Concordia UNIVERSITY

Bookstore

EMBEDDED SYSTEMS AND SOFTWARE DESIGN

LABORATORY MANUAL

COEN – 421

Winter 2008

PUBLISHED UNDER THE CONCORDIA UNIVERSITY BOOKSTORE
CUSTOM PUBLISHING PROGRAM

SGW Campus
J. W. McConnell Building
1400 de Maisonneuve Blvd. W.
848-2424 x 3615

Preface

This lab manual has been designed for COEN 421 - Embedded Systems Software Design, and used in the ECE Real-time Systems Laboratory. This laboratory is equipped with several systems including development stations, target systems; all connected through a Local Area Network. The development stations are desktop machines running QNX and mounting various file systems from ENCS servers.
This manual contains five experiments designed for COEN 421 students. The main purposes of these experiments are:

- To provide a practical experience in working on a RTOS platform.
- To improve student C/C++ and reusability skills in real-time software design.
- To improve student algorithm design skills.
- To gain experience with Integrated Development Environments.
- To practice and improve student communications skills.

Mr. Dan Li and Mr. Gao Bo have maintained and improved this manual based on new hardware settings in collaboration with Dr. F. Khendek who is currently teaching COEN 421. We would like to thank all the members of the ECE Department who have contributed to the development of the Real-time Systems Laboratory and to this manual, especially Dr. M. Mehmet Ali , Dr. P. Sinha, D. Chu and G**.** Gosselin.

CONTENTS

Figures and Source codes

1 INTRODUCTION

1.1 Laboratory Rules

Considering the large number of students attending the lab and in order for the lab to operate properly, the students are asked to abide by the following rules:

- ❖ No eating or drinking is permitted in the laboratory.
- ❖ Overcoats and briefcases are not permitted in the laboratory.
- ❖ Students are supposed to sign in and sign out whenever they enter and leave the laboratory.
- ❖ Students should bring their own laboratory manual. Any student who is more than 30 minutes late will not be permitted into the laboratory.
- ❖ The locomotives (trains) used in the experiments are very expensive and hence proper care must be taken so that incidents like train collision and train bumping do not occur.
- ❖ Constant rebooting of the system and dummy terminal should be avoided.
- ❖ No locomotives should be exchanged from one bench to another.
- ❖ Upon entering and leaving the laboratory, students should check whether the locomotives are in proper working condition or not.
- ❖ Maximum care should be given to the train board and the single board computer.
- ❖ Students should immediately report to the demonstrator if any of these are damaged or missing on their workbench. Failing to do so will result in students being charged for damages or losses.

1.2 Scope of Real Time Systems Laboratory

The main purposes of real time systems laboratory are as follows:

1. To provide a practical experience working on a real time platform.

2. To improve C++ and reusability skills. For the codes to be reusable for other project components develop a POSIX compliant set of classes for pthread, semaphores and timers. Some of the basic pthread functionalities are:

- Creating and initializing a thread
- Cancelling or exiting a thread
- Getting a thread ID
- Yielding or relinquishing a processor
- Joining a thread
- Detaching a thread
- Similarly, Semaphores (using pthread_mutex and counter) and Timers have to be designed which would provide some basic functionality.

3. To improve syntax and algorithm checking.

4. To learn QNX and the Integrated Development Environment (IDE).

5. To provide experience in report writing.

1.3 Organization of this Manual

This manual provides a practical example in real-time programming and an overview of the various experiments that are to be performed by students in the real-time lab.

Each experiment is divided into the following sections:

a) Objectives
b) Track diagram
c) Lab description
d) Class hierarchy diagram
e) Design issues
f) Hints and Sample code
g) Check points

The first part gives the objectives of the experiment.

The second part gives a rough idea about the experiment in the form of a diagram.

Next comes the theoretical part of the experiment. The theory in this manual contains only a brief experimental procedure and so students are asked to write their own descriptive procedure in their lab reports.

The next section deals with the class hierarchy diagram. This is again a rough idea for students to get a good understanding of the programming they are supposed to do. The class hierarchy diagram provided in the manual is only a suggestion, and its up to the students to come up with their own hierarchy diagram in their lab reports for each one of the experiments.

The last two sections provide a brief overview of the design issues, hints and sample code as part of the different experiments.

Finally the experiment concludes with some questions as checkpoints.

1.4 Execution of the Experiments

Each experiment must be studied in advance. All the classes and functions needed for the programming of a particular experiment must be decided well in advance. The basic classes that are to be used in experiments# 2, 3, 4 & 5 like basic PThread, Mutex, Semaphore, Timer, RWlock should have been done as part of programming assignment before the start of the actual experiments.

Since the laboratory represents a significant portion of the student's practical training, it is imperative that the students perform all the experiments. If a student has missed an experiment due to circumstances entirely beyond his/her control, that student will have the opportunity to perform it at the end of the term. Any student who misses more than one experiment will not be eligible for any form of passing grade ("R" grade).

1.5 Lab Report

For each experiment a lab report must be written which can be regarded as a record of all activities, observations and actual coding pertaining to that experiment. Lab report should be well organized and well presented and should contain as much information as possible.

TEMPLATE FOR LAB REPORTS

Page 1 - Cover Page
Include: Lab number, Lab Title, Student name(s), Student ID(s), Team ID, Workstation ID, Due date.

Page 2 - Contributions by each member
In one page (about 1 paragraph per person), explain what work each team member accomplished.

Pages 3 to n, (where n <= 15)
1. Objective:
In two or three sentences, state the objectives of the lab experiment.
2. Introduction:
Introduce the experiment in three parts:

2.1 Background:
Provide any necessary background information about the experiment so that someone outside of the class, who hasn't read your previous work, can understand your context.

2.2 Approach:
Give an overview of how you intend on meeting the objectives stated in section 1.

2.3 Requirements:
In your own words (i.e. don't just copy from the lab manual), state what the requirements of the experiment are. Prioritize them.

3. Analysis:
Write a page or two discussing what to do exactly. Detail your understanding of the requirements, or how you interpret them. What does your interpretation imply? Use-Case Diagrams (with natural language descriptions) and scenarios may help.

4. Design:
Write two or three pages discussing how you plan to solve the problem. Describe any algorithms or important procedures. A detailed Class Diagram (with attributes and methods), Interaction Diagrams, State Diagrams, or Activity Diagrams may be necessary to explain your design.

5. Implementation:
In a tabular format, say what was implemented, what was partly implemented, and what was not implemented. Also state if you were able to test each item, and if it passed or failed the test. It would be a good idea to describe the test too. Your requirements listed in the table will come from section 2.3 and section 3 of your report (e.g. Use-Cases). For example:

Requirements	Implemented	Partly Implemented	Not Implemented	Tested
Accelerate Train	X			Yes
Decelerate Train	X			Yes
Stop Train at a particular Track Segment		X		No
Monitor Train Speed			X	N/A

You should also discuss any problems encountered, or why you were not able to implement features to satisfy all the requirements of the experiment. State whether these problems are solvable and how?

6. Future Work:
In one or two paragraphs, describe what your future project plans are and how the work accomplished in this experiment fits into the picture.

7. Answers to Lab Manual Questions:
Answer the questions in the lab manual here.

8. Lessons Learned:
What did you learn? What would you do differently next time?

9. Conclusion:
Briefly summarize your results (successes and otherwise). Highlight what was the most important concept(s) or issue(s) that you dealt with. Give a brief statement related to future work.

1.6 Introduction Of Real Time System Lab

Overview

The lab is made up of QNX develop workstation and some small board computers (SMC) as the targets running QNX and mounting various file systems from ENCS servers.
In the lab you can access to the outside world by using web browser and you can access your home by using the tool ssh.
For the purpose of reporting any problem, please send an email to: danli@encs.concordia.ca or helpdesk@ece.concordia.ca.

You can also remotely access the QNX system from outside Concordia.ca by using secure ssh. There are two QNX remote-access servers installed.
For Linux users:

1) ssh login.encs.concordia.ca
2) ssh mackay.realtime.private or ssh peel.realtime.private
   If you are working in ENCS network, you just run the second step only.

For Window users, you may have to install a SSH client, such as Secure SSH client. Once you have a SSH client, you can access the encs.login server:

login.encs.concordia.ca

then you can use 'ssh mackay.realtime.private' or 'ssh peel.realtime.private'. Once you remotely log in QNX, you can't run an application with GUI.

Getting Started

At the lab, you may choose any machine and use it interchangeably with the others. Your home directory will be the same on every QNX machine.

Once you have logged in, start a "Terminal" from the menu on the right. From there you will have access to the gcc and g++ compilers and the editor, vi.

