

ER Diagram



Figure - ER Diagram

As we can see in Figure 1, there are three main physical components of the physical train system that we will be primarily concerning ourselves with. In this section of the document, we will discuss how these entities relate to each other in regards to Controller. Keep in mind that these relations aren’t implicit to the physical railroad system in itself.

First, we have sections. A section is defined as being a segment of track that is located between exactly two sensors. Sections may overlap with each other. Sections may also contain 0 or more switches. Let’s consider a few example sections represented in Figure 2:

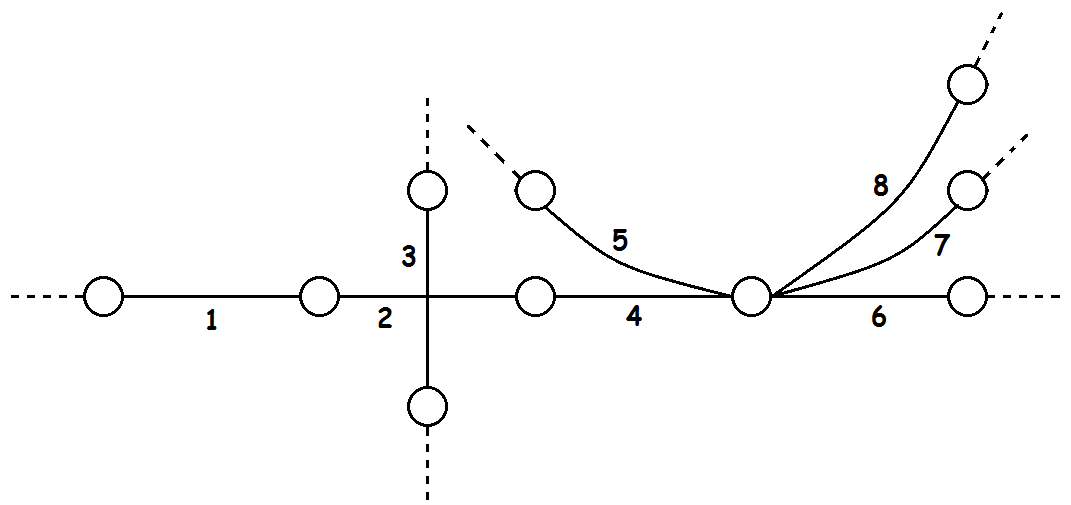


Figure - Example Sections

Section 1 is the most basic type of section. There are no overlapping sections and no switches. Sections 2 and 3 are distinct sections that overlap each other. Sections 4 and 5 contain the same switch. If the switch is closed, then a train will be allowed to travel through section 4 but not 5. Similarly, if a switch is thrown, then a train will be allowed to travel through section 5 but not 4. Note that sections 4 and 5 are considered to be overlapping each other. Sections 6, 7, and 8 contain two switches and also all overlap each other. A train will only be allowed to travel through one of these sections at any given time. The section depends on the state of the switches. Keep in mind that this is only a sample of the types of sections that may exist. It is by no means an exhaustive representation of all the different configurations of sections.

Next, we have the train itself. A train can be “located” by a list of sensors. For example, the train in Figure 3 is located by sensors 5, 4, 3, and 2.

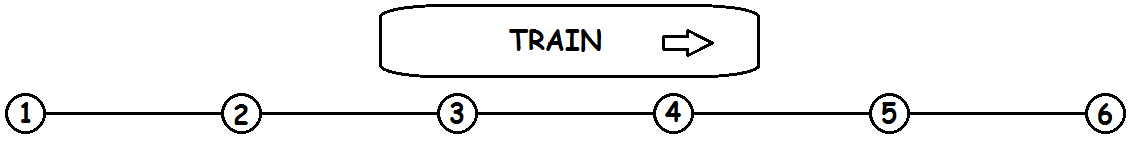


Figure - Locating a Train

Trains can occupy, reserve, or block sections. A train occupies a section when any portion of that train is contained within it. **Under no circumstances should a section ever be occupied by more than 1 train!** A train can reserve up to 1 section. A train should only reserve the next section it will enter in the direction of travel. For example, assuming the train in Figure 3 is traveling in the direction of the arrow then it should have the section located in between sensor 5 and 6 reserved. A train that is not in motion should hold no reservations. No train may occupy or reserve a section that is reserved by another train. A train may block a one or more sections. A section will be blocked if it overlaps with another section that is either occupied or reserved.

In Controller, trains will contain stateful information about themselves. This includes the state

As you may have already inferred, a sensor is the device on the track that separates different sections. A sensor may have one or more sections on either side of it. Sensors are a crucial component of the entire railroad system, and we will discuss them more deeply in the “Hardware Components” section of this document. For the time being, it is sufficient to understand that the sensors are tripped by magnets. One magnet is placed on the front of the train, and a second magnet is placed on the back of the train. This will facilitate the ability for us to keep track of when trains enter and leave sections.

A switch can be thought of as a split in the railroad track. The path that a train will travel through the switch is determined by the state of the switch at that time. Switches have two main states: a closed state and a thrown state. These can be seen in Figure 4. Note that there is also a third state not illustrated in this image. This is the “moving” state. The switch is in this state when it is in the process of changing from the closed state to the thrown state, or vise-versa. Looking at Figure 4, we will refer to the left side of the switch as the “narrow” end, and the right side as the “wide” end. If a train moves toward a switch from the narrow end and the switch is in the moving state, then the train will need to stop moving and wait until the switch enters the closed or thrown state. If a train moves toward a switch from the wide end and the switch is not set in the proper direction, then the train will need to stop moving and wait until the switch is set in the proper state. More information regarding switches can be found in the “Hardware Components and Context Diagram” section of this document.

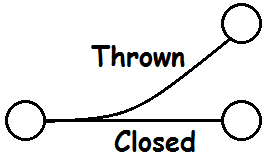


Figure - Switch States

Hardware Components and Context Diagram

Martin April 1: Minor modification

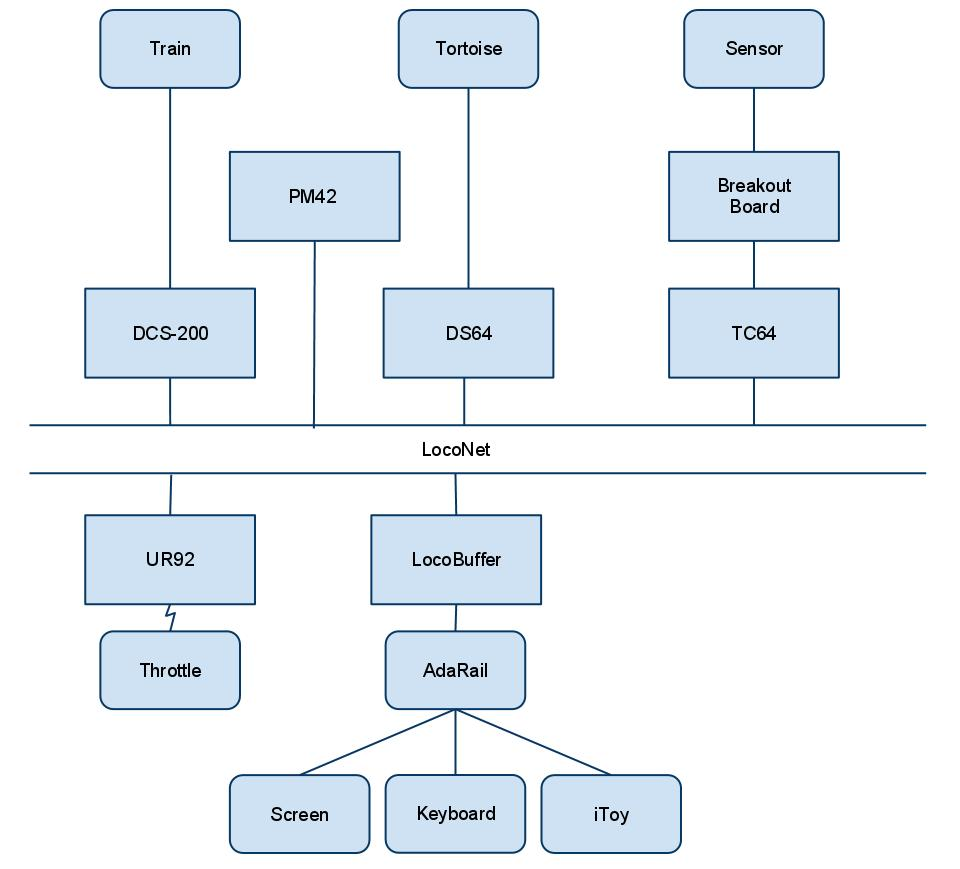


Figure - Context Diagram

## LocoNet

The LocoNet is the railroad system’s internal networking system. LocoNet messages are sent through this network so that the different hardware components can talk to each other. This is the network that Controller will be monitoring to intercept messages sent from a throttle, process them, and then send messages back out to the physical trains. Messages are sent on the LocoNet whenever a physical railroad component changes state, a sensor fires, a command is sent from a throttle, etc.

## Throttle – UT4R



Figure - UT4R Throttle

The UT4R throttle is the hand-held device that the engineers will be using to control the trains. Each throttle has several important components:

* A switch on the top which controls the direction of the train.
* A large knob for controlling the speed.
* Four dials used to select the address of the train we wish to control. For our purposes, we will insert the virtual address provided by Controller here. We will discuss this further later on in this document.
* Seven function buttons, and a shift key.

The directional switch can either be set to forward, backward, or idle. When the switch is set to idle, a message goes out over the LocoNet setting the train’s speed to 0, but no change of direction is given. Flipping to forward or backward will send out a message on the LocoNet informing the train of the directional change.

The speed control knob is used to set the speed to a value between 0-127. Each time the knob is moved, a message goes out over the LocoNet with the updated speed.

The address selection dials can collectively be set to any value between 0-9999. By specifying a virtual address here, all the commands from the throttle will pass through Controller. By specifying a physical train address, all the commands from the throttle will bypass Controller and will go directly to the train itself. We won’t want to do this, because it would cause Controller to lose track of the train. The LED in the middle of the throttle will blink red if the address selected isn’t fully available for communication. Holding down the steal button (F5) will commence communication over the LocoNet with the DCS-200 to take control of the address. Once communication has succeeded, the LED changes to green. Throttles are capable of stealing from each other.

For the puproses of our system, we are using six of the function buttons:

*Martin March 31:*

* F0 – lights on or off.
* F1 – bell on or off.
* F2 – horn on or off.
* F8 – mute on or off.
* F5 – for use with the control program to indicate close next switch in front of train
* F6 – for use with the control program to indicate through next switch in front of train.
* Initially, all the function keys are OFF.
* For function keys other than F2, left click on the key to toggle it between ON and OFF.
* For function key F2, hold the down to turn F2 ON, and release to turn F2 OFF.
* *In Addition, the F2 key has a 'Hold' feature for those who may need F2 to stay on. The 'Hold' feature can be set by holding the F2 key down for 15 seconds. Tapping it again turns it off.*

Within the system, each individual command is sent to the appropriate train and processed as it is received from the throttle instead of being queued up and sent all at once.

## UR92



Figure - UR92

This transceiver takes input from the throttle either through a cable, infrared communication, or radio wave communication and then passes it to the LocoNet.

## DCS-200



Figure - DCS-200

This command station provides power to the PM42’s (described below) which powers the track and controls communication with the trains by sending messages through the tracks themselves.

When a LocoNet message is sent to a train, it first goes through the DCS-200 which converts it into a signal on the track. Before a throttle can communicate with a train, it must be registered with “slot” in the DCS200. This registration process is described in more detail in the LocoNet Messages section of this document. For the time being just understand that when messages are sent to a train, they are actually addressed to a slot number contained in the DCS-200. When the DCS-200 receives these messages, it converts them into the train address and sends them out onto the tracks.

## PM42



Figure - PM42

There are two of these providing power to the layout, each with two subdistricts resulting in a total of 4 power subdistricts in the layout. The PM42s detect short cirucuits in the subdistricts and manage the polarity of the sections. For the purposes of this project, it’s important to understand what the PM42s do, but our software won’t be directly interfacing with them.

## Train



Figure - Train Engine

These are, of course, the physical trains on the track. Each consists of a locomotive with zero or more attached cars. For this system, each train will have a magnet attached to its undercarriage at the front and the back of the train, allowing us to track its position as it fires sensors. As described above, we communicate with the trains through the DCS-200 which converts LocoNet messages into messages the train’s decoder can read off of the track and process. Trains provide no acknowledgement that a command has been received, at least not one that propagates onto the LocoNet.

## TC64



Figure - TC64

All sensors are connected to a TC64 through a BRK2X5-B-FT. Since up to eight sensors can connect to a BRK2X5-B-FT and the TC64 has 8 ports, a total of 64 sensors can be associated with a single TC64. For our purposes, the TC64 simply takes the very simple signals form the sensors and converts them into general sensor reports on the LocoNet.

See [1] for more information regarding the TC64s.

## Breakout Board (BRK2X5-B-FT)



Figure - Breakout Board

The Breakout Boards are the connection point between the sensors themselves and the TC64s.

## Sensor



Figure - Sensor

The most straightforward element of the entire system, sensors do one thing and one thing only: they send signals when a magnet passes them. Groups of eight or fewer sensors are all connected to a BRK2X5-B-FT which simply takes all of their inputs and puts them onto a single ribbon cable which it then patches into one of the eight ports of a TC64. See the LocoNet Messages section of this document for more information on the LocoNet sensor messages the TC64 sends.

## DS64



Figure - DS64

Similar to the TC64, the DS64 provides an interface between the physical switches (Tortoises) and the LocoNet. Each DS64 can be associated with up to for Tortoise switches. In several cases, there are in fact five switches attached to the DS64, but two of them are connected to the same physical input/output ports on the DS64, resulting in them both being seen as a single logical switch.

See [2] for more information regarding the TC64s.

## Switches (Tortoises)

|  |  |
| --- | --- |
| Figure - Switch | Figure - Tortoise |

Switches allow two segments of track to diverge or converge (the difference being a purely semantic one, depending on the direction you approach the junction from). Each switch has two possible positions: closed and thrown. In the close position, a train approaching the narrow end will go straight along the main track; in the thrown position, a train approaching from the narrow end will veer off onto the diverging track

Each switch is connected to a DS64 which converts messages to/from the switch into/from LocoNet messages. The term “tortoise” refers to the mechanical motor that actually moves the switches.

See [3] for more information regarding the Tortoises.

## LocoBuffer



Figure - LocoBuffer

This adapter converts from the RJ-12 LocoNet cables to a USB connection which is used to connect to the computer. The LocoBuffer also takes care of certain low-level LocoNet details such as collision detection on the medium and subsequent backoffs. By using LocoBuffer, we can simply read data off the LocoNet and put data on the LocoNet without worrying about these things.

See [4] for more information regarding the LocoBuffer.

LocoNet Messages

The single best source of information for the LocoNet is located in [5]. While the beginning sections are interested, the relevant information really begins on page 5 with the “MESSAGE Format” section.

LocoNet messages consist of binary multi-byte message. The first byte of every message is an OPCODE which will let us know the length of the message and if subsequent response message is required. The length of the message can be 2, 4, 6, or N bytes. An OPCODE byte will always have its most significant bit set to 1. All non-OPCODE bytes will have their most significant bits set to 0. The last byte in a message is a CHECKSUM byte. The CHECKSUM is calculated by taking the 1’s Complement of the byte wise Exclusive Or of all the bytes in the message, except the checksum itself.

Note that in several places throughout documentation, the phrase CONSISTing or CONSISTed locomotives is used. This is referring to when several locomotives are somehow linked together. It is not applicable to this project, and we will not be dealing with this feature.

We will now explain the LocoNet messages that are of particular interest to us.

Martin April 1: The check sum byte is computed as the xor of the preceding bytes in the message and is then complemented.

## Power On Message

For more information on this message, refer to [5], page 8.

**OPC\_GPON**

**GLOBAL POWER on message: <0x83><CHK>**

## Power Off Message

For more information on this message, refer to [5], page 8.

**OPC\_GPOFF**

**GLOBAL POWER off message: <0x82><CHK>**

## Sensor Report Message

This is the LocoNet message that is generated by a TC64 when a sensor is fired. It is important to realize that the message does not contain any stateful information, like which train tripped the sensor. It simply reports that a particular numbered sensor was tripeed. It is the job of Controller to determine *who* tripped it. For more information on this message, refer to [5], page 9. The format of the message is as follows:

**OPC\_INPUT\_REP**

**SENSOR report message: <0xB2><SN1><SN2><CHK>**

SN1 <0,A6,A5,A4,A3,A2,A1,A0>

SN2 <0,X,I,L,A10,A9,A8,A7>

I = 1 => add 1000 to the address

L = 1 => sensor high else low

CHK checksum (xor preceding bytes and then negate)

## Switch Set Message

This is the LocoNet message to throw/close a particular switch. For more information on this message, refer to [5], page 9. The format of the message is as follows:

**OPC\_SW\_REQ**

**TURNOUT set message: <0xB0><SW1><SW2><CHK>**

SW1 <0,A6,A5,A4,A3,A2,A1,A0>

SW2 <0,0,DIR,ON,A10,A9,A8,A7>

DIR = 1(close switch)/0(throw switch)

ON = 1(output on)/0(output off) <-- set this to 1

switch number = 128(SW2 and 0x0F) + SW1 + 1 (Martin April 1)

CHK checksum (xor preceding bytes and then negate)

## Switch Report Message

This is the LocoNet message that is sent when a switch has finished moving. For more information on this message, refer to [5], page 9. The format of the message is as follows:

**OPC\_SW\_REP**

**TURNOUT report message: <0xB1><SW1><SW2><CHK>**

SW1 <0,A6,A5,A4,A3,A2,A1,A0>

SW2 <0,1,I,L,A10,A9,A8,A7>

L = 1 => switch closed else thrown

CHK checksum (xor preceding bytes and then negate)

## Slot Registration Message

Martin April 1: See the online help in my railroad simulator for more details.

This is by far the most complex LocoNet message we will need to deal with in Controller. As briefly described in the “Hardware Components and Context Diagram” section of this document, the DCS-200 translates messages between addresses and slot numbers. In our system, whenever a train is registered, we will be dealing with two addresses: the physical train address and the virtual address. The virtual address will be generated by Controller. This is the address that the user will dial into their throttle. The physical train address is the address of the train the user wishes to control with the throttle. By convention, the physical train address is usually the number pained on the side of the train. When a new train is registered with Controller, we must get two slot numbers from the DCS-200. We must get a virtual slot number which will correspond to the virtual address. We will also need to get a physical train slot number which will correspond to the physical train address. Once these slot numbers have established, the messages on the LocoNet will be addressed to slot numbers, not virtual addresses or physical train addresses. Controller will receive message from the throttle on the LocoNet that are addressed to the virtual slot number. Similarly, when Controller sends messages to the physical train, it will send the messages out on the LocoNet addressed to the physical train slot number.

First, we will describe the sequence of messages that are exchanged in the transaction to register a slot number. Afterwards, we will describe each message individually.

**Sequence:**

1. Controller sends OPC\_LOCO\_ADR (0xBF)
   1. If there are sufficient slots, DCS-200 responds with OPC\_SL\_RD\_DATA (0xE7)
   2. If no slots are available, DCS-200 responds with OPC\_LONG\_ACK (0xB4) fail code 0, registration fails due to insufficient available slots (Martin April 1: This should never happen to us.)
2. Based upon the response from step 1
   1. If slot is COMMON, IDLE, or NEW then Controller sends OPC\_MOVE\_SLOTS (0xBA) – NULL move
   2. Otherwise, the slot is IN\_USE or UP\_CONSISTED. The registration process fails because the address is already in use.

**Message Descriptions:**

**OPC\_LOCO\_ADR**

**<0xBF><0><ADR><CHK>**

ADR <0,A6,A5,A4,A3,A2,A1,A0>

CHK checksum (xor preceding bytes and then negate)

ADR is the address you wish to register a slot to. For our purposes, they may be a physical train address or a virtual address. For more information on this message, refer to [5], page 8.

**OPC\_SL\_RD\_DATA**

**<0xE7><0E><SLOT#><STAT><ADR><SPD><DIRF><TRK><SS2><ADR2><SND>**

**<ID1><ID2><CHK>**

SLOT#, SPD, and ADR are exactly what you’d expect. The DIRF bytes is the same that is found in the Direction/Function Set Message description. For the purposes of slot registration, the only byte in this message which is of interest is the STAT byte. The only bits of this byte that are of interest to us are the 2nd, 3rd, and 4th most significant bits. If the 2nd most significant bit is 1, then the slot is up-consisted and cannot be acquired. In this case, registration should fail. If the 3rd and 4th most significant bits are both set to 1, then registration should also fail. Otherwise, the registration can proceed to performing the NULL move. For more information on this message, refer to [5], page 10.

**OPC\_LONG\_ACK**

**<0xB4><LOPC><ACK1><CHK>**

This is a generic acknowledgement message. The LOPC byte is generally set tot the OPCODE of the message that the acknowledgement is replying to. The ACK1 byte is an appropriate response code. For our circumstances, when this is used to indicate that there are no available slots, the LOPC byte is set to 0. For more information on this message, refer to [5], page 9.

**OPC\_MOVE\_SLOTS**

**<0xBA><SRC><DEST><CHK>**

The SRC byte is the source slot that you wish to move into DEST. For the purposes of registration, we will only be using the NULL move functionality message which means we set both the source and destination to the same slot, namely the one we are setting to IN\_USE. For more information on this message, refer to [5], page 8.

## Speed Set Message

This is the LocoNet message that is sent when a we wish to change the speed of a train. These messages are sent by the throttle when an engineer movies the speed knob or breaks. Controller will process these messages coming from the throttle. Controller will send these messages to the physical train, assuming it doesn’t pose a threat. For more information on this message, refer to [5], page 10. The format of the message is as follows:

**OPC\_LOCO\_SPD**

**SPEED set message: <0xA0><SLOT#><SPD><CHK>**

SLOT# 0..127 the slot number associated with the address

SPD 0..127

CHK checksum (xor preceding bytes and then negate)

## Direction/Function Set Message

This is the LocoNet message that is sent when we wish to change the direction of a train or perform a function like toggling the lights or generating a sound. Controller will process these messages from the throttle. Controller will send these messages to a physical train if they pose no threat. For more information on this message, refer to [5], page 10. The format of the message is as follows:

**OPC\_LOCO\_DIRF**

**DIRECTION/FUNCTION set message: <0xA1><SLOT#><DIR\_STATE><CHK>**

SLOT# 0..127 the slot number associated with the address

DIR\_STATE <D7,D6,D5,D4,D3,D2,D1,D0>

D7 : 0, always

D6 : 0, ignored

D5 : 0 request direction forward; 1 request direction

backward

D4 : toggles between 0 and 1 when F0 pressed; initially 0

D3 : toggles between 0 and 1 when F4 pressed; initially 0

D2 : toggles between 0 and 1 when F3 pressed; initially 0

D1 : 1 when F2 pressed; 0 when F2 released

D0 : toggles between 0 and 1 when F1 pressed; initially 0

CHK checksum (xor preceding bytes and then negate)

## Sound Set Message

This is the LocoNet message that is generated when the engineer presses a function key on the throttle. Controller should interpret the meaning of the function key and forward the appropriate messages to a physical train or switch. For more information on this message, refer to [5], page 10. The format of the message is as follows:

**OPC\_LOCO\_SND**

**SOUND set message: <0xA1><SLOT#><SND>CHK>**

SLOT# 1..127 the slot number associated with the address

DIR\_STATE <D7,D6,D5,D4,D3,D2,D1,D0>

D7-D4 : 0, reserved for future use

D3 : toggles between 0 and 1 when F8 pressed; initially 0

D2 : toggles between 0 and 1 when F7 pressed; initially 0

D1 : toggles between 0 and 1 when F6 pressed; initially 0

D0 : toggles between 0 and 1 when F5 pressed; initially 0

CHK checksum (xor preceding bytes and then negate)

Functional Requirements

The entire system is started by running the ASCII User Interface application on the PC connected to the LocoNet by the LocoBuffer. This application is written in Ada with a C component used to control access to the LocoBuffer.

The system receives all messages sent over the LocoNet (via the LocoBuffer). The engineers trying to drive the trains will be using the UT4R throttles, an iThrottle, or some other device like a web browser. For a description of the UT4R throttle, refer to the “Hardware Components and Context Diagram” section of this document. For a description of the iThrottle, refer to the “iThrottle” section of this document. Any device that an engineer uses (throttle, etc) will be set to virtual LocoNet addresses. We call them virtual addresses because they refer to a LocoNet address which is no actually associated with a train. Controller receives messages sent to these virtual addresses, determine if they need modification, and then forwards them on to the actual trains if they are permissible. The system maintains an internal data representation of the tray layout in memory. This allows Controller to check which actions are safe and which are not.

## Train Behavior

* A train can only move if it has reserved the next section in the direction of travel.
* As soon as a train crosses a sensor in the direction of travel, it must either reserve the next section or wait.
* If a train wants to move, but can’t get a reservation for the next section in the direction of travel, then it stop and wait until it can get the reservation.
* If a moving train wishes to change direction, it must request a reservation on the next section in the new direction of travel. If the reservation cannot be made, then the train must stop and wait until it can get the reservation.
* A train whose speed has voluntarily been set to zero by the engineer is said to be in a halted state.
* A halted or waiting train may change direction.
* A train gives up any reservations it may have once it enters a halted state.
* A train responds to a non-zero speed command only when it’s in a moving or halted state.
* The train remains in the halted state the entire time its speed is set to zero.
* If the speed of a train is non-zero and the next section is free, then the train may attempt to reserve that section and enter a moving state if the reservation is successful.
* If the speed of a train is non-zero and the train does not have a reservation on the next section, then the train must go into a waiting state.
* A train should never try to cross a switch that is in motion.
* The engineer may change the setting of the next switch along the path he’s currently traveling in. The next switch must be beyond any sections that the train is occupying or has reserved.
* When a train enters a siding, it should park itself automatically. This means it should slow down, stop when it reaches the last sensor, change direction, and move slightly forward to clear the sensor.
* A train is in the “begin halt” state when the engineer has set the train’s speed to zero, but the physical train hasn’t quite come to a stop yet.
* A train is in the “begin wait” state when the train is getting ready to enter the wait state, but the physical train hasn’t quite come to a stop yet.

## Section Behavior

* A section is surrounded by exactly two sensors.
* A train can only reserve the next section in the direction of travel.
* A section can be reserved by at most one train.
* When a section is occupied or reserved, all overlapping sections are considered to be blocked.
* When a section is set to a free state, all overlapping sections are considered to be free as well unless they overlap with a third section that is set in the occupied or reserved state.
* A request to reserve a section succeeds only if the section is free.
* If a train requests to reserve a blocked section, that train will enter a waiting state while it waits for the section to become free.

## Switch Behavior

* A switch can be in either a closed, thrown, or moving state at any given time.
* A request to close a switch that is already closed with be ignored. Same for thrown switches as well.
* If a train approaches a switch from the wide end and the switch is not set in the proper direction to allow the train through, the train must wait until the switch is aligned properly before it can get the reservation and continue traveling.
* There can be zero or more switches in each section. Sections can have more than one switch.
* A train approaching a switch from the narrow end does not have any restrictions on the orientation of the switch. As long as the switch is not in a moving state and the train has successfully obtained a reservation for the section, it should be able to proceed into the section.

## Sensor Behavior

* A sensor has one or more sections on either side of it. A sensor may have multiple sections located on both sides of it. The only exceptions to this rule are sidings where one side of the sensor contains zero sections.
* The magnet located on the bottom of a train may stop directly on top of a sensor.

ASCII User Interface

*Martin March 31: See AdminThrottle.pptx*

Layout XML

This XML file is a representation of the topology of the track layout that the trains will be running on. The XML captures information about sections, sensors, and switches, and how all of these components relate to each other.



Figure - Sample Layout

Figure 22 is an example track layout. We have tried to make this layout inclusive of all the different types of situation that can exist, but it is by no means exhaustive. Pay particular attention to the sections contained between sesnors 23, 24, 25, 26. This is an example of a train yard. Note that these sections contain more than one switch. Also pay close attention to the sections contained between sensors 5, 6, 15, and 16. This is a “crossover”. Note that the crossover consists of two switches, but both of these switches have the same ID. This means that when one of these switches is thrown/closed, the other will be thrown/closed as well. The XML that encapsulates the topology of this layout. See Controller.pptx, and read the notes section for slide “XML File Structure” (Martin April 1).

Tasks and Protected Types

Martin April 1: the diagrams can be found in Controller.pptx

In this section of the document, we will provide a quick high-level overview of the architecture of the system. We will then move into describing each task and protected type that is included in Controller. The data that should be stored in each task and protected type will also be specified. We will cover the interfaces into each protected type. Since the tasks communicate via messages, we will reserve that discussion for the “Internal Messages” section later in this document.

## Controller Architecture Overview

Controller consists of a number of tasks and protected types. In Ada, you can think of a task as a thread of execution. Protected types are essentially classes/objects which can only be accessed by on thread at a time. It is very important for us to use protected types because Controller will have many threads running concurrently that will want to update data in the system. Protected types prevent multiple tasks from updating data at the same time, thus preventing corruption.

Protected types can be interfaced with directly via function, procedure, and entry calls. In Controller, tasks can only be interfaced with via messages. Almost every task in Controller has a message queue associated with it. A task reads messages off of its queue and processes it accordingly. Tasks and protected types may send messages to queues.

xxx (updated by Scott 04/01) Communication to the LocoBuffer is facilitated through what we call the LocoBuffer Server. This is a component of Controller that isn’t tightly coupled with any other component of Controller. It’s only responsibility is to translate messages to and from the LocoBuffer. The LocoBuffer Server will take LocoNet messages off of the LocoBuffer, translate them into internal messages, and then send them to the clients that are currently connected to the server. The clients will be instances of Controller that are running. Generally speaking, we will only have one client running, but it is possible to add more if necessary. Controller communicates with the LocoBuffer Server via a TCP/IP connection. Controller will contain a socket that references this connection with the LocoBuffer Server. It can read/write messages to and from this socket in order to read/write messages to and from the LocoBuffer.

(updated by Scott 04/01) The messages that are passed between tasks and the LocoBuffer Server are managed by the Command Queue Manager. We will go into more detail about the Command Queue Manager below. For the time being it will suffice to understand that the Command Queue Manager routes messages to the appropriate queues, or to the LocoBuffer Server via a socket.

Each train that is registered with Controller has a corresponding Train Task. Contrary to what you might think, the Train Tasks are actually rather dumb. They don’t do much of the heavy lifting of Controller. They essentially store state information about each train like speed, direction, etc.

The Layout Task is responsible for processing any messages coming in from the LocoNet that the Train Tasks shouldn’t worry about. These types of messages include switch messages, register train messages, etc. We call it the Layout Task because most of these messages will affect the current state of the track layout.

The Layout Manger is a protected type that contains the internal data representation of the physical track layout. The idea is that Controller’s internal data representation of the track should be in sync with the physical state of the track at all times. This means the internal data representation of the track must always know the location of trains, states of the switches, etc.

(updated by Scott 04/01) The Slot Lookup Table is a protected type that keeps track of train slot numbers and their corresponding addresses. The LocoBuffer Server will rely heavily on the Slot Lookup Table. It needs to translate messages coming off the LocoNet from being address to slot numbers to being address to virtual addresses. It also needs to translate messages going out on the LocoNet from Controller from the virtual address to the physical train slot number.

File I/O is a protected type that does exactly what you’d imagine – writes to files located on the local computer’s filesystem. This protected type is utilized when Controller wishes to write to a log file for debugging purposes, or when Controller wishes to save the current state of the system into an XML file.

## LocoBuffer Server

(updated by Scott 04/01) The LocoBuffer Server is responsible for communication to and from the LocoBuffer. It is responsible for taking messages off of the LocoNet, translating them into internal messages, and then sending them to the Controller clients that are connected to the server. Refer to the section of this document titled “LocoNet Messages” to see how LocoNet messages are formatted. The LocoBuffer Server is also responsible for taking messages from the clients, converting them into LocoNet messages, and sending them out onto the LocoNet. The LocoBuffer Server needs to communicate with the Slot Lookup Table in order to translate messages from slot numbers to virtual addresses. The messages we’re interested in grabbing off of the LocoNet are messages that are addressed to a slot number that corresponds to a virtual address of a train that is registered with Controller. The messages we’re interested in sending onto the LocoNet are messages that are addressed to a slot number that corresponds to a physical address of a train. So, when the LocoNet messages are translated into internal messages, the addresses must be translated from virtual slot numbers to virtual address. When internal messages are translated into LocoNet messages, the addresses must be translated from virtual addresses to physical train slot numbers.