You can launch the Momentics IDE by clicking Launch->Development-> "Integrated Development Environment".

You have two ways to develop and debug your programs. One way is to use the command line gcc/g++ compiler with or without using makefile to build your source file into QNX executable files, and then use GDB on target to debug them. The other way might be a better way. You can use the powerful Momentics IDE to edit, compile and debug your program. You can check the execution result on the target board in a console window; you can set break points to stop the program at any place in your source code; and you can see the multi-threads information.

Your QNX account

Your QNX account is almost same as your ENCS account except its /home directory, e.g. you can use your ENCS account to login QNX network, but pay attention to the different /home directories. On any QNX workstation, you also can access your ENCS /home by 'ssh login.encs.concordia.ca'. If you are opening your ENCS /home, you also can easily open your QNX /home. For example, if your user id is 'a_qnxuser', you can go to your QNX /home by **'cd *~/teaching/realtime/home/a/a_qnxuser*'.** In this case you also can copy files between these two home directories.

1.7 Introduction of QNX & Momentics IDE

It is a quick start to Qnx Momentics 6.2.1 Integrated Development Environment (IDE). The purpose is to introduce you to the QNX software environment of the real time system laboratory, and to help you start writing your first program for QNX real-time system in a short time.

How to get help from QNX?

In QNX Photon GUI, you can click "Help" on system shelf, which should be the second item of "Application" on the right side of your desktop by default.
You can also use the browser, by clicking Launch->Internet->Mozilla, to access the web site "www.qnx.com". You might need to set the proxy server after you first login. It can be done like that: launch the Mozilla, click:
Edit->Preferences...->Advanced->Proxies, select the "Manual proxy configuration", and key in "10.0.8.2" port "80" for all the services. Please note that it is the proxy settings for Concordia's QNX laboratory only.
And of course, if you want to study the QNX and Momentics on Windows, Linux or any other systems, you can use any Internet browser to visit QNX's web page to get more information.

Hello world program

You have two ways to build your program, by command line or IDE.
Using the command line method, you need to type your program by using vi or ped and save it as hello.cpp. Then use g++ to compile it.
g++ hello.cpp –o hello
You can run your program by keying in: **./hello**

Next, we start to build, "Hello world", in Momentics, first, Click Launch->Development-> "Integrated Development Environment".

Second, please make sure you are under the "C/C++ Development Perspective". You should keep in mind that at any time there are several perspectives (or kind of views) in the IDE. If you cannot find certain buttons you are supposed to have, maybe you are under the wrong perspective. Our purpose is to develop our first program, so the right perspective is "C/C++ Development Perspective". You can select this perspective, by clicking the button on your left side as the figure shown below.

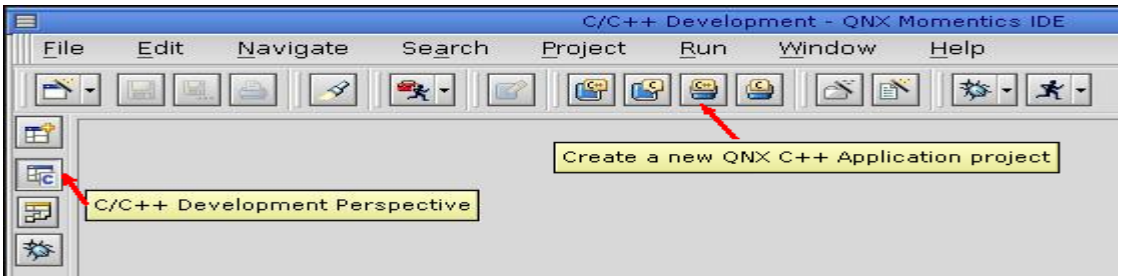


Figure 1: Different perspectives

Then, let us create our first C++ project. Why it is a project, not a program? According to QNX's manual, "In the IDE, a project is a collection of related resources (i.e. folders and files) for managing your programs. Although you can work on individual resources (e.g. edit a file), most of what you do in the IDE is project-oriented -- you build projects, manage versions of projects, share projects with other programmers, and so on. " And, do not to be afraid, it is so convenient to create a "Hello world" project, you do not even have to key in the keyword "printf" or "cout". All the things can be done automatically. You can find the details to create the project from the book QNX Momentics Professional Edition User's Guide, which can be found by clicking "Help" on system shelf.
Let us do it step by step.

Click the "QNX C++ Application button", if you cannot see it, you can select "File->New-> QNX C++ Application button", or "File->New-> Other…" and select what you want.

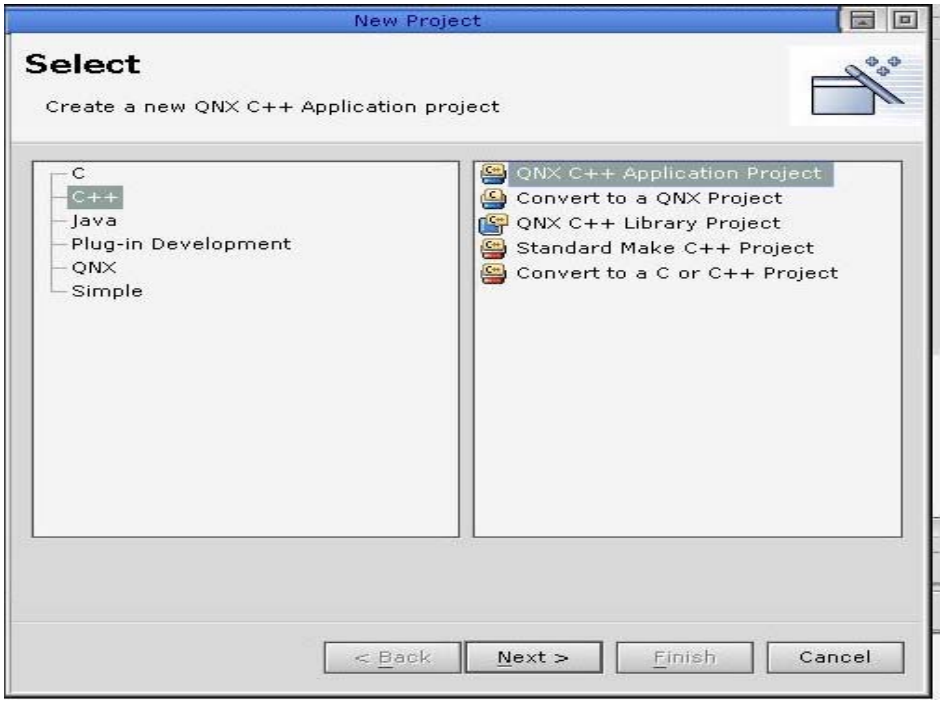


Figure 2: Create a new project

Key in the project name, let's say “helloworld”.
Click “next”, check the “x86 little endian” box, and click “finish”.

Figure 3: Application settings

You will then get the hello world program. It will print, “Welcome to the Momentics IDE” in the console window.

helloworld.cc

```
#include <cstdlib>
#include <iostream>

int main( int argc, char *argv[] ) {
    std::cout << "Welcome to the Momentics IDE" << std::endl;
    return EXIT_SUCCESS;
}
```

Figure 4: Hello world

Don't like it? You can key in your own, such as “Hello world”.

Build your binary codes

You need to convert your C or C++ source codes into machine codes in order to execute them on target board. It is called as build.

Switch the perspective to "QNX System Builder Perspective", right-click the "Helloworld" folder, click "Rebuild Project". After a few seconds, you will have your first binary codes in the real-time world.

Target or local host?

In most cases, the real-time codes are supposed to run on embedded target boards. You have many choices of the target CPUs and peripherals. But, the target board can also be the PC that you are using to develop the codes, if the PC has a QNX operating system. You need to set Momentics before you treat your PC (localhost) as a target. Fortunately, you need to do it only once, and then you can make use of it forever.

A debug server program, "qconn" has already been launched in the target board or your developing desktop PC. Momentics can communicate with it to download the codes, run the codes and debug them.

Because the "Helloworld" program can be run on you local machine, you can select it as your target. Switch the perspective to "Debug Perspective", click "Run->Run..."

Right-click the "C/C++ QNX Qconn(IP)", select "New". Then select your project, helloworld, and C/C++ applications, also "helloworld". You may find that there is another file named helloworld_g. That file is for debug; you may use it later.

In "Target Options", right click "Add Target". And choose target name as your wish, for example, "Localhost". In "QNX Connector Selection", uncheck the "Use local QNX Connector", in "Hostname or IP", key in "localhost" and let the port as default "8000". And of course, if you want to connect to a real embedded target, you can key in the name or IP address of that target. Click "Finish" to complete the target setting.