The LocoBuffer Server talks to the Command Queue Manager across a TCP/IP connection. It will have a socket for each client it needs to talk with. We’ll discuss the Command Queue Manager in more detail below.

### Data Members

**SlotLookupPtr**

SlotLookupPtr is a pointer to the Slot Lookup Table.

**TCP/IP data**

Data that corresponds to the TCP/IP connection between the LocoBuffer Server and the clients.

## Slot Lookup Table

The Slot Lookup Table is responsible for maintaining information about the relationships between virtual address, virtual slot numbers, physical train slot numbers, and physical train addresses. It will essentially contain a single table of records that does this. The index into the table will represent the virtual address. Each record will then contain the corresponding virtual slot number, physical train address, and physical train slot number.

### Functions, Procedures, and Entries

**requestVirtualAddress() : VirtTrainAddr**

This procedure call requests the next available virtual address in the system. It does this by iterating through the lookup table and finding the first empty entry. The index of that first empty entry is returned as an out parameter. This represents the virtual train address.

**createEntry(VirtTrainAddr, VirtSlot#, PhysTrainAddr, PhysSlot#)**

This procedure call will create a new record in the lookup table. The parameters include all the information necessary to create a record in the table. The VirtTrainAddr parameter will be the index in the table that the record should be created at. Note that this VirtTrainAddr value should be the same value that was returned by a previous call to the requestVirtualAddress() entry call.

**getVirtualAddress(PhysSlot#) : VirtTrainAddr**

This procedure will return as an out parameter the virtual address that is associated with a given physical train slot number. This is the procedure that the LocoBuffer Read Task will use to translate the addresses of messages coming off the LocoNet.

**getPhysSlot#(VirtTrainAddr) : PhysSlot#**

This procedure will return as an out parameter the physical train slot number that is associated with a given virtual address. This is the procedure that the LocoBuffer Write Task will use to translate the addresses of messages leaving Controller to the LocoNet into the properly physical train slot number.

### Data Members

**LookupEntry(VertSlot#, PhysTrainAddr, PhySlot#)**

This is not an instance variable. It is an Ada record type. Instances of these records will be stored in the lookup table, which is described below. Instances of this record will contain a virtual slot number, a physical train address, and a physical slot number.

**LookupTable**

LookupTable will contain LookupEntries. Each entry in the LookupTable represents an association between a virtual address, virtual slot number, physical train address, and physical train slot number. The indexes into the table represent virtual addresses. The rest of the data is stored in the LookupEntry itself.

## Keyboard Read Task

The Keyboard Read Task is responsible for interpreting commands given by a user at the UI. For a full description of the types of commands a user may enter, refer to the section of this document titled “ASCII User Interface”. When the user enters a command, the Keyboard Read Task translates that command into an internal message and forwards it to the Command Queue Manager who will then place it on the appropriate queues.

It is also important to know that the Keyboard Read Task is one of the few tasks in Controller that do not have a queue associated with it. Its only job is to interpret messages from the keyboard, translate them into the proper internal message format, and then forward them to the Command Queue Manager.

### Data Members

**CmdQueueMgrPtr**

CmdQueueMgrPtr is a pointer to the Command Queue Manager.

## Screen Write Task

The Screen Write Task is responsible for displaying real-time information about Controller to the user’s computer screen. This task will read from its queue and display the information to the user’s screen accordingly.

There is one strange thing about the Screen Write Task that isn’t apparent. The Screen Write Task will keep a cache current state information about the system. This includes information like the speed of trains, the state of all the sections, etc. The reason we do this is because users may swap between different screens in the UI. We need to be able to quickly update the screen with current information when a user swaps to a new screen. This may seem completely redundant, and in a lot of ways it is. However, in an alternative approach we would have to ask each Train Task and the Layout Manager to report back with all of their information whenever a user swaps between screens in the UI. This would result in a fair amount of additional complexity as well as a ton of internal messages being generated in a short span of time. Keep in mind that the Screen Write Task only needs to know the information it has to display to the screen. This means that it doesn’t have to know how sections and sensors relate to each other in the way that Layout Manager needs to. Whenever the Screen Write Task receives a message, it updates its cache of data and then updates the screen.

### Data Members

**QueuePtr**

QueuePtr is a pointer to the LocoBuffer Read Task’s queue. The queue itself is stored in the Command Queue Manager which is why we only have a pointer to the queue here.

**Train(VirtTrainAddr, PhysTrainAddr, State, Speed, Direction, Sensors)**

This is not an instance variable. It is an Ada record type. Instances of these records will be stored in the TrainList as described below. Each instance of this record will represent a single train. Instances of this record will contain the train’s virtual address, physical address, current state, speed, direction, and a list of sensors that locate the train.

**TrainList**

TrainList will contain values of the Train record type as described above.

**Switch(SwitchID, State)**

This is not an instance variable. It is an Ada record type. Instances of these records will be stored in the SwitchList as decribed below. Each instance of this record will represent a single switch on the layout. Instances of this record will contain the switch’s ID and the current state of the train.

**SwitchList**

SwitchList will contain values of the Switch record type as described above.

## iToy Read Task

The iToy Read Task is responsible for interpreting data coming from iThrottles, translating that data into internal messages, and then sending those messages to the Command Queue Manager. The iToy Read Task will not need to worry about translating address like the LocoBuffer Read Task. This is because the iToy data uses the same virtual addresses that Controller uses.

It is also important to know that the iToy Read Task is one of the few tasks in Controller that do not have a queue associated with it. Its only job is to interpret messages coming from iThrottles, translating those messages, and then forwarding them to the Command Queue Manager.

### Data Members

**CmdQueueMgrPtr**

CmdQueueMgrPtr is a pointer to the Command Queue Manager.

**TCP/IP Data**

The iToyReadTask will need to keep track of whatever data is necessary to talk to each iThrottle via TCP/IP.

## iToy Write Task

The iToy Write Task is responsible for pushing real-time information about the current state of the train system to the iThrottles. Information about a specific train, such as speed, should only be pushed to the iThrottle that is controlling that train. Other information, like the states of switches, should be pushed to all the iThrottles.

Similar to the Screen Write Task, the iToy Write Task will keep a cache of current state information. We do this for the exact same reasons mentioned in our description of the Screen Write Task.

### Data Members

**QueuePtr**

QueuePtr is a pointer to the LocoBuffer Read Task’s queue. The queue itself is stored in the Command Queue Manager which is why we only have a pointer to the queue here.

**TCP/IP Data**

The iToyReadTask will need to keep track of whatever data is necessary to talk to each iThrottle via TCP/IP.

**Train(VirtTrainAddr, PhysTrainAddr, State, Speed, Direction, Sensors)**

This is not an instance variable. It is an Ada record type. Instances of these records will be stored in the TrainList as described below. Each instance of this record will represent a single train. Instances of this record will contain the train’s virtual address, physical address, current state, speed, direction, and a list of sensors that locate the train.

**TrainList**

TrainList will contain values of the Train record type as described above.

**Switch(SwitchID, State)**

This is not an instance variable. It is an Ada record type. Instances of these records will be stored in the SwitchList as decribed below. Each instance of this record will represent a single switch on the layout. Instances of this record will contain the switch’s ID and the current state of the train.

**SwitchList**

SwitchList will contain values of the Switch record type as described above.

## Command Queue Manager

The Command Queue Manager is a protected type that is responsible for placing messages on their appropriate queues. Tasks throughout Controller will give the Command Queue Manager messages. It is the job of the Command Queue Manager to place these messages on the appropriate queues. It determines which queues to place the message on based off of what type of message it is.

The Command Queue Manager contains all of the queues associated with all the tasks. The tasks themselves contain pointers to the queues. This makes things much cleaner because all the queues are located in one located as opposed to being dispersed throughout Controller.

Each queue will be of the same type, but we break them up into different variables. For example, we have a list of train queues, a ScreenWriteQueue, etc. The reason behind this is that it will allow the Command Queue Manager to decide which queues to place certain messages on.

(updated by Scott 04/01) The Command Queue Manager is also the task that is responsible for communicating with the LocoBuffer Server. Whenever the Command Queue Mananger receives a message that should be sent out on the LocoNet, it sends that messages to the LocoBuffer Server via a socket. Whenever the Command Queue Manager receives a message from the LocoBuffer Server via the socket, it places the messages on the appropriate queues.

### Functions, Procedures, and Entries

**registerQueue(Type, VirtTrainAddr) : QueuePtr**

This function will create a new queue and pass back a pointer to that queue as a return value. The Type parameter will be a value of an enumerated type. This parameter will let the Command Queue Manager know what type of queue to create; a train queue, LayoutQueue, LocoBufferWriteQueue, etc. Every queue except the train queues should only be created once.

**sendMessage(Message)**

This procedure will place the message passed in as a parameter on the appropriate queues. It looks at what type the message is, and places it on one or more queues.

### Data Members

**TrainQueues**

A list of train queues. Each queue in the list corresponds to a single Train Task. The queue at location should correspond to the TrainTask whose virtual address is .

**LayoutQueue**

The queue associated with the Layout Task.

**LocoBufferWriteQueue**

The queue associated with the LocoBuffer Write Task.

**ScreenWriteQueue**

The queue associated with the Screen Write Task.

**iToyWriteQueue**

The queue associated with the iToy Write Task.

## Train Task

Each Train Task corresponds to a single train registered with Controller. The tasks contain state information about their corresponding trains. Train Tasks do not do much of the “heavy lifting” in Controller. They do not contain much of the deep logic. They are essentially placeholders to keep track of the current state of each train.

The first thing a Train Task should do when created is call the registerQueue() function located in the Command Queue Manager. This will establish a message queue for the Train Task.

Quite a few of the messages coming into Controller will be routed to the Train Task. Some of these messages include speed messages, direction messages, etc. Refer to the section of this document titled “Internal Messages” for a complete list of information about all the internal messages of Controller.

### Data Members

**QueuePtr**

QueuePtr is a pointer to the LocoBuffer Read Task’s queue. The queue itself is stored in the Command Queue Manager which is why we only have a pointer to the queue here.

**LayoutPtr**

LayoutPtr is a pointer to the Layout Manager. This allows the Layout Task to make function, procedure, and entry calls into the Layout Manager.

**FileIOPtr**

FileIOPtr is a pointer to the File I/O protected type. We will utilize this pointer when we need to write an XML file when saving the state of Controller. Refer to the section of this document titled “Save Controller State” for more information on this.

**VirtTrainAddr**

This represents the virtual address of the train.

**State**

This represents the current state of the train. The states include moving, halted, waiting, begin halted, and begin waiting. A train is in the moving state when its speed is set to a non-zero value and it has a reservation on the next section in the direction of movement. A train is halted when it not moving and doesn’t wish to be moving. The physical train should no longer be moving to be in a halted state. A train is in the begin halted state when its speed is set to zero, but the physical train hasn’t quite come to a full stop yet. A train is waiting when it is not moving, but wishes to be moving. A train enters the waiting state when it cannot get a reservation on the next section it would like to enter. This can occur include if another train is occupying the section or has it reserved. This can also occur if a switch in that section is set in a direction that would cause the train to derail. A train is in the begin waiting state when it’s entering the waiting state, but the physical train hasn’t quite come to a full stop yet. A more concise description of the train states can be found in the section of this document titled “Train States”.

**Direction**

This represents the current direction of a train. The train can be set in either the forward or backward direction, even if the train is not currently in motion.

**Speed**

This represents the current speed of the train. A train’s speed value ranges from 0 to 127. A train’s speed will be set to 0 when it is in the halted or begin halted states. If a train is in the waiting state, it’s speed will not be set to 0. This is because when a train gets the reservation it’s waiting for, the train will need to resume travel at the same speed it was in before it entered the waiting state.

**Front Sensor ID**

This is the ID of the sensor located directly in front of the train in the direction of motion. This variable is useful when a train wishes to move again after being halted or waiting. It allows us to be able to figure out what direction is forward, and thus which section to attempt to make a reservation into. Referring to Figure 24, we see that the Front Sensor ID of the train is 5.

**Back Sensor ID**

This data member is very similar to Front Sensor ID except it represents the sensor immediately behind the train. Another way to say this is it represents the last sensor that the back of the train tripped. Referring to Figure 24, we see that the Back Sensor ID of the train is 17.

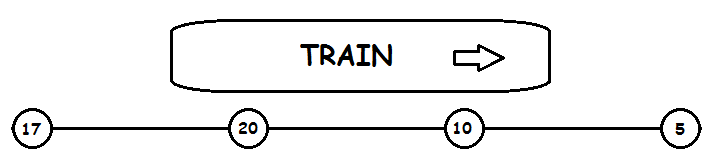


Figure - Front Sensor ID and Back Sensor ID

## Layout Task

The Layout Task is responsible for dealing with messages entering Controller that the Train Task can’t deal with. These types of messages include switch messages, sensor messages, etc. We called this the Layout Task because most of these messages deal involve the changes to the current state of the layout. Also, the Layout Task almost always makes function calls into the Layout Manager when processing these messages.

### Data Members

**QueuePtr**

QueuePtr is a pointer to the LocoBuffer Read Task’s queue. The queue itself is stored in the Command Queue Manager which is why we only have a pointer to the queue here.

**LayoutPtr**

LayoutPtr is a pointer to the Layout Manager. This allows the Layout Task to make function, procedure, and entry calls into the Layout Manager.

**SlotLookupPtr**

SlotLookupPtr is a pointer to the Slot Lookup Table. The Layout Task contains this pointer because it will be the Layout Task that creates an entry in the Slot Lookup Table once a train has been registered with the LocoNet, the train task spawned, and the appropriate data updated on the Layout Manager. Refer to the “Register Train” section of this document for a complete description of what happens when a train is registered.

**Temporary Sensor List**

This variable is needed when a train is being registered with the system. It will hold the list of sensors that initially locates a new train until the virtual address of the train has been established. Refer to the “Register Train” section of this document for a complete description of what happens when a train is registered.

## Layout Manager

The Layout Manager is the protected type responsible for maintaining the internal data representation of the track layout. The Layout Manager contains most of the deep logic of Controller. The most difficult thing the Layout Manager needs to do is figure out which train set of a sensor when it is fired and whether it was the front or the back of the train that set that sensor off. This can be difficult because when a sensor is fired, there is not stateful information contained in the message. The only thing we’re told is the ID of the sensor that was tripped.

The Layout Manager contains three main types of records. There are section records, sensor records, and switch records. These records are described in their entirety below. The Layout Manager will contain an array of section records, an array of sensor records, and an array of switch records. Each section record that exists represents a single section on the track layout. Similarly, each sensor record that exists represents a single sensor on the track layout. Also, each switch record that exists represents a single switch on the track layout. Relations are made between these three types of entities via pointers. The IDs of each section, sensor, and switch should correspond to its index into the array. This will set us up for constant time access when we wish to refer to a specific section, sensor, or switch in the future. Keep in mind that the IDs may not be continuous values – there may be some gaps. The gaps in the arrays should be set to NULL.

The internal data representation of the layout is initially set up when Controller starts and the user enters the Layout XML filename. Refer to the section of this document titled “Layout XML” for a full description of this file. The Layout Manager opens this file and parses the XML data, creating instances of the three different types of records as it goes and forming the relations between them via pointers. While parsing the XML, the Layout Manager also sends out messages to the Command Queue Manager to set the switches on the layout into the positions specified in the XML. A complete explanation of how the XML is parsed can be found in the section of this document titled “Start Controller”.

### Data Members

**CmdQueueMgrPtr**

CmdQueueMgrPtr is a pointer to the Command Queue Manager.

**FileIOPtr**

FileIOPtr is a pointer to the File I/O protected type.

**Section(ID, State, BlockingSections, Switches, Sensor1, Sensor2, VirtTrainAddr)**

This is not an instance variable. It is an Ada record type. Instances of these records will be stored in the SectionList data member as described below.

The ID data member is the same ID located in one of the section tags inside the XML file that is parsed when Controller starts. Keep in mind that these are unique and arbitrary IDs. The physical track layout does not have section IDs.

The State data member represents the current state of the section. The states include free, occupied, reserved, and blocked. A section is occupied if any part of a train is located within the section. A section can be reserved by a train which means that a train will be entering the section in the near future, assuming the train’s speed and direction do not change. The train that holds a reservation on a section must be located in an adjacent section. A section is blocked if an overlapping section is either occupied or reserved. A section is free if it is neither occupied, reserved, nor blocked.

The BlockingSections data member is a list of pointers to other section records. This list represents the sections that will be blocked if the section is occupied or reserved. You can also think of this as a list of overlapping sections.