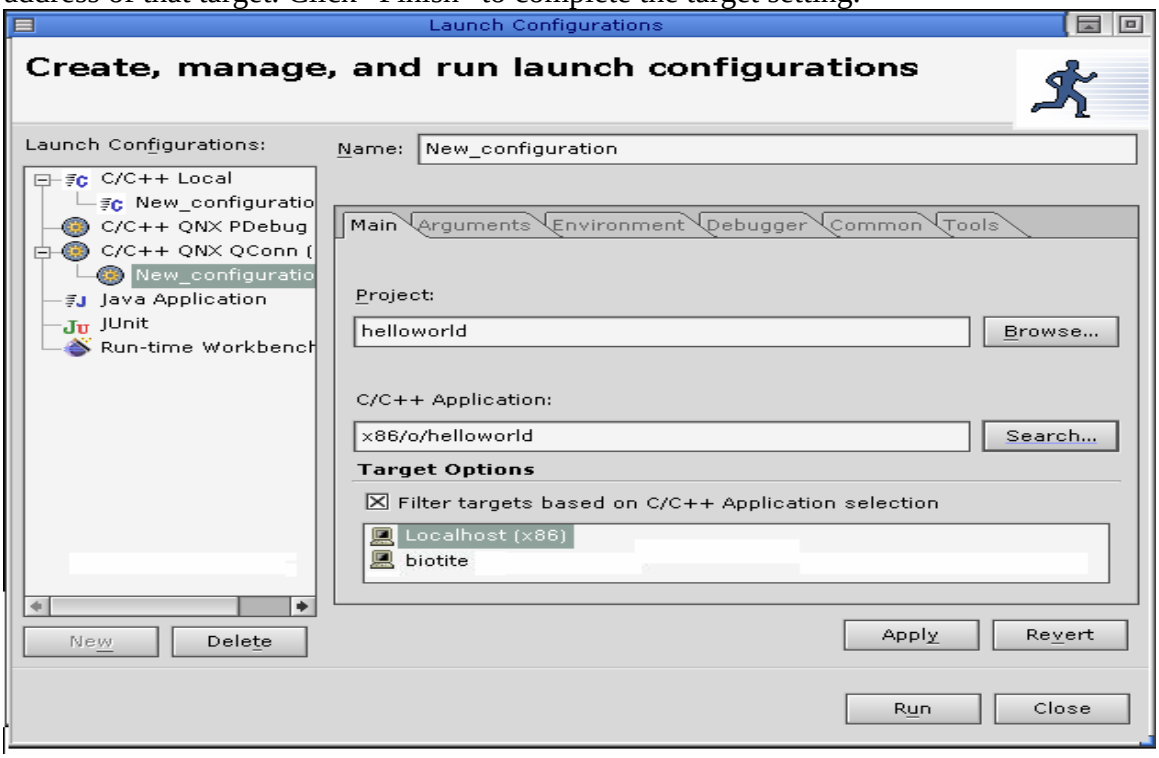


Figure 5: Target selection

Click the target name you just created, and then click “Run”, it will bring you “Hello world!” in console window.
If you want to test I/O ports, you need to use the real target board, which has a QNX OS image and has “qconn” running. If so, you can also choose its name or IP. Then you can use this debug GUI to debug your applications that are running on the real target board.

Debug “Hello world!”

Do you like to debug your “Hello world!”? There is of course no need for it. But if your codes are very complex, you may wish to debug them instead of running and seeing their result. You can set breakpoints, inspect the variables and CPU registers, to step over or step into your source codes,..etc.
So let us add some codes so that you can have a chance to inspect the variables, because there are no variables to be seen in “Hello world!”.
Add some lines as below under your “Hello world!” line.

```
int i, sum=0;
for (i = 1; i<=100;i++)
      sum+=i;
std::cout << “Sum of 1+2+3+..+100 is:” <<sum <<std::endl;
```

Then build it.
To set a breakpoint, you can simply double click or right click on the small gray area to the left of your source view. A small circle is placed there to indicate the breakpoint is set. Click “Run->Debug...”, select the target “locolhost”, select the file “helloworld_g”, and click debug. The “xxx_g” files contain debug information, so if you want to debug the program, you must select the “xxx_g” files, and if you just need to execute them and to see the result, please select the file without “_g” in their name.
The program will run and stop at the first line of your program; you can click “resume” to resume it. And it would stop at the breakpoint you set. Then you can see the variables, and to use the debug functions like “step over”, “step into”, read/write CPU registers ... etc.

2 FREQUENTLY ASKED QUESTIONS

How to get help & report problems?

Send e-mail to helpdesk@ece.concordia.ca or danli@encs.concordia.ca.

How to access QNX from home?

For Linux users:

1) ssh login.encs.concordia.ca
2) ssh mackay.realtime.private or ssh peel.realtime.private

   If you are working in ENCS network, you just run the second step only.

For Window users, you may have to install a SSH client, such as Secure SSH client (http://www.ssh.com/ or http://hp.vector.co.jp/authors/VA002416/teraterm.html.).
Once you have a SSH client, you can access the encs.login server:

login.encs.concordia.ca

then you can use ‘ssh mackay.realtime.private’ or ‘ssh peel.realtime.private’. Once you remotely log in QNX, you can’t run an application with GUI.

How to access Internet

Open Mozilla, then click on "Edit" and "Preferences". Click the "Advanced", and select "Proxies", select the "Manual proxy configuration" in the HTTP Proxy field put **'proxy-realtime.encs.concordia.ca'** and port **3128.**
You should be able to browse the web using the ENCS proxy web server.

How to print my program?

No printing directly yet. You can however access your QNX files from the rest of the ENCS network, and print them.
For example, your ENCS ID is 'a_qnxuser', you QNXhome directory can be accessed in your ENCS account at:
'*/teaching/realtime/home/a/a_qnxuser'*.

Can I install QNX at home?

QNX is free for personal use and downloadable from get.qnx.com.

You can install it either as a stand-alone operating system on its own partition (not extended though, you must have a free primary partition) or as a package under Windows.

The QNX web site provides information on how to install each version. A searchable "Knowledge Database" is also available at qdn.qnx.com, and human help can be found on various newsgroups available at the news server inn.qnx.com. (A good group to start with is qdn.public.qnxrtp.newuser).

How to access a floppy?

In order to use a floppy disk, you have to mount it first and then unmount once you are done with it. The command "dosflopmount" will mount a DOS-formatted floppy in the directory /floppy. You may then read and write file to the floppy. When you are done, type "dosflopunmount" to clean up and make sure all the data was written to the disk. (As opposed to being in the buffers.)

Be careful, other students can read/write your floppy while it's mounted.

3 HARDWARE COMPONENTS

3.1 Dummy I/O display

In the first set of experiments it is necessary to have some interface circuitry to visualize the data sent to the different interface attached to the single board computer.

The interfaces used are:

- Arcom IN16 -PC104 interface board
- Arcom OUT16 -PC104 interface board
- Serial input/output -COM 1 and/or COM2
- The goals are:
  - To provide data to be read by the IN16 board, we need a series of 16 switches.
  - To view data written to the OUT16 interface board, we need a series of 16 LEDs.
  - To view the data sent by the serial output and to send data to the serial input, we need a display and a data generator. The data has to be sent and received at a specified speed (baud rate) and within certain parameter.
- The dummy I/O display board has the following capabilities:
  - To view the data sent by the serial output, an LCD display with a serial input interface.
  - To send data to the serial input, a small ASCII data generator with a set of define strings or sentences.
- The board also includes 16 Switches and 16 LEDs.

Note: The serial interface experiments should be done using both COM1 and COM2. COM1 would be used with interface board and COM2 would be use to talk to the Easy DCC command station.

Interface board specifications

- Circuit generates data at 9600 baud, 8 bits, no parity, one stop bit.
- Generates 0 to 9, a to z, A to Z and some control character i.e. line feed, carriage return.
- Generates some small streams of command to talk to the Easy DCC command station.
- Pressing buttons do the choices of patterns and string.
- LCD display with serial input. The characteristic of the serial communication is the same as the one from the generator or with the same flexibility.
- Loop back connection between the data generator and the LCD input. It would have to be done so that it does not conflict with the data sent to the LCD by the SBC. The

header connector that would enable the loop back must disable the data coming in from SBC.

- Possibility of modification of DTE and DCE with jumpers.
- 16 switches and 16 LED to interface properly to the IN 16 and the OUT16 Arcom boards.