The Switches data member is a list of pointers to switch records. This list represents the switches that are located within the section. If no switches are located within the section, then the list is empty.

The Sensor1 data member is a pointer to a sensor record. This pointer represents the sensor that is located on one end of the section. Similarly, the Sensor2 data member is also a pointer to another sensor record. This pointer represents the sensor that is located on the other end of the section.

The VirtTrainAddr data member will contain the virtual address of a train that is occupying the section or has it reserved. If the section is neither occupied nor reserved, then this data member should be set to NULL or 0.

**SectionList**

This is the list of sections that exist on the track layout. The list contains instances of the Section record type which is described above. The list is initially created when the XML file is parsed when Controller is first started.

**Sensor(ID, State, Sections1, Sections2)**

This is not an instance variable. It is an Ada record type. Instances of these records will be stored in the SensorList data member as described below.

The ID data member is the same ID located in one of the sensor tags inside the XML file that is parsed when Controller starts. Keep in mind that these are unique IDs that correspond to the physical sensor IDs on the track layout.

The State data member keeps track of whether it’s the first or second time the sensor has been fired. To understand what this means, it is important to understand what happens when a magnet comes in close proximity to a sensor on the track. When the magnet initially comes into close proximity with the sensor, the first sensor message is fired. When the magnet leaves the proximity of the sensor, a second sensor message is fired. The first and second messages are identical, so it’s important that Controller keeps track of this. The reason to have two messages sent is so that we’ll know when a train’s magnet passes over the sensor as opposed to having the train’s magnet stop on top of a sensor. This there are two states in which a sensor can be, either a *magnet over* state or a *magnet not over* state. This information is used by the layout manager to determine how to handle sensor fired messages coming in.

The Sections1 data member is a list of pointers to section records. These sections represent the sections that are located on one side of the sensor. This list should contain one or more sections. Similarly, the Sections2 data member is a list of pointers to section records. These sections represent the sections that are located on the other side of the sensor.

**SensorList**

This is the list of sensors that exist on the track layout. The list contains instances of the Sensor record type which is described above. The list is initially created when the XML file is parsed when Controller is first started.

**Switch(ID, State, Sections)**

This is not an instance variable. It is an Ada record type. Instances of these records will be stored in the SwitchList data member as described below.

The ID data member is the same ID located in one of the switch tags inside the XML file that is parsed when Controller starts. Keep in mind that these are unique IDs that correspond to the physical switch IDs on the track layout.

The State data member represents the current state of the switch. The states include closed, thrown, and moving. The meaning of these states is discussed at the end of the “ER Diagram” section of this document.

The Sections data member is a list of pointers to section records. These sections represent the sections that contain the switch.

**SwitchList**

This is the list of switches that exist on the track layout. The list contains instances of the Switch record type which is described above. The list is initially created when the XML file is parsed when Controller is first started.

**XMLFilename**

This is the filename of the XML file that we parse the layout from when Controller first starts. We keep track of this filename in case the user chooses to save the state of Controller. In that case, the filename will be saved to another XML file that is produced when a user saves the state of Controller. Refer to the section of this document titled “Save Controller State” for a full description of saving the state of Controller.

### Functions, Procedures, and Entries

**parseXML(Filename) : boolean**

This function call into the Layout Manager is responsible for parsing the XML file that contains the topology of the physical track layout. The filename of the XML document is passed in. It will open that file and parse the XLM contained therein while creating the instances of Section, Sensor, and Switch and adding them to their respective arrays. Remember that switches have an initial state in the XML. As the switches are parsed from the XML, messages will be sent to the Command Queue Manager to set the switches to the appropriate initial state on the physical track layout. The function will return true if the XML was successfully parsed and the data structures were successfully created. Otherwise, the function will return false.

**sensorFired(SensorID)**

This procedure is responsible for determining which train fired a sensor, and whether it was the front or back of that train which triggered it. The only parameter that it receives is the ID of the sensor that was fired. We can determine which train triggered the sensor and whether it was the front or back of the train that triggered it by analyzing the states of the sections near the sensor. Once we have determined which train triggered the sensor and whether it was the front or back of that train, we will update the states of the appropriate sections in our data structures.

If the front of a train triggered the sensor, we will need to send out some messages to the Command Queue Manager. First, we will generate a SectionOccupyMsg. This message will notify the appropriate tasks, like the Screen Write Task, that a train has just entered a new section. We will also generate a FrontSensorMsg and send it to the Command Queue Manager. This message will contain the sensor ID of the next sensor that the front of the train should trigger if it continues moving and in the same direction. This message should be addressed to the train that triggered the sensor. We generate this message so that the Front Sensor ID of the Train Task can stay up-to-date.

If the front of the train trigged the sensor, we will also need to make an attempt to reserve the next section for the train. The reservation will be successful if the next section’s state is set to free. Otherwise, the reservation will fail. If the reservation is successful, we will generate a SectionReservedMsg. This will notify the appropriate tasks, like the Screen Write Task, that a train has just made a reservation on a section. If the reservation failed, we will generate a TrainWaitMsg. This will notify the Train Task that it must enter the waiting state since it could not get the reservation and thus is not safe to continue moving.

If the back of a train triggered the sensor, we will generate a BackSensorMsg and send it to the Command Queue Manager. This message will contain the sensor ID of the sensor that was just fired. We generate this message so that the Back Sensor ID of the Train Task can stay up-to-date. Also, if the back of a train triggered the sensor, then we will generate a SectionFreeMsg. This message will notify the appropriate tasks, like the Screen Write Task, that a section is now free. This message is also sent to every Train Task. When a Train Task receives it and the Train Task’s current state is set to waiting, then the Train Task will attempt to make a reservation again by calling the attemptReservation() procedure described below. When a Train Task receives it and the Train Task’s current state is set to anything other than waiting, then the message is discarded and ignored.

**placeTrain(VirtTrainAddr, SensorList, return)**

This procedure is called when a new train is being registered. It takes in a list of sensors as a parameter. This list of sensors locates where the train wishes to be placed. The Layout Manager checks the state of all the sections between the sensors in the list. The location is valid if the state of all the sections is free. The location is not valid if the state of any of the sections is anything other than free. If the location is valid, then the data structures are updated so that they become occupied by the virtual address provided as a parameter. A SectionOccupyMsg is also generated and sent to the Command Queue Manager for each section that became occupied. This message will notify all the appropriate tasks, like the Screen Write Task, that a train is now occupying these sections. The procedure will then return a true boolean value in return parameter, which is simply an out parameter that acts in the same manner that a function’s return value does. If the location was not valid, then no changes are made to Layout Manager’s data structures and no messages are generated. In this case the procedure returns a false boolean value in the return parameter.

**occupySeg(FrontSensorID, TrainID)**

This procedure is called when a train is in a begin halt or moving state and a front sensor goes off then the train makes a call into the layout manager to change the reserved segment to occupied. The front sensor is passed because the reserved segment should be right next to it and it will be obvious for the layout manager which segment needs to be occupied by finding the reserved segment with a trainID which matches that which was passed in. If the layout manager can successfully occupy the segment then a SectionOccupiedMsg is sent out from layout manager to the command queue manager to inform the user that the occupy occurred. Only a reserved segment can be occupied, if there are no reserved segments next to the front sensor with a matching trainID then the system has entered an error state and sends an error message to the command queue manager.

**freeSeg(OldBackSensorID, TrainID)**

This procedure is called by a train task when the train is in a begin halt, begin wait, or moving state and it receives a BackSensorMsg message. This means that the train needs to free the segment it just moved out of. The old back sensor is passed along with the trainID of the train sensor just went off. The old back sensor is passed because the segment which needs to be freed should be the only segment which is adjacent to the sensor and is occupied by the trainID which is passed in. If there is no segment next to the sensor which matches this description or there are more than one which fits this description then the system has entered an error state sends an ErrorMsg to the command queue manager. If it kind find a segment next to the sensor which is occupied by TrainID then it sets this segment to free and sends out a SectionFreeMsg.

**attemptReservation(VirtTrainAddr, FrontSensorID, return)**

This procedure is called when a train that is in the halted or waiting state wishes to resume moving again. It is called directly from a Train Task. The procedure will be able to determine which section to attempt to make a reservation on based off of the FrontSensorID parameter passed in. This parameter represents the next sensor the train will fire if it continued moving in its current direction. The section can be reserved if its current state is set to free. In this case, then we will update the data structures and send a SectionReservedMsg to the Command Queue Manager. A boolean value of true will be returned in the return out parameter. The section cannot be reserved if its current state is anything other than free. In this case, we will not make any updates to the data structures, and we will return a false boolean value in the return out parameter.

**dropReservation(VirtTrainAddr, FrontSensorID)**

This procedure is called when a train enters the halted state. This is because a train that is halted should never be holding a reservation. It is called directly from a Train Task. The procedure will be able to quickly determine which section the train currently has reserved based off of the FrontSensorID parameter that is passed in. This parameter represents the next sensor the train will fire if it continues moving in its current direction. When the reservation is dropped, a SectionFreeMsg will be sent to the Command Queue Manager. This message will notify the appropriate tasks, like the Screen Write Task, that a section is now free. This message is also sent to every Train Task. When a Train Task receives it and the Train Task’s current state is set to waiting, then the Train Task will attempt to make a reservation again by calling the attemptReservation() procedure described above. When a Train Task receives it and the Train Task’s current state is set to anything other than waiting, then the message is discarded and ignored.

**updateNextSwitch(VirtTrainAddr, FrontSensorID)**

This procedure is called when an engineer attempts to close/throw the next switch in front of the train they’re controlling. For our purposes here, we’ll consider the “next switch” to be the next switch that a train will hit if it continues moving in its current direction. We will not consider a switch located in a section reserved by the train to be the “next switch”. In this case, the switch after that will be the “next switch”. The procedure will recursively traverse through the internal data representation of the layout until the next switch is found. We will know what direction to traverse down based off of the virtual address of the train and FrontSensorID which are both passed in as parameters. Once the switch we wish to throw has been found, we must check the current state of every section that contains this switch. If all of the section’s states are set to either free or blocked, then we may continue closing/throwing the switch. If any of the section’s states are set to occupied or reserved, then we may not continue closing/throwing the switch. To actually close/throw the switch, we will generate a PhysicalSwitchCloseMsg or PhysicalSwitchThrowMsg and send it to the Command Queue Manager.

**switchFinishedMoving(SwitchID, State)**

This procedure is called when a message comes off the LocoNet saying that a switch has finished moving. The procedure simply indexes into the array of switches using the SwitchID that is passed in as a parameter. The switch’s state is updated from moving to either closed or thrown, depending on the value of the State parameter.

**occupySections(VirtTrainAddr, SensorList)**

This procedure is called by a Train Task when the system is being restored from a restore file to a saved state. It passes in a sensor list and a virtual address. The train is then placed on the layout in the location specified by the sensor list.

**saveState(Filename)**

This procedure is called from the Layout Task when a user wishes to save the current state of Controller to an XML file. In this procedure, we will write out all the information that is important to the current state of the tack layout to an XML file. The XML file should be saved at the filename specified by the parameter. The type of information we’re interested in is the current state of switches, the location of trains, the reservations trains currently hold, etc. We write to the XML file via the save() procedure call in the File I/O protected type. We use this protected type to write to the File because it is possible for Train Tasks to wish to write to the file at the same time. For a detailed discussion about saving the state of Controller, refer to the section of this document titled “Save Controller State”.

**restoreLayout(filename)**

This procedure is called by the Layout Task when a user wants to restore the state of Controller from a restore file. The user may only restore the state of Controller from a restore file when Controller first starts.

**getSensorList(VirtTrainAddr, FrontSensor, BackSensor) : SensorList**

This procedure is called by a Train Task in order to get a list of all the sensors that locate the train. Generally, this function is called when a train is saving it’s state to the restore file. The train passes in its virtual address, next front sensor, and previous back sensor. This will allow the Layout Manager to look in its data structures and figure out which sensors locate the train.

## File I/O

The File I/O protected type is responsible for ensuring that multiple Controller tasks can’t write to the same file at the same time. There are two files that we’re primarily concerned with. The first is the restore file that Controller writes to when saving the state of Controller. This should be an XML file. For a complete discussion on saving the state of Controller, refer to the section of this document titled “Save Controller State”. The second file we’re concerned with is the log file that is written to while Controller runs. This is a simple text file. For a complete discussion on the log file, refer to the section of this document titled “Log File”.

It is necessary to have a protected type to facilitate I/O, because in both of the situations just described, multiple tasks may wish to write at the same time. If we didn’t use a protected type, then the data written to the files could very well be interleaved together.

### Functions, Procedures, and Entries

**saveLayout(filename, data)**

This procedure is called from the Layout Manager when we need to save information about the current state of the layout to a restore file. By “current state of the layout”, we’re referring to the information that the Layout Manager knows about. This includes the location of trains on the track, the reservations a train might have, the state of all the switches, etc. All this data should be included in a single string that is passed into this function as the data parameter. The filename of the restore file is provided as a parameter.

**saveTrain(filename, data)**

This procedure is called from a Train Task when we need to save the current information of a single train into the restore file. By “current information of a single train”, we’re referring to the information that a Train Task knows about. This includes a train’s virtual address, speed, direction, etc. All this data should be included in a single string that is passed into this function as the data parameter. The filename of the restore file that it should be written to is provided as a parameter.

**restore(filename) : restore info**

This procedure is called by the Layout Task and the Layout Manager. Its purpose is to open up the restore file, given by the filename parameter, and return information used to restore Controller to the saved state. If the function is called by the Layout Task, then it will use the information about the trains to spawn off the new Train Tasks. If the function is called by the Layout Manager, then it will the information about the track layout to update Layout Manager’s data structures properly.

**updateLog(filename, line)**

This procedure is called by the Layout Manager and Train Tasks. It is used to write a single line of data to the log file with the name provided as the filename parameter. A timestamp should automatically be written at the beginning of any line of data that is passed in.

Package List

In the previous section we discussed the tasks and protected types that make up Controller. In this section we will talk about how to group those tasks and protected types together in a logical fashion to form packages. Below, we will list what pieces of code should be included in each package and why.

## LocoBuffer Package

*Martin March 31: Here is a new design for this piece.*

The LocoBuffer Package will contain the LocoBuffer Server and the Slot Lookup Table. We’ve chosen to package these two together because the LocoBuffer Server must reference the Slot Lookup Table regularly in order to translate messages between slot numbers and addresses.

## Keyboard/Screen Package

The Keyboard/Screen Package will contain the Keyboard Read Task and the Screen Write Task. We’ve chosen to group these tasks together in the same package because they both deal with user interaction via the ASCII user interface.

## iToy Package

The iToy Package will contain the iToy Read Task and the iToy Write Task. It is convenient to place both of these tasks in the same package because they are both going to be communicating with an iToy via TCP/IP. It is fair to assume that both of these tasks will need share similar information that will allow them to communicate over the network. By placing both tasks in the same package, this type of can be encapsulated within the private section of the package so that it’s not freely available to all the other tasks and protected types in Controller.

## Command Queue Package

The Command Queue Package will only contain the Command Queue Manager. We justified putting the Command Queue Manager in its own package because is central to Controller, and there are no other tasks or protected types that serve a similar purpose. The Command Queue Package should also contain the Client Socket which Controller uses to communicate with the LocoBuffer Server.

## Layout Package

The Layout Package will contain the Layout Manager and the Layout Task. Besides having similar names, it makes sense to include these two in the same package because the types of messages that the Layout Task handles usually have a direct impact on the Layout Manager’s internal data representation of the track.

## Trains Package

The Trains Package will contain the Train Task. We felt justified in giving trains its own package because it doesn’t identify very closely to any other tasks or protected types in Controller. Train Tasks do make a number of function calls to the Layout Manager; however including Train Tasks in the Layout Package didn’t sit well with us.

The Trains Package is somewhat unique in that the package itself contains a procedure outside of any task or protected type. This procedure is the spawnTrain() function, and it is described as follows:

**spawnTrain(VirtTrainAddr, State, Direction, Speed, FrontSensorID, BackSensorID)**

This procedure is responsible for creating a new Train Task. It will be called from the Layout Task when the system is first starting up, or when the system is being restored from a saved XML file. The virtual address of the train should be passed in via the VirtTrainAddr parameter. The FrontSensorID should be the ID of the first sensor the front of the train will hit if it moves in the forward direction. The BackSensorID should be the ID of the sensor immediately behind the back of the train. Both of these values can be found from the Layout Manager after the train has been placed in its data structures.

In the case that the system is first starting up, State should be set to halted, Direction should be set to forward, and Speed should be set to 0. These are the default values for an initial train. In the case that the system is being restored from a saved XML file, these values should be set to whatever was specified in the XML.

## File I/O Package

The File I/O Package will only contain the File I/O protected type. Since many different tasks and protected types may use the File I/O protected type to write to a log file, it didn’t make much sense to bundle it inside one other package.

Communicating with the LocoBuffer

A crucial aspect of Controller is how to communicate with the LocoBuffer. Without this piece of functionality, Controller will not be able to communicate with the throttle or the physical track layout. This would render Controller useless.

The only component that talks with the LocoBuffer is the LocoBuffer Server. Nothing else should ever communicate directly with the LocoBuffer.

First, let’s explain how the LocoBuffer is represented on the computer. This will give us the foundation of understanding how software can speak with a hardware component. The LocoBuffer is connected to a computer via a USB port. The LocoBuffer also comes with some software drivers which must be installed on any computer that wishes to communicate with the LocoBuffer. The LocoBuffer is represented on the computer as a COM port. Although we will try to exclusively refer to this port as a COM port, it can also be referred to as a serial port.

A COM port isn’t a very advanced type of adapter. It essentially passes serialized information. It is the responsibility of the sender to take meaningful data and serialize it in some way. It is the responsibility of the receiver to take the serialized information and convert it back into meaningful data. For our purposes, the LocoBuffer Server will act as the receiver and the sender of data. The data being sent over the COM port must be LocoNet messages. For more information about these messages, refer to the section of this document titled “LocoNet Messages”.

The LocoBuffer Server will take serialized data from the COM port. This serialized data represents LocoNet messages. It will interpret these LocoNet messages and then convert them into the proper internal messages that Controller uses. It will then send these messages to the Command Queue Managers of all the Controller clients currently connected to the LocoBuffer Server via TCP/IP.

The LocoBuffer Server will receive internal messages from the Command Queue Managers of all the Controller clients currently connected to the LocoBuffer Server via TCP/IP. Keep in mind that Controller and the LocoBuffer Server uses sockets to facilitate this TCP/IP communication. Once a message is received, it will then convert the message into a LocoNet message, and then send it to the COM port.

Now, up until this point, we’ve led you to believe that the LocoBuffer Server talks directly with the COM port. This isn’t *exactly* true. It responsible for doing the conversion between the serialized data and the internal messages. However, the actual communication with the COM port is done via two functions written in C. We use C because that language is better suited for dealing with this low level task. We’ll refer to the C code that is used as the cLocoIO package. Note that for the sake of clarity, we omit the cLocoIO package from other sections of this document. In all other sections of the document, we abstractly refer to the LocoBuffer Server as communicating with the LocoBuffer. Refer to Figure 25 for an illustration on how some of the different components relate to each other.

Martin April 1: This has changed



Figure - Communicating with the LocoBuffer

As previously mentioned, the cLocoIO only has two functions that the LocoBuffer Server utilizes. We will call the first function readFromCOM(). As you would expect, this function reads serialized data from the COM port and returns it from the function. This function takes no parameters and returns an unsigned \_\_int8.

We will call the second function writeToCOM(). Again, as expected, this function writes data to the COM port. This function takes one parameter of type unsigned \_\_int8 and returns void.

There will also be a third function in cLocoIO which is responsible for initializing the connection with the COM port. We’ll call this function initializeCOMPort(). It takes no parameters. It will create a handle to the COM port. This requires the configuration of a number of settings. The handle should be saved as a global variable so that the readFromCOM() and writeToCOM() functions can access it.

To make your lives exponentially eaiser, we’ve included one implementation of cLocoIO in Figure 26. This implementation was written by Cody Baxter in Fall 2010.

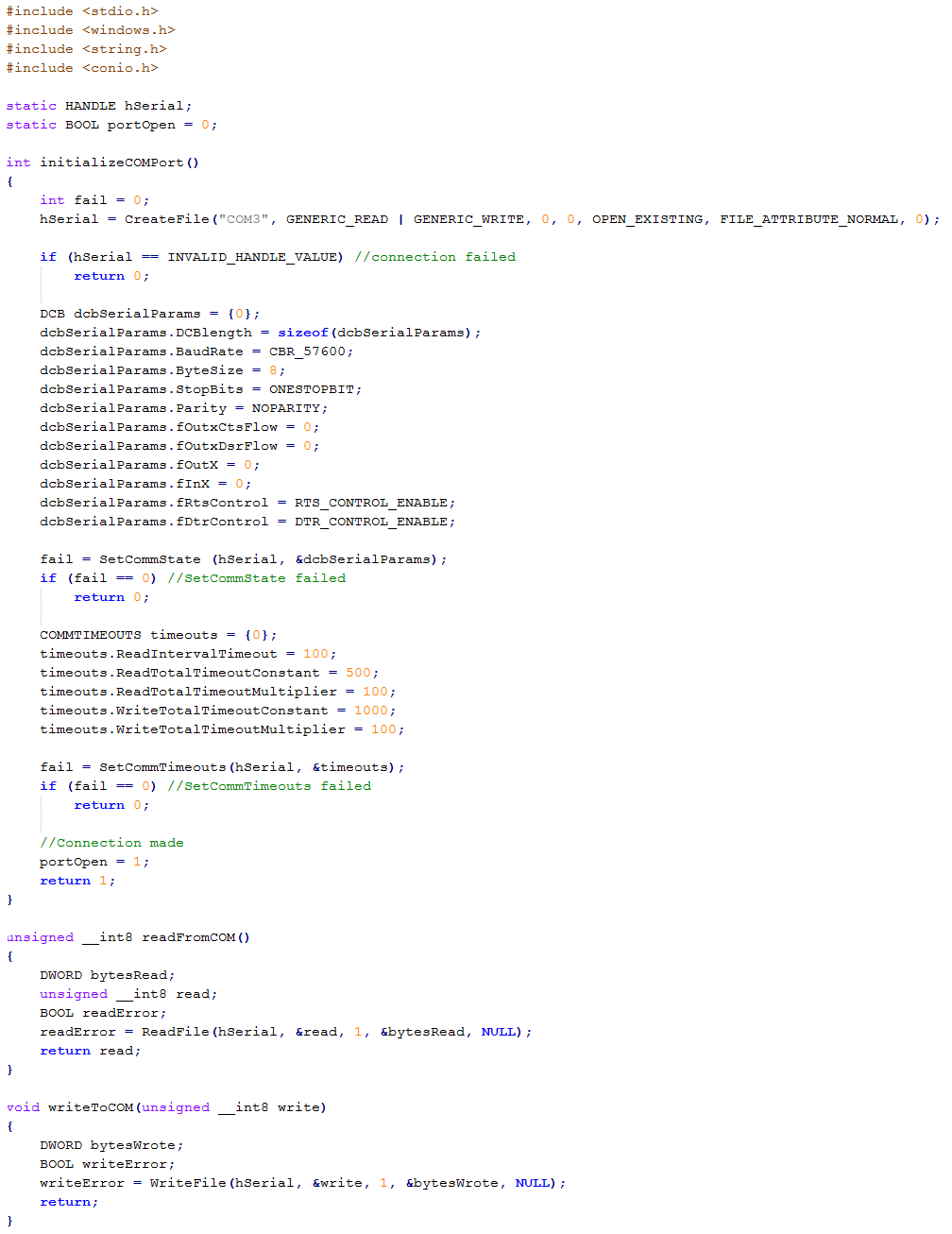


Figure - cLocoIO

Now, we must answer the question of how does our Ada code communicate with these C function? This can be done using Ada’s pragma Import. Pragma Import directs the compiler to use code written in a foreign language. It essentially tells our Ada code that there exists some C function that we’ll use. We assign a name to that function so we can reference it in our Ada code. Examples of how to do this have been included in Figure 27. Note that this code is merely an example of how to use Ada’s pragma Import. The package in this code is only an example package and does not represent a package that should actually be included in Controller. The code included in this example should be added to the LocoBuffer Package.

In this example, Ada will recognize three functions. The first is the initializeCOMPort() function that was written in C. This function takes no parameters and returns an integer. This function can be referenced in the Ada code through the name initiailizePort(). The second function is the readFromCOM() function that was written in C. This function takes no parameters and returns an Interfaces.Unsigned\_8 data type. This function can be referenced in the Ada code through the name readData(). The third is the writeToCOM() function that was written in C. This function takes in one parameter of type Interfaces.Unsigned\_8. It has no return value. This function can be referenced in the Ada code through the procedure name writeData().

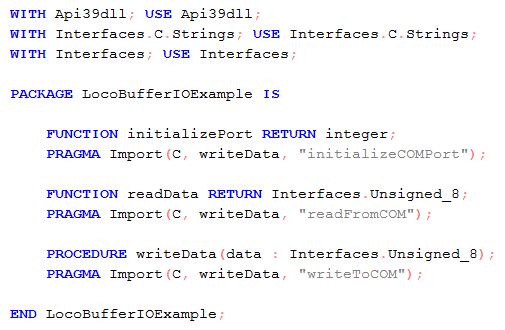


Figure - Ada Pragma Import Example

Train States

Martin April 1: These probably contain mistakes and omissions.

In this section of the document, we will go into more detail on the different states a train can be in, and how a train can enter or leave these states. Keep in mind that there are five main states: moving, halted, waiting, begin halted, and begin waiting. A train is in the moving state if it has a speed greater than zero. A train is in the halted state if it has a speed of zero, and they physical train is no longer moving. A train is in the begin halted state if it’s getting ready to enter the halted state, but the physical train still has some momentum and hasn’t come to a full stop yet. A train is in the waiting state if it has a speed greater than zero, however it is still stopped. A train enters the waiting state because it was unable to gain a reservation on the next section it wishes to enter. The reason the speed of the train is greater than zero is because when the train is finally cleared to move again, we want to restore its speed back to the same value it was before it was forced to wait. A train is in the begin waiting state when it is getting ready to enter the waiting state, but the physical train still has some momentum and hasn’t come to a full stop yet.



Figure - Train Moving State Diagram

Refer to Figure 28 for the state diagram of a moving train. If a train receives a speed message while in the moving state, it must determine whether the speed in the message was zero or greater than zero. In the case that the speed is greater than zero, the train will remain the moving state. If the message’s speed is equal to zero, then the train will enter the begin halt state.

If the front of a train has tripped a sensor, an attempt is made to get a reservation for the following section. If the reservation is successful, the train remains in the moving state. If the reservation is unsuccessful, the train enters the begin wait state.

If the back of a train has tripped a sensor, we simply remain in the moving state.



Figure - Train Begin Halt State Diagram

Refer to Figure 29 for the state diagram of a train in the begin halt state. Once a train’s physical velocity is zero, it enters the halt state. We know when a physical train has actually stopped moving because a timer is set when the train’s speed is set to zero. When the timer expires, we know that enough time has elapsed that the physical train should no longer be moving.

If a train receives a speed message while in the begin halt state, it must determine whether the speed in the message was zero or greater than zero. In the case that the speed is equal to zero, the train will remain in the begin halt state. If the message’s speed is greater than zero, then the train will attempt to make a reservation. If the reservation is successful, then the train enters the moving state. If the reservation is unsuccessful, then the train enters the begin wait state.

If the front of a train has tripped a sensor while in the begin halt state, then we check if that train has a reservation on the section we just entered. It is important to know that a train’s reservations are not dropped until a train enters the halted state. If the train does have a reservation on that section, then the train remains in the begin halted state. If the train does not have a reservation on that section, the we have entered an error condition. This is because Controller should never enter a section that it does not have reserved.

If the back of a train has tripped a sensor while in the begin halt state, we simply remain in the begin halt state.



Figure - Train Halted State Diagram

Refer to Figure 30 for the state diagram of a halted train. If a train receives a speed message while in the halted state, we must check whether the speed in the message is zero or greater than zero. If the speed is equal to zero, then we simply remain in the halted state. If the speed is greater than zero, we must attempt to make a reservation. If the reservation is successful, then the train enters the moving state. If the reservation is unsuccessful, then the train enters the waiting state.

If either the front or the back of a train trips a sensor while in the halted state, then Controller has entered an error condition. Since the train has no velocity, there is no reason why it should have tripped a sensor.



Figure - Train Begin Wait State Diagram

Refer to Figure 31 for the state diagram of a train in the begin wait state. Once a train’s physical velocity is equal to zero, the train enters the waiting state. We can tell when a train’s physical velocity is equal to zero because a train has a timer that is set once a train enters the begin waiting state. When that timer expires, the physical train should have lost momentum and it’s speed should be zero.

If the front of a train trips a sensor while in the begin wait state, then Controller has entered an error condition. This is because the train must have entered the begin wait state by not being able to obtain a reservation for the next section it wishes to enter. If front of the train tripped a sensor, then that means we must have entered the section that we couldn’t get the reservation for.

If the back of a train trips a sensor while in the begin wait state, then we simply remain in the begin wait state.

If a train receives a speed message while in the begin wait state, then we must check whether the speed in the message is zero or greater than zero. In the case that the speed is zero, the train enters the begin halt state. In the case that the speed is greater than zero, the train remains in the begin wait state. We do not attempt to make a reservation, because trains will be notified when they may attempt to make a reservation again.

As we just mentioned, a train is told via a message that they can attempt to make a reservation again. If the reservation is successful, the train enters the moving state. If the reservation is unsuccessful, the train remains in the begin wait state.



Figure - Train Wait State Diagram

Refer to Figure 32 for the state diagram of a train in the waiting state. If either the front or the back of a train trips a sensor while in the waiting state, then Controller has entered an error condition. This is because a train should have no velocity while in the waiting state. Therefore, a train should not be able to trip a sensor while in the waiting state.

If a train receives a speed message while in the waiting state, we must check whether the speed in the message is zero or greater than zero. In the case that it’s equal to zero, then the train will enter the halted state. In the case that it’s greater than zero, the train will remain in the waiting state. We do not attempt to make a reservation, because trains will be notified when they may attempt to make a reservation again.

As we just mentioned, a train is told via a message that they can attempt to make a reservation again. If the reservation is successful, the train enters the moving state. If the reservation is unsuccessful, the train remains in the wait state.

Switch States

Martin April 1: Need to double check this

In this section of the document, we will go into more detail on the different states a switch can be in, and how a train can enter or leave these states. Keep in mind that there are three main states: closed, thrown, and moving.



Figure - Switch State Diagram

Refer to Figure 33 for the state diagram of a switch. When a switch is in the closed state, it may receive a message to either throw or close the switch. If it receives a message to close, then we do nothing and remain the closed state. If it receives a message to throw the switch, it enters the moving state. The events are symmetrical for a switch in the thrown state.

When a switch is in the moving state, it may receive a message to throw or close the switch. In either case we remain in the moving state. The switch will receive a message once it has finished moving. When this occurs, the switch will enter either the closed or thrown state.

Data Flow Diagram

Figure 34 is a representation of how data flows between the different tasks and protected types in Controller. It illustrates how the individual components of Controller relate to each other and interact.