The board layout is based on the layout and size of the single board computer, and so, it fits perfectly on top of the SBC and the I/O board.

3.2 SBC/GX1 Single Board Computer

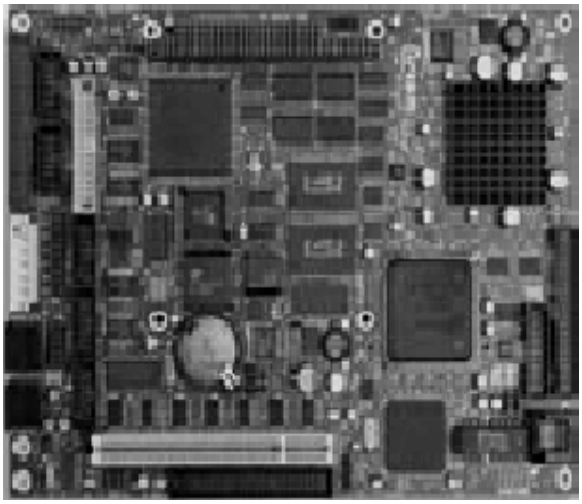


Figure 6: SBC/GX1 single board computer

The SBC-GX1 is a new low profile, high-performance, multimedia PC-compatible processor board.

The SBC-GX1 is an EBX form factor board, based on the 300MHz MMX-enhanced National Semiconductor Geode processor. It comes fully loaded with all standard PC interfaces as well as a full range of multi media features: Video (National Xpress Graphics), 10/100 base-TX Ethernet (PCI 2.1 compatible), on board flash drive (max. capacity 16Mb), dual USB, sound blaster compatible interface and 4 serial ports. The board is capable of driving high-resolution modes on CRT and flat panel displays simultaneously. A wide range of expansion options are provided via a compact flash socket, PC/104 bus connector and a standard PCI slot.

The board is available for all popular operating systems including, DOS (Data light's ROM-DOS and Flash Filing system pre-installed), Windows 95/98/NT, NT Embedded, CE, QNX, Embedded Linux and VxWorks.

3.3 PC/104-IN16 Board

Figure 7: PC104, IN16 board

Introduction:

The AIM104-IN16 is an 8-bit PC/104 module providing 16 opto-isolated inputs. The module provides up to 1500V electrical isolation between your PC/104 based control system and the electrical system you are monitoring. The isolation between adjacent channels is limited by the wiring and connectors to 100V. The module can be supplied in two configurations:

1. AIM104-IN16 Module inputs driven by external power source.

2. AIM104-IN16-DC/DC Module fitted with +24V isolated DC/DC converter to provide excitation supply.

The board includes jumper options for each channel to use the on board supply and also jumpers to select a 10ms input debounce filter.

Operation:

The status of each input is read from two I/O addresses, the base address and base address +1. The board decodes four READ only byte locations, however the address locations 2 and 3 are a repeat of locations 0 and 1. When an input channel is switched ON the value read by the host will be a logical “0”.

Each input is configured as follows:

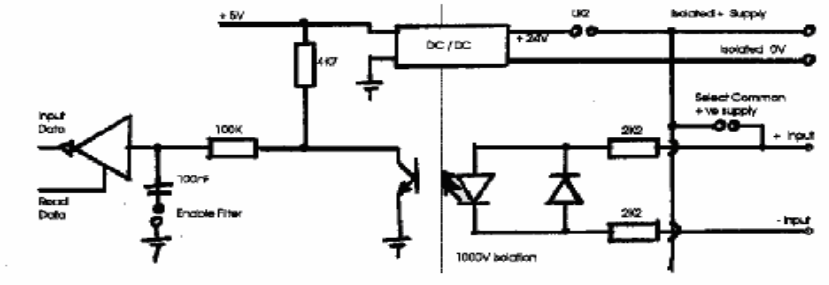


Figure 8: IN16 board schematic

3.4 PC/104-OUT16 Board

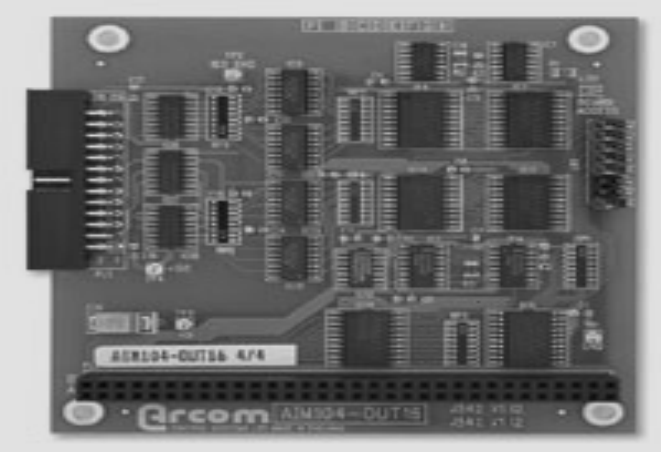


Figure **9**: PC104, OUT16 board

Introduction:

The AIM104 - OUT16 is an 8-bit PC/104 module providing 16 opto-isolated Darlington driver outputs. The module provides up to 1500V electrical isolation between your PC/104 based control system and the electrical system you are controlling. The isolation between adjacent channels is limited by the wiring and connectors to 100V. The outputs are configured as current sinking drivers with a drive capability of up to 600mA for channels 0 to 4 and up to 300mA for channels 5 to 15.

Operation:

Each output channel is switched ON by writing a “0” to the channel control register. The Darlington output will be sinking current when switched ON. On power-up or when reset is asserted, all the outputs are disabled and switched OFF. If the power on the PC/104 host system is switched off at any time all outputs will be switched OFF. The outputs are enabled by writing to, base +2 address.

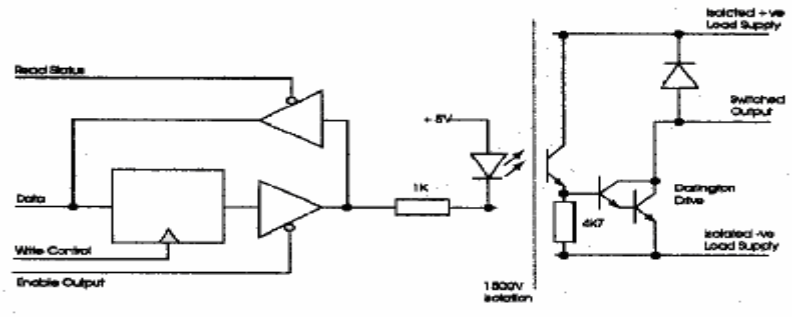


Figure **10**: OUT16 board schematic

3.5 Easy DCC command control system

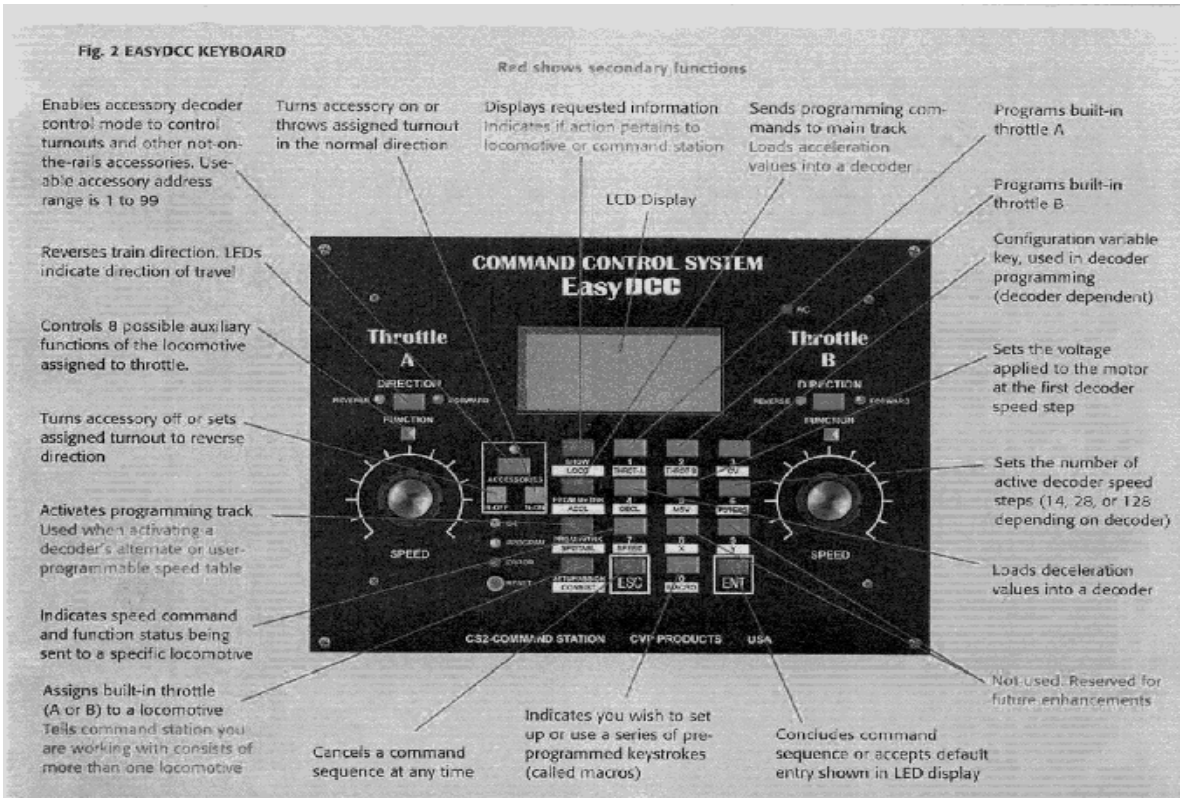


Figure 11: EasyDCC hardware control panel

Front panel controls and key definitions:

This page shows the key names, controls and functions of your Command Station. Refer to controller picture.
Familiarize yourself with the key and indicator locations.
The following information shows how to use your Command Station through the use of detailed keystroke sequences and options. In most cases, keys have dual functions.

Control or LED Primary Function Secondary Function(s):

AC LED---------- On when AC power is applied.
LCD---------- Display 16x2 character message display.
Direction A-------- Change direction of throttle A.
Direction B-------- Change direction of throttle B.
Function A--------- Show functions for Throttle A.
Function B--------- Show functions for Throttle B.
Speed knob-------- A Controls speed for throttle A.
Speed knob-------- B Controls speed for throttle B.
Accessories-------- Enables accessory mode.
R- OFF---------- Turn addressed accessory OFF or reverse Select reverse direction for consisted locomotive.
N- ON---------- Turn addressed accessory ON or normal Select normal direction for consisted locomotive.
OK LED---------- On for response from programmed locomotive.
PROGRAM LED-- On when program track is active.

ERROR LED------ On when there is an internal EPROM memory error.
RESET---------- Resets Command Station internal memory Clear consists and gangs (use with ENT and ESC).

Main Keyboard Definitions and Functions:

- SHOW---------- Displays requested information about locomotives, Command Station, throttles and accessories.
- LOCO---------- Usually means a specific locomotive address (CV#1) but may also be used for any decoder's address.
- PRGM MnTRK--- Access the main menu for programming decoders on the main track (ops mode programming).
- ACCL---------- Selects the Acceleration variable (CV#3).
- PRGM PrTRK---- Access the main menu for programming decoders on the programming track.
- SPDTABL--------- Speed table selection. Used to select the standard/default decoder speed table or a custom table.
- SETUP/ASSIGN-- Used to set-up consists, gangs, assign built-in throttles or program the command station.
- CONSIST---------- Used to build and change locomotive multi-unit consists or gangs of accessory decoders.
- THROAT-A------- Specifies Throttle A.
- DECL---------- Selects the Deceleration variable (CV#4).
- SPEED---------- Sets up speed and function monitoring of a locomotive. Also used for diagnostics.
- THROAT-B------- Specifies Throttle B.
- MSV---------- Selects Motor Starting Voltage (CV#2).
- X---------- When used with SHOW, shows the ID number and current address of a throttle.
- Y---------- Reserved for future enhancements.
- MACRO---------- Selects from a group of pre-programmed functions or allows custom DCC packets to be built.
- CV---------- Selects the main menus when working with Configuration Variables.
- #STEPS---------- Number of Speed Steps. Used to set-up decoder and Command Station to the desired number of steps.
- ESC ESCAPE----- Key that cancels a command sequence at any time.
- ENT ENTER------ Key is used to conclude a command sequence or accept default entries shown in the display.

RESET Key:

This key has several CLEAR functions that can cause loss of set-up data if not used properly. Please pay attention that it is not the same one to reset the target board. It only reset the EasyDCC panel.

The Command Station Display:

The 2 lines by 16-character display are used for all Command Station messages. In some examples, we may skip intermediate messages and show only those that are relevant to the

procedure. For example, this message is the “Normal Operation” message, which appears after the Command Station has initialized after power is turned on.

Severe Abbreviations:

Some words may be severely abbreviated to fit the display. For example this message shows three options from which you can select. The abbreviations are for specific keys. Without looking, see if you can guess which key goes with which word.
Answer: Spd=SPEED, #Steps=#STEPS, CONS=CONSIST.

Menu Options In Display:

When you see numbered entries with an equal sign, this is your clue as to which key is used to select that option. For this example, key 1 selects 14 speed steps; 2 selects 28 speed steps and 3 selects 128-speed steps.

Dual Function Keys (and sometimes multiple functions):

Many keys have a primary and a secondary function. The word in white lettering is the primary function. For this key, its primary function is the SHOW command. The word below it is in reversed lettering. For this key, the secondary function is to specify that the command be related to the locomotive. If in doubt, push the key you believe to be the correct one and verify by watching the subsequent message. With very few exceptions, you’ll find this to become second nature after a short while.

Key label LCD display abbreviation:

ACCL Accel Value
SPDTABL Speed Table
THROT-A A
DECL Decel Value
SPEED Spd
THROT-B B
MSV Start Value
MACRO Macro
CV CV#
#STEPS SpdSTEPS, #STEPS
R-OFF R
N-ON N

Command station initialization messages and their meanings:

Observe the initialization messages as the Command Station sequences through a self-test. If you want to see the messages again, push the RESET button.
To freeze a message, like the version number, push and hold the RESET key when the message appears. When the Normal Operation display appears your Command Station checks OK and it is ready.

Default Settings For Command Station:

Throttle A: pre-assigned to address 03
Throttle B: not assigned
Speed steps: 03 is set to 14 speed steps
Decoder Functions: all Functions off
Accessory decoder: all outputs off or Normal
Consists: None
Accessory Gangs: None
Memory OK
Memory BAD DCC Programmer
Command Station
Version xxx
Main Track Set-up Complete Normal Operation
Th-A=03 Th-B=--

Restore Factory Defaults:

To force a total system reset, or clear the entire memory, use the ENT+ Power Cycle command. Push and hold the ENT key. Push the RESET button, or turn the power off, then back on – "power cycle." The message "MEMORY BAD" will confirm the system memory has been cleared and factory defaults at left have been restored.

Changing the throttle address:

Changing the address to which throttle A or B is assigned is easy. The exact keystroke sequence is shown below:
This is a key point to remember: never have two throttles controlling the same address. If you discover the locomotive is running very poorly, immediately suspect that two throttles are attempting to control the same decoder.
For this example, throttle B will be assigned to locomotive having a decoder address of 21
Setup/Assign
ThrotB
21
ENT
ESC

De-assigning command station throttle:

To de-assign the Command Station throttle, simply leave the number blank when prompted for the loco address after pushing SETUP THROT-A or B. This function applies ONLY to the Command Station's built in throttles.
Setup/Assign
ThrotB
ENT
ESC

Locomotive doesn't run but was working fine:

Check: Booster red short circuit LED is "OFF".
If ON, there is a short on the layout.
Locomotive is on a powered part of the track.

Command Station throttle assigned to proper address.
Be sure only ONE throttle assigned to the address.
Command station set to 128-speed step on a decoder that has only 14/28-step setting.
Broken decoder wire near motor or track pickups.
Broken motor brush wire.
Locomotive isn't on the address you think it is.

Locomotive with decoder never did run:

Check: All the comments above, and, address may not be on factory default. Use programming track to program desired address.
Decoder wiring may be incorrect or a wire may be broken.
Motor brush reinstalled wrong or damaged during installation (check, replace).

Diagnostic Commands:

Monitor Speed Information From Throttle: SHOW, LOCO nn, ENT, SPEED

Check Command Station's transmitted speed steps: SHOW, LOCO nn, ENT, #STEPS

Check current function status: SHOW, LOCO nn, ENT, #STEPS

3.6 – Serial Communication

Communications:

Uses a baud rate of 9600, 8-bits byte, no parity, and one stop bit. Only ASCII characters are sent.

Commands:

All received commands must use the ASCII character set (upper case) and be terminated by a Carriage Return (Enter)(0d hex.). The command sequences must use the exact number of characters shown, for the command to be executed.