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP

(updated by Scott 04/01)



Figure - Controller Data Flow Diagram

Internal Messages

|  |  |  |  |
| --- | --- | --- | --- |
| **Message Name** | **Source** | **Destination** | **Data Members** |
| **TrainSpeedMsg** | LocoBuffer Read Task | Train Task | VirtTrainAddr  Speed |
| **PhysTrainSpeedMsg** | Train Task | LocoBuffer Write Task  Screen Write Task  iToy Write Task | VirtTrainAddr  Speed |
| **PhysTrainStopMsg** | Train Task | LocoBuffer Write Task, Screen Write Task,iToy Write Task | VirtTrainAddr |
| **PhysSwitchCloseMsg** | Layout Manager | LocoBuffer Write Task, Screen Write Task, iToy Write Task | Switch ID |
| **PhysSwitchThrowMsg** | Layout Manager | LocoBuffer, Write Task, Screen Write Task, iToy Write Task | Switch ID |
| **SensorMsg** | LocoBuffer Read Task | Layout Task | Sensor ID |
| **FrontSensorMsg** | Layout Manager | Train Task | VirtTrainAddr  New Front Sensor |
| **BackSensorMsg** | Layout Manager | Train Task | VirtTrainAddr  New Back Sensor |
| **SectionOccupyMsg** | Layout Manager | Screen Write Task  iToy Write Task | VirtTrainAddr  Section ID |
| **SectionFreeMsg** | Layout Manager | Screen Write Task  iToy Write Task | VirtTrainAddr  Section ID |
| **SectionReserveMsg** | Layout Manager | Screen Write Task  iToy Write Task | VirtTrainAddr  Section ID |
| **SectionBlockedMsg** | Layout Manager | Screen Write Task  iToy Write Task | VirtTrainAddr  Section ID |
| **TrainWaitMsg** | Layout Manager | Train Task | VirtTrainAddr |
| **RegisterTrainIMsg** | Keyboard Read Task  iToy Read Task | Layout Task | PhysTrainAddr  Sensor List |
| **RegisterTrainIIMsg** | Layout Task | LocoBuffer Write Task | PhysTrainAddr |
| **RegisterTrainIIIMsg** | LocoBuffer Read Task | Layout Task | PhysTrainAddr  PhysSlot#  VirtTrainAddr  VirtSlot# |
| **RegisterTrainIVMsg** | Layout Task | Screen Write Task  iToy Write Task | VirtTrainAddr  Sensor List |
| **UpdateNextSwitchMsg** | LocoBuffer Read Task  iToy Read Task  Keyboard Read Task | Train Task | VirtTrainAddr |
| **SwitchMoveDoneMsg** | LocoBuffer Read Task | Layout Task | Switch ID  Switch State |
| **StartSystemMsg** | Keyboard Read Task | Layout Task | Filename |
| **ParseCompleteMsg** | Layout Task | Screen Write Task | Filename |
| **ParseFailMsg** | Layout Task | Screen Write Task | Filename |
| **RestoreLayoutMsg** | Keyboard Read Task | Layout Task | Filename |
| **RestoreTrainMsg** | Layout Manager | Layout Task | Filename  VirtTrainAddr |
| **SaveLayoutMsg** | Keyboard Read Task | Layout Task | Filename |
| **SaveTrainMsg** | Layout Manager | Train Tasks | Filename |
| **SaveResultMsg** | Layout Task | Screen Write Task | Filename  Save Results (boolean) |
| **ErrorMsg** | Any | Screen Write Task | Error Type |

Analyzing Sensor Fire Message

In this section we will discuss how the train system knows what to do when it receives a sensor fired message. It will receive this message through the LocoBuffer Read Task which will translate the message into an internal SensorMsg, containing the ID of the sensor that was fired. This message will be passed to the Command Queue Manager who will forward it to the Layout Task. This task then makes the sensorFired() function call into the Layout Manager. From here the system looks up the record in the list of sensors matching the ID given. It can do this in constant time since the index into the array will match the ID of the sensor. Once we locate the sensor record, we can look at the adjoining sections, to determine which train fired the sensor, whether it was the front or back of the train that fired it, and whether or not we’ve reached an error condition.

The best way to understand the logic involved is to see several examples of combinations of states the sections can be which adjoin the sensor that fired. There are only two valid safe states. We define a safe state as being one which does not produce an error condition. This first is if the sensor is surrounded by an occupied and a reserved segment where they are each occupied or reserved by the same train. This means the sensor was tripped by the front of the train because reserved sections can only be located in front of the train in the direction of travel. An illustration of this situation can be seen in Figure 35. We know that the new front sensor of the train is the sensor located at the other end of the reserved section, and this is sent in the FrontSensorMsg to the Train Task that reserved the section. If the sections were occupied and reserved by different trains then an error condition has been reached and an ErrorMsg needs to be sent in order to halt Controller.

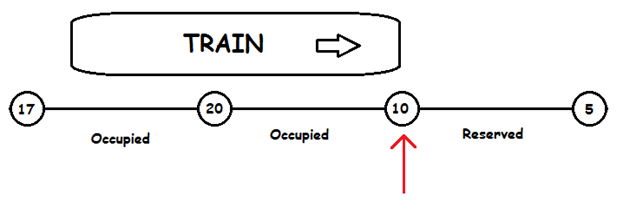


Figure - Front Sensor Firing

In another situation, the sensor that went off may be surrounded by two sections which are both occupied by the same train. In this case, we need to follow the track out one more section in either direction to confirm that one of these sections of track further out is either free, occupied, reserved, or blocked by another train. If this is the case, then we know it was the back of the train that set off the sensor. This is because the next sensor that the back of the train will hit will be surrounded by two sections that are both occupied by the train. Then one more section out in one of the directions, the section should not be occupied nor reserved by the train. An illustration of this situation can be seen in Figure 36. In this situation, the new back sensor of the train is the one which was just fired. The ID of this sensor is placed in a BackSensorMsg which is sent to the Train Task corresponding to the train we were just working with. If they both occupied by the same train then a sensor went off in the middle of the train and we are in an error state and as such must send an ErrorMsg to the Command Queue Manager.

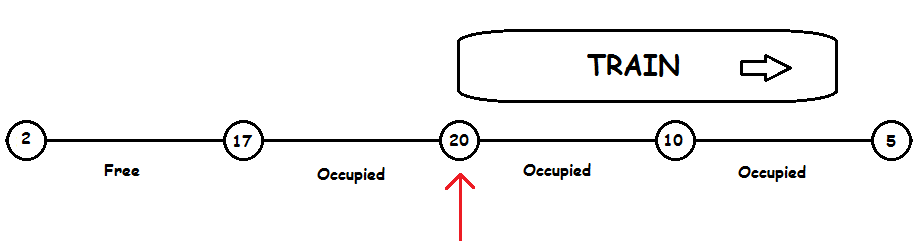


Figure - Back Sensor Firing

Any other situations that may occur when a sensor fires are considered to be invalid. An example of such a situation is similar to what we saw when the front sensor fired. The only difference is if the two sections adjacent to the sensor that fired are reserved and occupied by different trains. This would mean that a train entered an invalid section and the Controller must be halted. Another example of such an error state situation occurs if a sensor goes off and the two adjacent sections are occupied by the same train, and the two sections past those are also occupied by the same train. This would mean that a sensor went off in the middle of a train.

We should always check to see if the front of a train fired a sensor before checking to see if the back of the train fired a sensor.

In the paragraphs above we were always talking about sensors firing in a more broad manner as if the system only gets one message when a sensor is fired. This is not true, in fact whenever a train passes over a sensor it gets two messages, one stating that the trains magnet is on top of a sensor and the other stating that the trains magnet is moving off the sensor. The reason this is important is that if a train moves on top of a sensor and stops then it only receives one of these messages and then if it restarts it can cause problems with the trains location and reserving sections. This may seem like a complicated problem but it is really relatively easy to solve, the sensor object within the system has a state variable which is either in the *magnet above* or *magnet not above* state, this enables us to respond to the messages correctly by knowing which message the train is receiving, the first or the second. For handling a sensor fire message we need to determine which state end of the train caused the sensor to fire and what state the sensor is in, from these two pieces of information we can determine the correct course of action.

The front sensor **must** only respond to the second sensor fires message (the sensor which fired is in the *magnet above* state) and the back sensor **must** only respond to the first sensor message (the sensor which fired is in the *magnet not above* state). By respond we mean that the Layout Manager sends out a FrontSensorMsg or BackSensorMsg. We can determine which end of the train caused the sensor to fire by the logic stated above. If the Layout Manager gets a sensor fired message and the end of the train and state of the sensor don’t correspond to causing the Layout Manager to “respond” in the manner stated above, then the sensor must toggle its state and not FrontSensorMsg or BackSensorMsg is sent.

While you may be wondering why it is important for the sensors in the front or back of the train to respond to a particular sensor fired message and not just say the first, it becomes clear after a few situations with trains stopping on sensor, resuming or changing direction and resuming. For example if a train is moving and the front magnet trips a and the train stops before the moving off the sensor and getting the second message then the train would occupy the new segment and be in the halted state. The problem arises when the train changes direction and now the “back” of the train moves off the sensor causing it to fire, but this will look to the system as if the sensor firing is in the middle of the train and this will cause the train set to enter an error state. There are other such situations as this which cause issues and it because of this that it is imperative that the train’s sensors have state and that the Layout Manager only responds by sending a message in the correct set of circumstances.

Start Controller

(updated by Scott 04/01)

In this section we will describe what happens when the system is first started up. When starting the system, the user may choose to read in an XML file or a restore file. For information about each of these types of files, refer to the sections of this document titled “Layout XML” and “ASCII User Interface”. Refer to the collaboration diagram in Figure 37.

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP



Figure - Starting Controller

1. When Controller is first launched via a separate executable file which has the responsibility of starting all the necessary protected types and tasks. First instances of the protected types (Command Queue Manager, File I/O, Slot Lookup Table and Layout Manager) are started. We start these first because the tasks will need references to these objects. Next, the tasks are started. When instantiating a task, we can pass in the references to the protected types that they need access to. For example, when the Layout Task is started, it is passed references to the Layout Manager, Slot Lookup Table and Command Queue Manager because these are all the protected types that this task must talk to.
2. Once the Screen Write Task is started, it sends a message to the screen to ask the user if they want to start the system with either an XML file or a restore file. This is the startup screen described in the section of this document titled “ASCII User Interface”.
3. The Keyboard Read Task reads in the users response and determines from the first character (x or r) if the file is a restore or XML file. If the user selects the restore file, stop reading this section of the document and instead refer below where we discuss what happens when the system is restored. If the user selects an XML file, continue reading.
4. Once we have determined that we’ll use the XML file to startup, the Keyboard Read Task sends a StartSystemMsg to the Command Queue Manager.
5. The Layout Task then dequeues this message from its queue.
6. The Layout Task then makes a parseXML() function call into the Layout Manager and passes in the name of XML file as a parameter. The Layout Manager reads the XML file and parses it. While it’s parsing, it creates the data structures which represent section, sensors, and switches. For a full description of this data structure, refer to the section of this document titled “Tasks and Protected Types”.
   1. Once the Layout Manger has finished parsing the XML file, multiple PhysSwitchCloseMsg’s and PhysSwitchThrowMsg’s are sent to the Command Queue Manager. These messages set all the switches on the layout to their proper orientation, as described in the XML.
   2. The Layout Manager communicates with the File I/O protected type in order to write to the log file.
7. If the parseXML() function returns as being successful, then the Layout Task sends a ParseCompleteMsg to the Command Queue Manager. If the parseXML() function returns as being unsuccessful, then the Layout Task sends a ParseFailedMsg to the Command Queue Manager.
   1. The Command Queue Manager sends the PhysSwitchCloseMsg/PhysSwitchThrowMsg sent from #6a to the LocoBuffer Server via the Client Socket. The LocoBuffer Server makes the appropriate conversions and sends the message out on the LocoNet.
   2. The Screen Write Task dequeues the message sent in #7, and either shows the user the command input UI or asks the user to enter the name of another XML file.

Register Train

(updated by Scott 04/01)

In this section, we will describe what happens in Controller when a train is registered. Refer to the collaboration diagram in Figure 38.

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP



Figure - Register Train

1. The user enters a command to register a train. This can happen via the keyboard or an iToy. In either case, command is translated into a RegisterTrainIMsg. This message contains the physical train address of the train being registered as well as a list of sensors that locates the train. The sensors should be listed in order from front to back.
2. The message is send to the Command Queue Manager where it is added to the Layout Task’s queue.
3. The Layout Task dequeues the message from it’s queue for processing. The Layout Task stores the physical train address and the list of sensors in a temporary location.
4. The Layout Task sends a RegisterTrainIIMsg to the Command Queue Manager. The Command Queue Manager sends this message to the LocoBuffer Server via the Client Socket. The message contains the physical train address of the train being registered.
5. Once the LocoBuffer Server receives the message from #4, it is responsible for getting the virtual address that will be associated with the physical train address. This can be done via the getVirtualAddress() function call in the Slot Lookup Table. The LocoBuffer Server then writes the appropriate messages to the LocoNet to register the physical train address and the virtual train address. The LocoBuffer Server will receive a message back that contains the physical train slot number and the virtual train slot number. The LocoBuffer Server creates a RegisterTrainIIIMsg which is sent back to the Command Queue Manager of Controller via the Server Socket. This message contains the physical train address, the physical train slot number, the virtual train address, and the virtual slot number. The Command Queue Manager forwards this message to the Layout Task.
6. The Layout Task dequeues the message from its queue for processing. It associates the sensor list that we saved earlier to the addresses in the message. It can do this because the physical train address saved along with the sensor numbers will match the physical train address contained in the message.
7. The Layout Task calls the placeTrain() function in the Layout Manager in an attempt to place this new train on the track. If the function returns false, an ErrorMesg is sent to the Command Queue Manager. The Layout Manager will communicate with the File I/O protected type in order to write to the log file.
8. The Layout Task spawns a new Train Task by calling the spawnTrain() function located in the Trains Package. The newly spawned Train Task will communicate with the File I/O protected type in order to write to the log.
9. The newly spawned Train Task will call the registerQeueue() function in the Command Queue Manager in order to get a message queue established.
10. When the spawnTrain() function call from #14 returns, the Layout Task will call the createEntry() procedure located in the Slot Lookup Table. This will create a new entry in the Slot Lookup Table that associates a virtual address, virtual slot number, physical train address, and physical train slot number.
11. The Layout Task sends a RegisterTrainIVMsg to the Command Queue Manager. This message contains the virtual address of the train as well as the list of sensors that locate the train. The Command Queue Manager places this message on the queues for the Screen Write Task and the iToy Write Task.
12. The Screen Write Task and the iToy Write Task dequeue the message.
13. The Screen Write Task updates its data structures and the screen with the new information. The iToy Write Task updates its data structures and pushes the new information to the appropriate iThrottles.

Train Speed

(updated by Scott 04/01)

In this section, we will describe what happens in Controller when a train speed message is sent into the system. Refer to the collaboration diagram in Figure 39.

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP



Figure - Train Speed

1. A message to change a train’s speed may come off of the LocoNet, from the iThrottle, or from the keyboard. In any case, the message will get translated into a TrainSpeedMsg by the LocoBuffer Server. This message will contain the virtual address of the train and the speed value.
2. The LocoBuffer Server calls the the getVirtualAddress() function in the Slot Lookup Table in order to get the proper virtual address for the message.
3. The TrainSpeedMsg will be sent to the Command Queue Manager over TCP/IP via the socket connection. The Command Queue Manager will place the message on a Train Task’s queue. The Command Queue Manager knows which task’s queue to put it on by looking at the virtual address in the message itself.
4. The Train Task dequeues the message for processing. If the Train Task’s current state is moving, then update the Train Task’s speed variable and skip to #6.
   1. If the Train Task’s current state is halted and the speed message’s speed is greater than zero, then try to gain a reservation by calling the attemptReservation() function the Layout Manager. If the reservation is successful, update the Train Task’s speed variable. Skip to #5.
   2. If the Train Task’s current state is moving and the speed message’s speed is equal to zero, then the train enters a begin halt state and starts a timer. Once the timer has gone off then the train enters the halt state and the train task makes a call to dropReservation() function in the layout manager. Note that the train task should continue as usual and send the PhysTrainSpeedMsg as seen in #7.
   3. The Train Task will communicate to the File I/O protected type in order to write to the log file.
5. If a reservation was made during #4, then the Layout Manager will send a SectionReservedMsg to the Command Queue Manager. If a train was halted during #4, the Layout Manager will send a SectionFreeMsg to the Command Queue Manager once the train’s timer has gone off and the train makes the dropReservation() call. This will give any waiting Train Tasks the opportunity to reserve the section they’re waiting for, if the section is now free. The Layout Manager will communicate with the File I/O protected type in order to write to the log file.
6. If the Train Task’s speed variable was updated, then create a new PhysTrainSpeedMsg and send it to the Command Queue Manager. The Command Queue Manager will place this message on theScreen Write Task’s queue, the iToy Write Task’s queue, and will send it to the LocoBuffer Server via the Client Socket.
7. The Screen Write Task and iToy Write Task dequeue the message. The Screen Write Task updates its internal data structures. The iToy Write Task updates its internal data structures. The LocoBuffer Server receives this message via the Server Socket. The LocoBuffer Server translates the message into a LocoNet message addressed to the corresponding physical slot number.
8. The LocoBuffer Server will call the getPhySlot#() function in the Slot Lookup Table in order to get the proper physical train slot number that the LocoNet message should be addressed to.
9. The LocoBuffer Server writes the LocoNet message to the LocoBuffer. The iToy Write Task pushes the updated information to the appropriate iThrottles. The Screen Write Task updates the screen with the information.

Sensor Fire

(updated by Scott 04/01)

In this section, we will describe what happens in Controller when a sensor is fired. The system must determine which train set the sensor off, whether it was the front or end of the train, and if the system needs to make a reservation, do nothing, or halt the entire system. Refer to the collaboration diagram in Figure 40.