The format below shows the data and the number of characters required. Spaces are shown as delimiters in the command sequences below. The spaces are ignored by the command station and may be used or not used by the user, as he desires.

If the command executes, properly or otherwise, various responses will be transmitted and are described at each command description. If the command received was invalid or, for whatever reason, could not be decoded, a response of ?<CR> will be transmitted. This response also indicates that the system is now ready for another command.

Command Modes:

At power-up/reset, the command station will default to the operation mode. (There are 2 modes: Operation and Programming). In the Operation mode the command station will respond only to Operation Commands; in the Programming mode it will respond only to Programming Commands.

Display Memory:

Format:
`F y xxxx` where y=hex number of 16-byte lines (0 implies 16), and xxxx=hex RAM memory address.

Example:
F 2 0C00 Display 2 16-byte lines of memory starting at location 0C00h (3072 decimal).

The information transmitted is 32 ASCII bytes per line requested.
(e.g. '01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F'), followed by a terminating <CR>,
followed by O<CR>...to indicate system is ready for next command.
<"O" indicating ready for next Operation Command>

Kill Main Track:

Format:
K No data required. Causes all clocks to Main Track to stop. The Program Track clocks will continue but the Program Track will not be enabled unless Program Mode is selected.

Successful completion of this command will be indicated by transmitting O<CR> to indicate the system is ready for the next command.

Enable Main Track:

Format:
E No data required. Causes the Main Track clocks to be resumed.

Successful completion of this command will be indicated by transmitting O<CR> to indicate the system is ready for the next command.

Display Version Number:

Format:
V No data required. Causes the version number of the Command Station software and the release date to be sent to the PC.

The information transmitted is 12 ASCII bytes, beginning with a V, containing the information requested.
(e.g. 'V006 06 02 1997')...for version number 6
followed by a terminating <CR>,
followed by O<CR>...to indicate system is ready for next command.

Queue Packet:

Format:
Q xx xx ... xx Where xx is the hex value of each byte of the packet to be placed in the main command queue (up to 6 bytes may be specified, and must include the checksum byte). Once in the main command queue, the packet will be sent each cycle until it is replaced by another packet to the same decoder number.
When a decoder number is the same in a packet being received and in a packet already in the queue, the newest packet will replace the existing one. Two packets addressing the same

decoder cannot be in the main command queue. Packets may contain decoder numbers up to 10239 (hex 27FF). The maximum number of packets that may be placed in either queue is 32. (Operational note: If 32 3-byte packets are placed in a queue, that represents a time delay of approximately 1/4 second that is added between packets to any one decoder on the main track.)

The most useful Q command packet is setting the speed, direction and light of a train. You can use:

```
Q <TrainID> <Command> <CheckSum>
```

The one-byte "Command" has the bits structure that is listed below:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	Dir	Light	Speed			

Dir: 1 - Forward, 0 - Backward
Light: 1- Lights on, 0 - Lights off
Speed: speed coded in binary, 1111 = -1 = stop, 1110 = 14= top speed

CheckSum= TrainID^ Command

Remember, you need the Q and spaces between each packet!

Example:
Q 17 67 70 Put a command for loco (decoder) #23 for speed 6, forward direction, in the main queue with a checksum of 70.

Successful completion of placing the packet in the queue will be indicated by transmitting O<CR> to indicate the system is ready for the next command. If the packet will cause the queue limit to be exceeded the packet will not be placed in the queue. This will be indicated by transmitting R<CR>. <"R" indicating the command was decoded but refused>. The system will then be ready for another command.

By default, the EasyDCC does not enqueue packets it receives. If Q packets are not used, the packets would be received by the EasyDCC, processed, sent to the train, the train would respond to the command it received but would stop after its reception. Hence, by using Q packets it is assured that once a packet is received, the command enclosed in the packet will be repeated until a new packet is received by the EasyDCC.

3.7 Booster

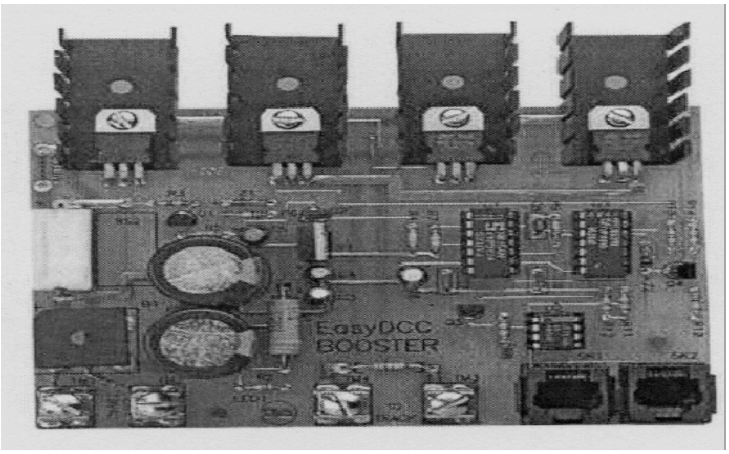


Figure 12: Booster

A booster, which acts as an amplifier, is used to strengthen the electrical signal of the digital packet commands received from the command station.

In DCC, a small electrical signal is transmitted through wires from the command station to the booster. The signal is received, amplified, and then sent to the track.

3.8 Decoder

Each locomotive is equipped with a decoder. The decoder has an ability to interpret the data sent by the command station, and to act accordingly.

While all decoders receive digital packet commands transmitted from the power station via tracks, only the decoder to which the command is addressed will respond.

The decoder also has an ability to vary voltage inputs to motors to control speed, direction of locomotive(s) and light on/off.

3.9 Occupancy Detector

The following circuit detects the presence of a locomotive on a track segment. Every layout is divided into 8 or 16 segments. Once a locomotive enters a segment and draws current, the circuit then detects its presence. It then puts a logic “1” to the corresponding output. AIM104-IN16 board processes this particular logic value.

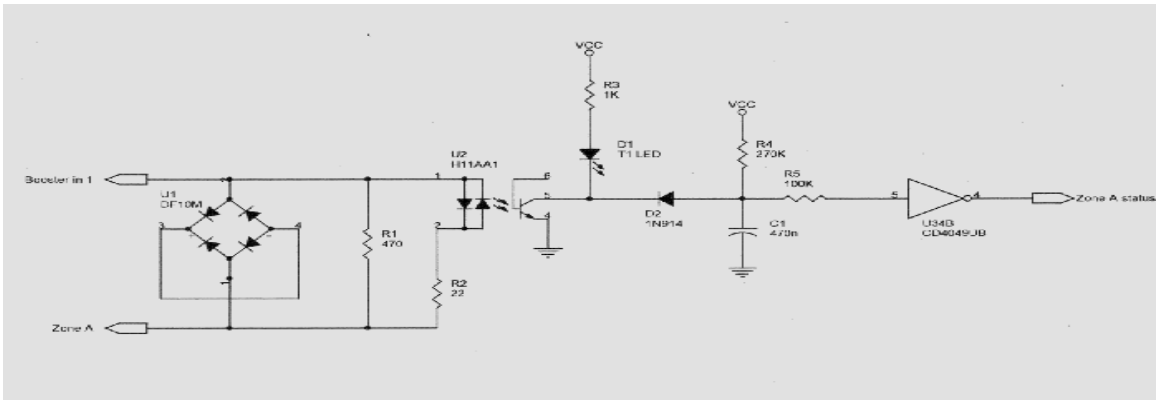


Figure 13: Occupancy detector

Current detection circuit functional description:

The jumper diode bridge is a way to get the required two-diode voltage drop. It is required that the bridge be able to switch at the DCC frequency, around 10 KHz. Parallel across the diode bridge is a 150 ohm resistor. This resistor supplies the current limiting for the internal led of the opto-isolator. There are two LED's because it is an AC input, so as the DCC signal alternates so do the led. This is how part of the sensitivity is gained back, one or the other LED is always lit. Once a locomotive draws current the transistor output sinks the capacitor voltage near zero then the output of the inverter goes to one. When the locomotive goes out of the zone, capacitor gets charged up to the threshold voltage of the inverter then it shows a logic zero.

Notes:

A major benefit of current detection systems is that they can easily detect locomotive current, regardless of how long a zone may be, or how the track is routed. And, as long as some current is present, they will respond properly to trains that stop, or enter in and leave out of a zone. But they will generally introduce some drop in the voltage, which actually reaches the motor.

3.10 –Turnout Driver

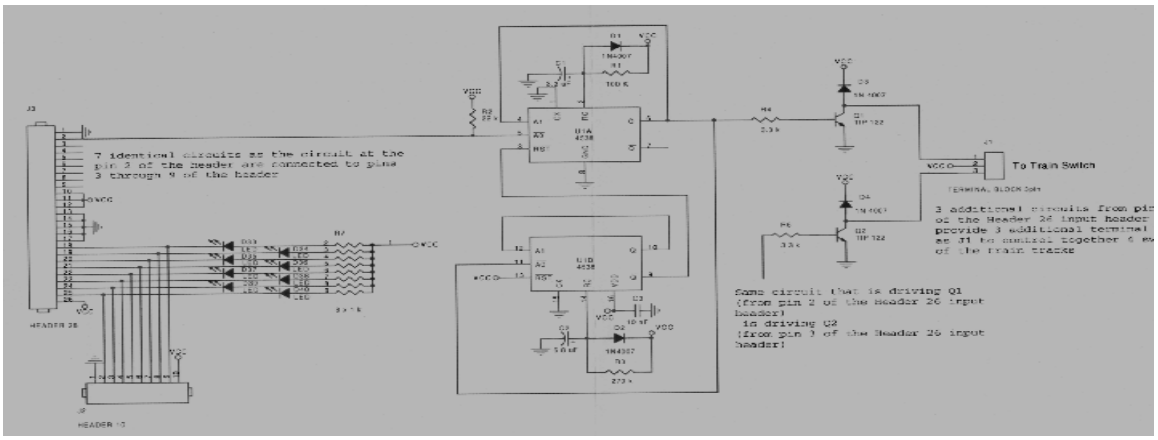


Figure 14: Turnout driver schematic

The switches installed on the experiment tracks are labelled A, B, C, D, E and F.

The switches A and B are installed on the straight track for the experiment#3.

Each switch has two positions: straight and curved.

Each switch is controlled by two output channels of the OUT16 board.

For experiment#3, the channels 0, 1, 2 and 3 of the OUT16 board control the switches A and B as follows:

Switch	Channel	Action
A	0	straight
A	1	curved
B	2	straight
B	3	curved

Each switch contains a solenoid with 2 windings that control the switch position: straight or curved.

When you send a short negative pulse on a control channel of a switch (a transition from '1' logic to '0'), a monostable is triggered, the corresponding solenoid winding is supplied with +12V for 0.2 second and the track is switched accordingly: straight or curved.

To prevent the burn out of a switch winding (when the +12V supply is kept on it for a few seconds or more) another monostable circuit disables the control negative pulses from the corresponding control channel of the switch for 4 second after the first monostable circuit is triggered (i.e. after receiving a valid control negative pulse). That is, any negative control pulse sent after a valid one on a control channel in a time frame of 4 sec. is ignored. After that 4-second time interval the control negative pulses on that channel are again enabled. However you can turn the switch to the other direction any time after it was switched to the opposite direction.

Also, any negative pulse with a pulse width of a few microseconds to a few seconds on a control channel of OUT16 board will trigger the corresponding supply circuit of a switch solenoid winding and the tracks will be switched accordingly (straight or curve). However the hardware protection circuit of the switch will disable any other control pulse for a time frame of 4 second on that winding to prevent overheating and burn out of that winding.

3.11 Traffic Lights

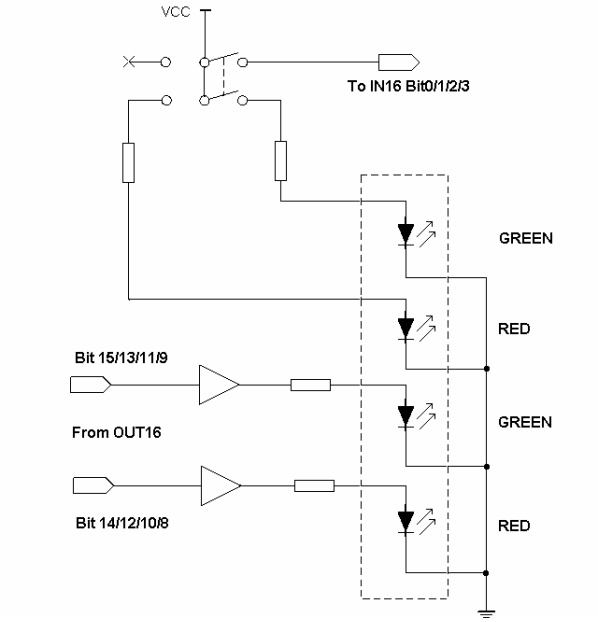


Figure 15: Traffic lights

Four traffic light units are installed on the platform. From the simplified schematic, we can see that in each unit, there are four lights, 2 red ones and 2 green ones. The lights of each unit have two groups. One group of lights (a green one and a red one) are controlled by a DPDT, which is used to turn on the red and green lights manually, and at the same time to send the same signal to a IN16 board so that the SBC board can detect the changes of the lights. The other group of lights are connected directly with bit 8~15 to the OUT16 board. So, the SBC board could control them.

Those four units are installed in the train station area, namely track segments 11, 12, 13 and 14.

The trains should respect the lights facing to them. For detailed information, please look into experiment 4.

4 SOFTWARE COMPONENTS

Following are the main header files containing different class interfaces, which could be used in coding for the various lab experiments. These interfaces, in fact, provide a clear design strategy for the various tasks to be accomplished in each of the experiments.

4.1 Pthread

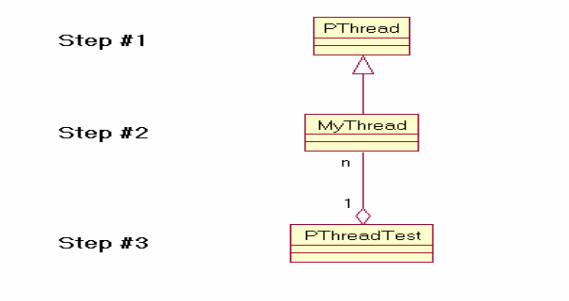


Figure 16: Pthread class

Description:

This class allows the creation of generic thread, which takes care of the internals of the POSIX Thread features. In order for a class to be a thread, that class only needs to inherit from this PThread class to obtain all the Pthread functionalities and to implement the run function, which represents the code of the thread. The thread code can be run simply by calling the start() function. The calling thread can wait for that process to finish by calling the join() function.

List 1: Pthread.hpp

```
#ifndef  PThread_HPP
#define  PThread HPP

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>    //includes all the global error variables.
#include <pthread.h>  //includes all the Posix Thread functions.
#include <unistd.h>   //includes Standard Symbolic Constants & Types.

#if defined(   sun ) || defined(   SUN   )
#include <thread.h>
#endif

#include <sched.h>
#include <time.h>
#include <signal.h>   //includes the Software signals.
#include <sys/types.h>
#define  pid t          pthread t
#define  INVALID_PID   ( -1 )

 //  Attribute                Default Value
 //  =========                =============
 //   detachstate             PTHREAD CREATE JOINABLE
 //   schedpolicy             PTHREAD INHERIT SCHED
 //   schedparam              Inherited from parent thread
 //   contentionscope         PTHREAD SCOPE SYSTEM
 //   stacksize               4096
 //   stackaddr               NULL
```

```
class PThread {
public:

   // Default Constructor
   inline PThread( bool autostart = false, bool autojoin = false );

   // Destructor
   inline virtual ~PThread();

private:
   // Not implemented.
   inline PThread( const PThread& src );
   inline const PThread& operator=( const PThread& src );

protected:
   // Implement this, when deriving the class.
   // It should be overriden by it's children...
   // This is a pure virtual function, like it is for the Java API Thread
interface.
   virtual void run() = 0;

   //  Callback can only call STATIC functions !
   //  ============================
   //  This routine is a stub which will call the run() function
   // given the class instance as the void pointer argument.
   //
   // We need this static function because the following won't work:
   // pthread create( &pid, &attr, run, null );
   //
   // The reason is that you do not know in advance the address of
   // a pure virtual function since the address of a pure virtual
   // function is undetermined until it is executed. The only way
   // to execute this function via pthread create is to refer
   // to some other procedure, the stub, which we know the address
   // for sure in advance, since it is a " static function", to call
   // the pure virtual function for us.
   // The only way for that stub to know who to call for us,
   // is to give a pointer to ourself, so that it can resolve
   // at run-time which is the last implemented run
   // in the PThread inheritance hierarchy.

   static void* PThread::stub( void* ptrThis )
   {
     PThread* me = (PThread*)ptrThis;

     // Call the derived classes version, since the function
     // run is implemented by it's children
     me->run();

     // run() should normally contain an Infinite Loop,
     // i.e. for(;;) { ... }
     // Terminate normally, just in case there is no infinite loop.

     me->pid = INVALID PID;
     pthread exit( NULL );

     return NULL;
   }

public:
   // Get calling thread's ID
   inline pid t getPID();

   void start( int start prio = 10 );

   // Cancel a PThread and make it invalid.
```

```
   inline void stop();

   // Let other threads run, via sched yield()
   inline void yield();

   // Exit a PThread
   inline void end();

   // Is this PThread started ?
   inline bool isStarted();

   // Join PThread
   inline void join();

   // Detach a Join PThread
   inline void detach();

protected:
  // So, children can access easily the PID
  pid t                        pid;
  bool                         isDetached;
  pthread attr t               attr;
  sched param                  param;
  int                          priority;
};

// Inlined functions
#include "PThread.inl"
#endif
```