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP



Figure - Sensor Fire

1. When a sensor gets fired on the physical track layout, a LocoNet message comes into the LocoBuffer Server. The LocoBuffer Server translates that message to a SensorMsg. The SensorMsg only contains the ID of the sensor that was tripped.
2. The LocoBuffer Server sends the SensorMsg to the Command Queue Manager via sockets. The Command Queue Manager receives the message and places it on the Layout Task’s queue.
3. The Layout Task dequeues the message.
4. The Layout Task calls the sensorFired() function in the Layout Manager. The Layout Manager determines which train tripped the sensor and whether it was the front or the back of the train that tripped it. If it was the front, then it checks to see if the train has the next section reserved. If it does not, then we have reached an error condition because we don’t want a train entering a segment it doesn’t have reserved for safety reasons. In this case, see Halt Controller section below for how to react.
5. The Layout Manager communicates with the File I/O protected type in order to write to the log file to log which sensor was fired and what train did it.
6. The Layout Manager creates either a FrontSensorMsg if it is the front of the train that caused the sensor to fire **and** the sensor is in the *magnet over* state. Alternatively the layout manager sends a back BackSensorMsg if the was the rear of the train which caused the sensor to fire and the sensor is in the *magnet not over* state. Then message is sent to the Command Queue Manager who places it on the appropriate Train Task’s queue.

If the state of end of the train and the state of the sensor don’t correspond to the cases listed above then the state of the sensor is toggled and no messages are sent, this ends the sensor fired script.

For more specific information on determining which end of the train caused the sensor to fire and how to determine whether to send a message or not see the **analyzing sensor fired messages** section of the document.

* 1. The Train Task dequeues the FrontSensorMsg or BackSensorMsg.

If it was the back sensor that went off, then it updates its current back sensor with the one passed to it, but retains a temporary copy of the old sensor. If it was in the halted, or waiting state then the system is in an error state and it sends an ErrorMsg to the Command Queue Manager (see Controller Halt section below)**.** If the train is in the moving, begin halt, or begin wait state then we may continue.

If it was the front sensor that went off then the train updates its current front sensor with the one passed to it. If it is the begin wait, waiting, or halt state then it sends an ErrorMsg to the Command Queue Manager (see Controller Halt section below). If it is in the begin halt or moving state then the system is in a non-error state and may proceed.

* 1. If the train received a FrontSensorMsg and the train is in either the begin halt or moving state then in calls the occupySeg function in the layout manager and passes the trainID and the new front sensor.
  2. If the train received a BackSensorMsg then it calls the freeSeg function in the layout manager and passes its old back sensor ID and its trainID and the layout manager frees the end segment of this train.
  3. The train task makes an attemptReservation() function call into the Layout Manager if it was the train received a FrontSensorMsg and the train is in the moving state. The train passes its virtual address and the new front sensor in the function call. If it can reserve the next segment then the function returns with the section ID that just got reserved and the train continues moving. Otherwise it returns -1 and the train enters the begin wait state.
  4. The layout sends either a segment SectionFreeMsg, SectionBlockMsg or a SectionReserveMsg depending on what messages it receives and what it is able to perform.
  5. If the train asks for a reservation and doesn’t get it then the train needs to change its speed to 0 and send out a PhysTrainStopMsg with its trainID.
  6. The Layout manager updates the log file by making the updatelog function call in the File I/O task. It passes in the a string containing what occurred (segment freed, segment occupied, segment reserved, whether they were successful and the train that made the calls).
  7. If the train was not able to get the reservation, then the Command Queue Manager sends the PhysTrainStopMsg from #9b to the Client Socket which will relay it to the LocoBuffer Server.
  8. The iToy Write Task may get PhysTrainStopMsg message sent in #9b and also the messages that the layout manager sent (SectionFreeMsg, SectionBlockMsg, SectionReserveMsg) and updates its data structures accordingly. It also pushes the new information to the appropriate iThrottles.
  9. The Screen Write Task may get the PhysTrainStopMsg message sent in #9b and also the messages that the layout manager sent (SectionFreeMsg, SectionBlockMsg, SectionReserveMsg) and updates its data structures accordingly and displays the new information to the screen.

Update Switch from Throttle

(updated by Scott 04/01)

In this section, we will describe what happens in Controller when a switch is updated from a throttle. Refer to the collaboration diagram in Figure 41.

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP.



Figure - Update Switch from Throttle

1. The command to update a switch will come from either the LocoNet, iThrottle, or keyboard. In any case, the command will be translated into a UpdateNextSwitchMsg.
2. If the command came from the LocoBuffer Server, then a getVirtualAddr() function call needs to be made to the Slot Lookup Table in order to find out what virtual address to translate the message to.
3. The UpdateNextSwitchMsg is sent to the Command Queue Manager via TCP/IP using the sockets. The Command Queue Manager places the message on the appropriate Train Task’s queue.
4. The Train Task deqeues the message.
5. The Train Task calls the throwNextSwitch() function located in the Layout Manager. The Layout Manager determines which switch needs to be closed/thrown, and whether the switch can be closed/thrown at that time. The switch cannot be closed/thrown if any of the sections in which the switch is located in are in a non-free state. Layout Manager communicates with the File I/O protected type to write out to the log file.
6. If the switch was able to be closed/thrown, the Layout Manager produces a PhysicalSwitchCloseMsg or PhysicalSwitchThrowMsg and sends it to the Command Queue Manager. The Command Queue Manager places this message on the LocoBuffer Write Task’s queue, the iToy Write Task’s queue, and the Screen Write Task’s queue.
7. The iToy Write Task and Screen Write Task dequeue the message. The Command Queue Manager sends the LocoBuffer Server the message via the Client Socket.
8. The LocoBuffer Server calls the getPhysSlot#() function located in the Slot Lookup Table. This will provide the LocoBuffer Server with the correct physical train slot number to send the LocoNet message to.
9. The LocoBuffer Server sends the LocoNet message through the LocoBuffer which updates the physical switch on the track layout. The iToy Write Task updates its data structures and pushes the new data to the appropriate iThrottles. The Screen Write Task updates its data structures and displays the new information on the screen.

Switch Finish Move

(updated by Scott 04/01)

In this section, we will describe what happens in Controller when a physical switch has finished moving. Refer to the collaboration diagram in Figure 42.

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP



Figure - Switch Finish Move

1. A message comes off the LocoNet that contain the ID of the switch that has finished moving and what state the switch is in.
2. The LocoBuffer Server converts the message into a SwitchMoveDoneMsg and sends it to the Command Queue Manager via the Server Socket. The Command Queue Manager gets the message from the Client Socket and places it on the Layout Task’s queue, the Screen Write Task’s queue, and the iToy Write Task’s queue.
3. The Layout Task dequeues the message and calls the switchFinishedMoving() function located in the Layout Manager. The Layout Manager locates the switch in its data structures and updates it accordingly.
4. The Layout Manager communicates with the File I/O protected type in order to write to the log file.
5. The Screen Write Task and iToy Write Task deqeue the message sent to them from #2.
6. The Screen Write Task updates its internal data structures and writes the updated information to the screen. The iToy Write Task updates its internal data structures and pushes the information the appropriate iThrottles.

Save Controller State

In this section, we will describe what happens in Controller when the user chooses to save the current state of Controller. Refer to the collaboration diagram in Figure 43

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP



Figure - Saving Controller State

1. First the Keyboard Read Task reads the command from the user and determines that the user requested for the system to be saved. It then sends a SaveTrainMsg to the Command Queue Manager containing the filename that the user wishes to save the state of the track to.
2. The Layout Task then dequeues this message and makes a saveState() function call to the Layout Manager, passing in the filename.
3. The Layout Manager then calls the saveLayout() function in the File I/O protected type, passing in the data it wishes to save and the filename the data should be saved to.
4. Upon getting the saveLayout() function call the File I/O writes the data in standard save XML format to the filename indicated.
5. After the Layout Manager makes the saveLayout() function call it sends a SaveTrainMsg to the Command Queue Manager with the filename that the user wanted the date saved to. The Command Queue Manager then puts the message onto each of the trains queues.
6. Each Train Task then dequeues the message.
7. The Train Tasks make the returnListOfSensors() function call to the Layout Manager, passing in the train’s front sensor as a parameter. The Layout Manager then steps through the sections from the start sensor to the end sensor occupied by the given virtual train address. It then returns this list of sensors to the train in front to back order.
8. The Train Tasks then makes a saveTrain() function call into the File I/O protected type, passing in the list of sensors returned from the returnListOfSensors() function call and the filename the data should be saved to. The File I/O protected type then writes the data to the restore file with given filename in the correct XML format.
9. The Train Task then sends a saveResultMsg to the Command Queue Manager which then puts this on the Screen Write Task’s queue.
10. The Screen Write Task then dequeues this message and writes to the screen the result of the save command along with the filename which the data was saved to.

Restore Controller State

(updated by Scott 04/01)

In this section, we will describe what happens in Controller when the user chooses to restore Controller’s state from a previously saved version. This can only occur during startup if a user chooses to restore Controller’s state instead parsing the layout from an XML file. Refer to the collaboration diagram in Figure 44.

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP



Figure - Restoring Controller's State

1. When Controller is first launched via a separate executable file which has the responsibility of starting all the necessary protected types and tasks. First instances of the protected types (Command Queue Manager, File I/O, Slot Lookup Table and Layout Manager) are started. We start these first because the tasks will need references to these objects. Next, the tasks are started. When instantiating a task, we can pass in the references to the protected types that they need access to. For example, when the Layout Task is started, it is passed references to the Layout Manager, Slot Lookup Table and Command Queue Manager because these are all the protected types that this task must talk to.
2. The Keyboard Read Task reads the string that the user entered. It will contain either an ‘x’ if they want to start the system from an XML or ‘r’ if they want to read from a restore file. See the **Start system** section if you wish to see the XML collaboration diagram. From here on out we assume the user entered R. The system then sends a restore layout message which include the filename the user entered where they want to restore the system from
3. Layout task then dequeues this message
4. The layout task then calls the restore function located within the layout manager and contains the filename that user wishes to restore from
5. The layout manager then calls the restore function within the file I/O which returns then opens the file which the layout manager passes to it and returns the contents of the file in a string. Which the layout manager then parses
6. The layout manager then sends either a parse failed message if the restore failed because it wasn’t in the correct format or the file didn’t exist. Or the layout manager sends a restore train message if it correctly parsed the file and it is ready for the trains to be restored. As the layout manager is reading it file and getting the switches it reads in the status of the switches and sends out either a throw switch message or close switch message based on what it read from the file
   1. The LocoBuffer Server gets the message from the TCP/IP connection using the Server Socket. It converts the message and sends it out to the LocoBuffer.
   2. If the file was incorrect or the layout of the restore file was incorrect then the screen write tasks dequeues the parse failed message and informs the user that the restore failed
   3. If the restore of the layout was successful then the layout task dequeues the restore train message which contains the file to restore from.
7. The layout task then calls the file I/O function restore which returns the contents of the given filename in a string
8. The layout task then parses the restore file and spawns off trains as the file indicates using the spawnTrain function in the train package. It then follows the sequence given in register train for registering trains. **See register train collaboration diagram.**
9. Once the trains have been registered in the system and occupied in the layout manager the layout task sends a parse result message to the command queue manager with the true if the train parse succeeded and false if the train parse failed and it sends the filename.
10. The screen write task then dequeues the parse result message and informs the user of the result of the parse.

Halt Controller

(updated by Scott 04/01)

In this section, we will describe what happens in Controller when the entire system must stop. This occurs when an error condition is reached. Refer to the collaboration diagram in Figure 45.

Martin April 1: Update to reflect that access to the keyboard/screen & LocoBuffer & iToys are via TCP/IP



Figure - Halting Controller

* 1. The layout manager makes an update log function call into the file I/O protected type and passes in the string containing the error that occurred with as much information as possible as the cause and description of the problem
  2. The layout manager then sends an ErrorMsg to the command queue manager with the id of the sensor which fire which we didn’t expect to fire. Which then puts this onto the queues of all the trains queues and the screen write queue.
  3. the trains then dequeue this message and sets its speed to 0
  4. the screen write task then dequeues this message
  5. The train tasks then sends physical speed change message to the command queue manager with a speed of 0 and its train ID. The command queue manager then puts this message onto the locobuffer write task.
  6. The screen write task then writes to the screen to inform the user that the system has entered an unsafe state and has stopped all the trains and also prints the sensor which fired that caused the unsafe state.

1. The LocoBuffer Server gets the message from the TCP/IP connection using the sockets. It puts messages onto the LocoNet to stop all the trains.

Log File

Controller uses a log file for debugging purposes. An entry should be made in the log file whenever the state of the train system changes. Examples of such state changes include the following:

* A train changes speed.
* A train changes direction.
* A train enters a new section.
* A train leaves a section.
* A train makes a new reservation.
* A train enters a waiting state.
* A switch is closed or thrown.
* The XML file or restore file are loaded.
* The state of the system has been saved to a restore file. the train entering a new section, a train entering a waiting state, a switch being closed/thrown, etc.

This list is by no means exhaustive, but rather should provide an understanding of the types of events that should be logged.

Each new entry in the log file should be added to a new line. At the beginning of each entry into the log file should be a timestamp. Each entry should include useful debugging information such as virtual train addresses, sensor IDs, switch IDs, train states, switch states, etc. Each entry should only include the information that is important to that event. For example, an entry in the log file about a switch being thrown should not include a virtual train address.

The log file can be written to via the File I/O protected type. We need a protected type to write to the log file, because multiple Controller objects may wish to write to the log file at the same time. For example, two Train Tasks may wish to write to the log file at the same time if their corresponding trains each enter new sections at the same time. If we allowed both of these tasks to write to the log file at the same time, then the log file would contain incoherent information.

Acceptance Tests

## Head-On Collision Test

1. Set two trains on the tack such that they’re a few sections apart and facing toward each other.
2. Start each train in the forward direction at the same time.
3. Monitor the trains as they enter and leave sections.
4. Both trains should automatically stop before they run into each other.
5. Once the trains have stopped, reverse the direction of both trains.
6. The trains should continue moving away from each other.

## Trailing Test

1. Set up two trains facing the same direction in adjacent sections.
2. Increase the speed of the first train to .
3. Increase the speed of the second train to .
4. Monitor the trains as they enter and leave sections.
5. The first train should continue moving unimpeded, assuming the switches along its path are set properly and there are no other trains in front of it. The second train should bounce back between moving and waiting. This is because the second train will only be able to continue moving when the first train has left a section. The second train will then speed forward, but will be forced to wait again until the first train leaves the next section.

## Track Crossing Test

1. Move a train onto a section which crosses with another section.
2. Move a second train so that it will attempt to enter the section that is being blocked.
3. Once the second train enters the section adjacent to the blocked section, it should enter the waiting state since it should not be able to get a reservation on the blocked section.

## Switch Test

1. Move a train towards a switch from the wide end, where the train should only be able to cross the switch if the switch is thrown.
2. Set the switch to closed.
3. The train should stop and enter the waiting state once it enters the section adjacent to the section with the switch in it.
4. Set the switch to thrown.
5. The train should automatically continue moving.

## Train Direction Test

1. Set a switch to be closed.
2. Move a train through the switch from the narrow end.
3. Once the back of the train has left the section with the switch in it, halt the train.
4. Set the switch to be thrown.
5. Reverse the direction of the train and give it some speed. The train should not move, but rather it should enter the waiting state.
6. Close the switch. The train should continue moving in the reverse direction.

This test allows us to not only make sure a train can switch direction, but also that Controller will still know the correct location of a train once it has switched direction.

## Yard Test

1. Have train in the back section of the yard (furthest from the way out to the rest of the track)
2. Position the switches so that they are all face the closed direction except the one closest to the rest of the track and the one closest to the train (this means that all switches should be in the correct direction to get the train out except the one closest to the end and the one closest the train)
3. Set the train to a positive speed in the forward direction
4. It should not move and should go into the waiting state
5. Have the train flip its next switch
6. The switch closest to the train should flip and the train should still not move
7. Use the keyboard to flip the final switch and the train should automatically resume at the speed you set earlier

## Update Switch Underneath Train Test

1. Position one train such that it is sitting on top of a switch
2. Position another train behind it where the next switch is the one the first train is sitting on
3. Use the second train to try to flip its next switch
4. The switch should not move
5. Use the keyboard to try to flip that particular switch
6. The switch should not move
7. Move the first train out of the switch segment and try to flip the second trains next switch, it should work now

## Halting Controller Test

1. Have several trains moving over the track
2. Place a magnet over a random sensor located nowhere near the trains
3. All trains should come to a halt and a message should appear one the screen indicating which sensor went off and that the entire system should halt.

## Save and Restore Test

1. Have the several trains positioned across the track
2. Use the save function to store the state of the track
3. Power off the system
4. Power back on and load from the restore file
5. Screen should load up with the old train numbers and they are located in the same location as when saved
6. Use the keyboard to throw a switch in front of one of the trains.
7. It should be the correct one for the trains old location
8. Use the throttle to get another train moving
9. The train should move in the direction it previously was going

## Attempt Update Switch in Reserved Section Test

1. Position a train so it is moving and the next segment contains a switch
2. Tell the train to flip its next switch
3. It should flip one the past the one that is in the reserved segment

iThrottle

## Overview

An iThrottle is a virtual throttle device that is run as an app on either an iPhone, iPod Touch, or iPad. This idea is extensible to other devices such as Android, but we won’t worry about that for our purposes here.

An iThrottle will communicate with Controller via a TCP/IP connection. Controller will act as the server for this connection and each iThrottle will act as the client. It is important to note that an iThrottle is completely disjoint from the LocoNet. This means that an iThrottle need not understand LocoNet messages. This is completely different in comparison to physical throttles which can only talk to Controller via the LocoNet.

## User Interface

The user interface consists of four different screens. Three of these are the main screens. These are the throttle screen, location screen, and switches screen. The fourth is an auxiliary screen used to register trains. One can navigate through the three main screens via buttons in a menu at the top of the app. One may also swipe their finger horizontally across the screen to navigate between the screens. We will describe each of the screens in more detail below.

### Top Menu

The menu along the top of the app consists of three buttons, the current engine number selected, and a register/unregister train button. Illustrations of the top menu can be seen in Figure 46 and Figure 47.

|  |  |
| --- | --- |
| C:\Users\schmals\Desktop\adarail_app\mock ups\v4\top_menu_01.png  Figure - Top Menu with Registered Train | C:\Users\schmals\Desktop\adarail_app\mock ups\v4\top_menu_02.png  Figure - Top Menu with no Registered Train |

In Figure 46 we see the top menu when a train is registered with the throttle. We know a train is registered because we are told the iThrottle is tied to the train with engine number 1204. We may use the “Unregister” button to disconnect the iThrottle from the train. In Figure 47 we see that the iThrottle is not reigistered to any train. We may use the “Register” button in order to tie the iThrottle to a train. Clicking this button will take the user to the place train screen described below.

In these images we can also see the three options a user can choose from to navigate among the main screens. The “Throttle” option is currently selected in these images.

### Throttle Screen

The throttle screen is responsible for allowing a user to change the state of the train associated with the iThrottle. An illustration of this screen can be seen in Figure 48.

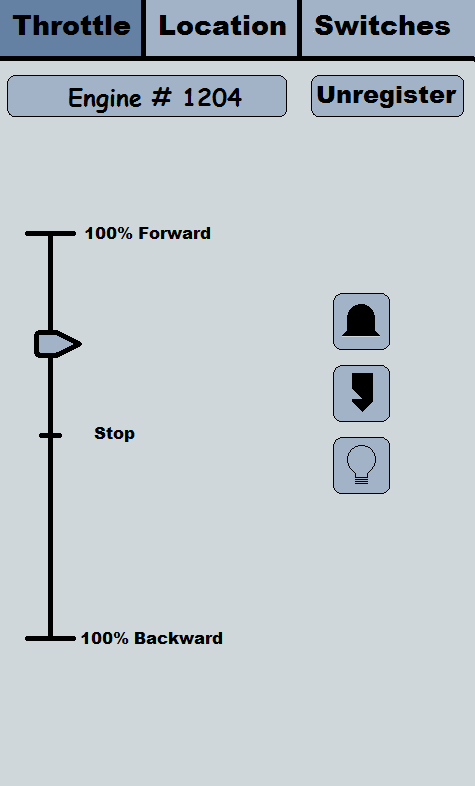


Figure - Throttle Screen

From this screen the user is able to change the speed and direction of the train using a slidebar located along the left. By tapping the slide bar twice, the train will be placed in the stop state.

The user may also toggle the horn, toggle the light, or toot the horn of the train using the three buttons along the right.

### Location Screen

The location screen is responsible for reporting to the user where the train is currently located. An illustration of this screen can be seen in Figure 49 and Figure 50.

|  |  |
| --- | --- |
| C:\Users\schmals\Desktop\adarail_app\mock ups\v4\locations_ui_01.png  Figure - Location Screen | C:\Users\schmals\Desktop\adarail_app\mock ups\v4\locations_ui_02.png  Figure - Location Screen of Lost Train |

In Figure 49, we see the location screen of a train that knows where it’s located. A train’s location is specified by a list of sensors. This is the list of sensors that locates a train. For an explanation of what the term “locating a train” means, refer to this document’s glossary. As we can see, the sensors that locate the train should be listed from front to back.

In Figure 50, we see the location screen of a train that has lost its location. This occur if Controller enters an error state. In this situation, the user must click the “Place Train” button at the bottom of the screen to manually specify the location of the train once again. Clicking this button will navigate the user the place train screen described below.

### Switches Screen

The switches screen is responsible for allowing the engineer of a train to update the states of switches in front of it. An illustration of this screen can be seen in Figure 51 and Figure 52.

|  |  |
| --- | --- |
| C:\Users\schmals\Desktop\adarail_app\mock ups\v4\switches_ui_01.png  Figure - Switches Screen #1 | C:\Users\schmals\Desktop\adarail_app\mock ups\v4\switches_ui_02.png  Figure - Switches Screen #2 |

The switches screen displays the switches located in its direction of travel one-by-one. First, the screen will only display the first switch in front of the train and an unclickable arrow button below. We will not list sections located in sections the train has reserved. Two radio buttons represent the stat that the switch may be placed in. These radio buttons represent the closed and thrown states. Initially, neither of these radio buttons will be selected. The user must make a selection. Upon making a selection, the arrow button underneath becomes clickable. When the user clicks the radio button, the next switch along the train’s direction of travel is displayed. Again, neither radio button is selected by default, and the user must make the selection before they are able to click the arrow button below to display the next switch.

So far in our description, we’ve been considering the case when a train is approaching the switch from the narrow end. In this situation, the engineer makes the choice of which direction the set the switch. If the train is approaching a switch from the wide end, then the user does not have a choice. There can only be one valid way the switch must be set in order for the train to pass. For these switches, only the radio button for the proper path is displayed and it is automatically filled in. Switch 32 in the figures above illustrates this situation.

Once a user has made the decisions on all the switches they would like to, then they may hit the “Commit” button. This will send all the switch close/throw messages to Controller. Controller will process the messages update the switches on the physical layout, if appropriate.

### Place Train Screen

The place train screen is responsible for manually specifying the location of a train on the track. An illustration of this screen can be seen in Figure 53 and Figure 54.

|  |  |
| --- | --- |
| C:\Users\schmals\Desktop\adarail_app\mock ups\v4\place_ui_02.png  Figure - Place Train #1 | C:\Users\schmals\Desktop\adarail_app\mock ups\v4\place_ui_01.png  Figure - Place Train #2 |

There are only two ways that a user can find themselves on this page. The first is if they have selected the “Register” button along the top of the app, if a train wasn’t already tied to the iThrottle. The second is if the user clicks the “Place Train” button located on the location screen. In this situation, the engine number of the train should be automatically filed in.

In order to place a train, the user must make sure a physical train address is entered under the “Engine #” textbox. The user may then start entering the list of sensor numbers that locate the train. This list of sensor numbers should be entered front to back. If you’re unfamiliar with what the term “locate a train” means, refer to the glossary of this document. As the user enters the sensor numbers, the iThrottle will automatically make suggestions for the next sensor number. We can see this in Figure 54. The user can tap the screen to accept the suggestion, or may continue typing to reject the suggestion and continue manually entering the next sensor number. Once the trains has been placed, the user can click the “Submit” button which will send the information to Controller so that the train may be registered with the system. In the case that an error occurs and the train cannot be registered, the place train screen is displayed again with an error message.

## Communicating with Controller

As previously mentioned, the iThrottles will be communicating with Controller via a TCP/IP connection. Since data sent across a TCP/IP connection is represented as a stream, there will need to be some sort of protocol or paradigm that is set up which will provide structure and order to this information. This will allow both Controller and the iThrottles to understand the information being sent across the network.

## Integration with Controller

Throughout this document, we have included the iToy Read Task and iToy Write Task in our discussion of the Controller design. These will be the two tasks responsible for interfacing with the iThrottle. As you may deduce, the iToy Read Task is responsible for reading data sent by the iThrottle, converting it into internal data messages, and sending it to the Command Queue Manager. The iToy Write Task is responsible for receiving messages on a queue from the Command Queue Mananger and sending message to the iThrotttle.

Similar to the Screen Write Task, the iToy Write Task should include a cache of data which represents the current state of the layout. This cache of data does not need to be nearly as extensive as the data contained in the Layout Manager. This data should only be able to answer simple questions like what the current state of a specific switch is, where a train is currently located, etc. This data will be updated whenever the iToy Write Task receives messages in its queue from the Command Queue Manager. By including this cache of data structures, the iThrottle will be able to get answers quick when requesting information. To illustrate this concept, we can consider what might happen when an iThrottle requests which switch comes next along a specified path. The request will come into Controller via the iToy Read Task. The iToy Read Task will send a message to the Command Queue Manager who will place the message on the iToy Write Task’s queue. The iToy Write Task will dequeue this message, reference it’s cache of data structures and then send the answer back out to the iThrottle. If we didn’t have a cache of data structures, then Controller would have to bring the Layout Manager into this mix in order to find the answer to the question, and that would increase congestion in the system and slow down the response time.

## Notes on the iThrottle

Unfortunately, this iThrottle design is nowhere near exhaustive in any way shape or form. It is meant to only give an overall high-level understanding of what the iThrottle should do, and how it can interface with Controller. Originally this design was meant to be must move extensive, but unforeseen circumstances during the CSCI 492 course have caused the priorities of the designer to be shifted onto the design of Controller instead of the design of this iThrottle.

Glossary

**Section** – A section is the area of track located between two adjacent sensors.

**Sensor** – A sensor is a device on the track layout that is triggered by magnets.

**Switch** – A switch (aka turnout) can be thought of as a split in the railroad track. There is a narrow end, and a wide end of the switch. The wide end is where the track diverges. See Figure 55.

**Closed** – A switch is closed if the it’s set in the close position. See Figure 55.

**Thrown** – A switch is thrown if it’s set in the throw position. See Figure 55.

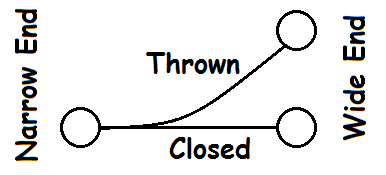


Figure - Switch

**Throttle** – A physical device used by an engineer to control a single train. For our purposes, we will be using the UT4R throttle. Refer to the section of this document titled “Hardware Components and Context Diagram” for a more in-depth description of this throttle.

**iThrottle** – A virtual throttle that is used by an engineer to control a single train. Refer to the section of this document titled “iThrottle” for a more in-depth description of this throttle.

**Physical Train Address** – The physical train address is the address of the actual physical train that is registered with the DCS-200. Generally, the address of each train is the number printed on the side of the locomotive.

**Physical Train Slot Number** – The slot number in the DCS-200 that corresponds to a physical train address. Throttles and Controller communicate with the DCS-200 via LocoNet messages that are addressed to slot numbers. The DCS-200 converts these slot numbers into physical addresses.

**Virtual Address** – The address that a throttle is set to. Controller will dynamically produce these addresses whenever a new train is registered with the system. These addresses do not correspond to a physical train.

**Virtual Slot Number** – The slot number that is associated with a virtual address. A throttle is sent to a virtual address which causes the throttle to place messages on the LocoNet that are addressed to a virtual slot number. Controller then picks up these messages and processes them.

**Internal Message** – These are the messages sent among the different software components of Controller. They are routed by the Command Queue Manager and placed on different tasks queues.

**LocoNet Message** – These are the messages on the LocoNet of the train system. These are the types of messages that Controller processes from the LocoBuffer. Controller also sends LocoNet messages out when we wish to make something happen to a physical train or on the physical layout.

**Yard** – The yard of a railroad can conceptually be thought of as a train parking lot. The yard is the area on a train system that trains typically go to park.

**Crossover** – A crossover is a section of track with two switches that can be set in such a way that a train moves from one piece of track to another parallel piece of track. In our system, both switches that makeup a crossover have the same address. See Figure 56 for a visual representation of a crossover. In this image, the red dots represent the two switches.

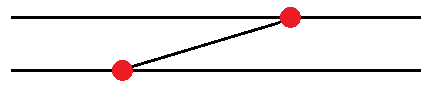
****

Figure - Crossover

**Moving** – A train is moving if its speed is non-zero and it isn’t in a waiting or begin waiting state. Typically, if the physical train is moving on the layout, then the train is in a moving state. Typically, if the physical train on the physical layout is moving, then the train should be in a moving state.

**Halted** – A train is halted if the engineer has willingly set the train’s speed to zero and the physical train is no longer moving.

**Begin Halted** – A train is in the begin halted state if the engineer has willingly set the train’s speed to zero, but the physical train hasn’t quite fully stopped yet.

**Waiting** – A train is waiting if the train would like to continue moving, but cannot because it was unable to get a reservation on the next section in the direction of travel. Also, the physical train must not be moving for it to be in a waiting state.

**Begin Waiting** – The only difference between a train in the waiting state and a train in the begin waiting state is that the physical train in the begin waiting state hasn’t quite come to a full stop yet.

**Blocked** – A section is blocked if an overlapping section is either reserved or occupied.

**Occupied** – A section is occupied if any portion of a train is located within the section.

**Reserved** – A train may reserve a section. A train can reserve a section if it’s located in an adjacent section and the train will enter that section if it continues in its current direction of travel. No other trains may reserve or occupy this section until the section has become free again.

**Free** – A section is free if it’s not in the blocked, occupied, or reserved state. A train may reserve the section if it’s free.

**Located** – We use the phrase “located” in regards to describing a train’s location. We’ll say a train is located by sensors 5, 10, 20, 17 if the train is sitting on top of sensors 10 and 20, and the train is surrounded by sensors 5 and 17. See Figure 57 for a visual representation of a train located by these sensors. Note that we list the sensors in order from front to back.

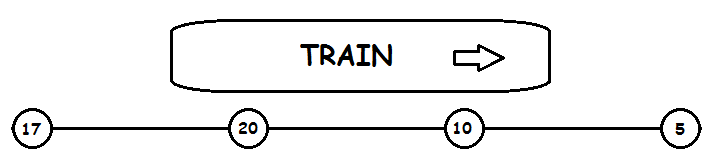


Figure - Train Located by Sensors

Additional Resources

Martin April 1: I need to fix these links.

|  |  |
| --- | --- |
| [1] | For more information on the TC64s, refer to <http://faculty.cs.wwu.edu/martin/cs492/Railroad/W10%20Design%20Document/TC-64-manual.pdf> |
| [2] | For more information on the DS64s, refer to <http://faculty.cs.wwu.edu/martin/cs492/Railroad/W10%20Design%20Document/ds64V4.pdf> |
| [3] | For more information on the Tortoises, refer to <http://www.circuitron.com/index_files/Tortoise.htm> |
| [4] | For more information on the LocoBuffer, refer to <http://faculty.cs.wwu.edu/martin/cs492/Railroad/W10%20Design%20Document/LB-USB-manual.pdf> |
| [5] | For more information on the LocoNet messages, refer to <http://faculty.cs.wwu.edu/martin/cs492/Railroad/W10%20Design%20Document/loconetpersonal.pdf> |
| [6] | Cody Baxter’s Operations Manual (link on Osborne’s site broken) |
| [7] | Controller Requirements Analysis Document (<http://faculty.cs.wwu.edu/martin/cs492/Railroad/Analysis/ada_rail_analysis.htm>) |
| [8] | Parts List (<http://faculty.cs.wwu.edu/martin/cs492/Railroad/W10%20Design%20Document/Parts_List.xls>) |
| [9] | Train Layout (<http://faculty.cs.wwu.edu/martin/cs492/Railroad/W10%20Design%20Document/Layout.ppt>) |
| [10] | Digitrax Information (<http://faculty.cs.wwu.edu/martin/cs492/Railroad/W10%20Design%20Document/Digitrax.zip>) |
| [11] | Railroad Circuit Kit Information (<http://faculty.cs.wwu.edu/martin/cs492/Railroad/W10%20Design%20Document/RR-CirKits.zip>) |