Example:

For instance, using such design, you could easily create a thread as follows:

List 2: Sample program to create a thread

```
#include "PThread.hpp"

class myThread : public Pthread

{
public:

   myThread( int id0 = 0 ) : id( id0), PThread() { }

   virtual void run()
   {
      for(;;) { printf( "Thread %d running \n", id ); }
   }
};

int main()
{
myThread t1( 1 );   // Display  "Thread 1 running"
myThread t2( 2 );   // Display  "Thread 2 running"
t1.start();
t2.start();
t1.join();
t2.join();
return 0;
}
```

4.2 Mutex

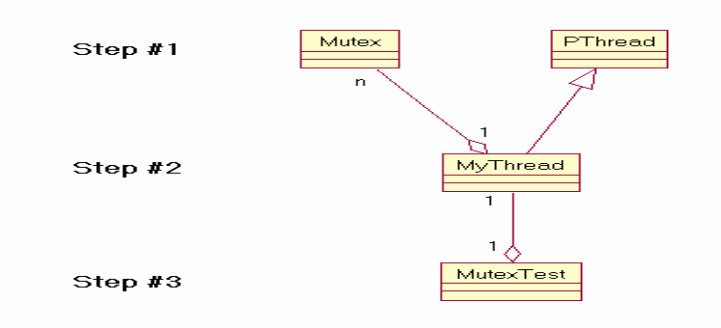


Figure 17: Mutex class

Description:

To protect a critical section, you can use this Mutex class, which will ensure that only one thread at a time can lock it.

List 3: Mutex.hpp

```
#ifndef  Mutex HPP
#define  Mutex HPP

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

// Uses:
// pthread mutex t
// pthread mutex attr t

class Mutex {
public:

  // Default Constructor
  Mutex( const char* mutex name = " " );

  // Destructor
  virtual ~Mutex();

  // Lock the mutex
  void lock();

  // Unlock the mutex
  void unlock();

  // Try to lock a mutex
  inline void tryLock();

  // Get the maximum priority (ceiling value) for the mutex
  inline int getPriorityCeiling();

  // Set the maximum priority (ceiling value) for the mutex
  inline int setPriorityCeiling( int new ceiling );

  // Get the mutex attribute type
  inline int getType();

  // Get the mutex attribute protocol
  inline int getProtocol();
```

```
protected:
  int          mutex type;
  int          protocol;
  int          priority ceiling;
  int          state;
  pid t        owner;
  char*        name;
  pthread mutexattr t     mutex attr;
  pthread mutex t         mutex;
  pthread condaddr t      cond attr;
  pthread cond t          cond;

// Not implemented or needed.
private:

  // Copy constructor
  inline Mutex( const Mutex& src );

  // Assign operator
  inline const Mutex& operator=( const Mutex& src );
};

// Inlined Functions
#include  "Mutex.inl"

#endif
```

4.3 Semaphore

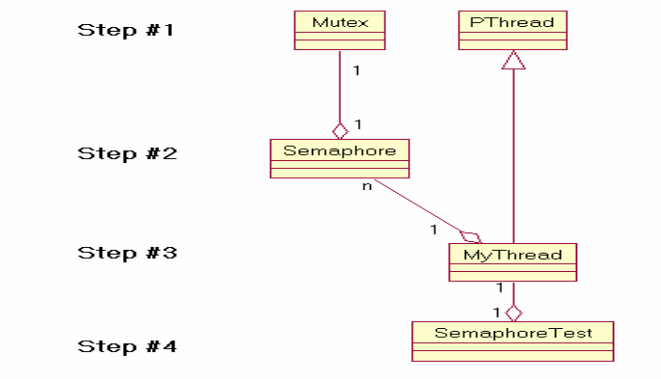


Figure 18: Semaphore class

Description:

A semaphore holds a counter that can take values from zero to a predefined maximum. Two atomic operations are allowed:
- wait, which basically waits if the counter is zero and then decreases the counter
- signal, which simply increases the counter

A counting semaphore can be implemented using the previous Mutex class and a condition variable. It can also be implemented using the set of sem_wait, sem_post and similar functions, which is less portable to other Unix platforms.

List 4: Semaphore.hpp

```
#ifndef  Semaphore HPP
#define  Semaphore HPP

#include "Mutex.hpp"
class Semaphore : public Mutex
{
private:
  size t    counter;

public:
  Semaphore( size t initial value = 0, const char* sem name = " " );
  inline virtual ~Semaphore() { }

    void P();
    void V();

  size t get();
  void  set( size t value = 1);

// Not implemented.
private:

  // Copy constructor
  inline Semaphore( const Semaphore& src );

  // Assign operator
  inline const Semaphore& operator=( const Semaphore& src );
};

#endif
```

4.4 Timer

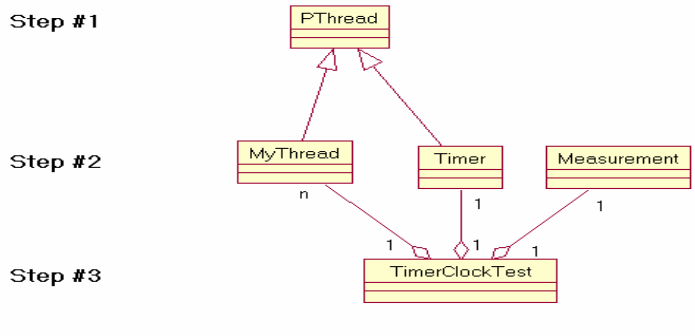


Figure 19: Timer class

Description:

The Timer class can be used to notify when a given amount of time has elapsed. The Timer could be either periodic or non-periodic, depending on the constructor value.

List 5: Timer.hpp

```
// Required library:  -lrt

#ifndef Timer HPP
#define Timer_HPP

#include "PThread.hpp"
#include <unistd.h>
#include <errno.h>
```

```
#include <pthread.h>
#include <signal.h>
#include <sys/time.h>

#if !defined( linux )

// Signal information
#include <sys/siginfo.h>

#endif

#if defined(   QNXNTO   )

#include <sys/neutrino.h>
#include <sys/netmgr.h>
#endif
// 16-17
#define  TimerSignal  SIGUSR1

class Timer;

// Only one timer can exist at a given time
// Timer*  Timer::theTimer = NULL;

class Timer : public PThread {

public:

    const unsigned long  Period 1 ms    =    1000000L;
  const unsigned long  Period 10 ms   =   10000000L;
  const unsigned long  Period 20 ms   =   20000000L;
  const unsigned long  Period 50 ms   =   50000000L;
  const unsigned long  Period 100 ms  =  100000000L;
  const unsigned long  Period 250 ms  =  250000000L;
  const unsigned long  Period 500 ms  =  500000000L;
  const unsigned long  Period 750 ms  =  750000000L;

  Timer( unsigned long nsec = 0, unsigned long sec = 0,
   bool periodic timer = true );

protected:

  static Timer* theTimer;
  virtual void run();

  // Should be implement by children class
  virtual void tick( int signo ) = 0;

  // Use the global variable for to call the real handler
  static void wrapper( int signo )
  {
     assert( signo == TimerSignal );
     assert( theTimer != NULL );
     theTimer->tick( signo );
  }

protected:
  clockid t             timerid;
  struct sigevent       event;
  struct itimerspec     t;
  bool                  periodic;
#if defined(   QNXNTO   )
  int                     pulse id;
  int                     timer pid;
  int                     chid;
  timer t                  timer id;
  struct  pulse     pulse;
#endif
};

#endif
```

4.5 Measurement

Description:

The Measurement class can be used to calculate precisely in terms of clock cycles an elapsed amount of time.

List 6: Measurement class

```
#ifndef Measurement HPP
#define Measurement_HPP
#include <stdio.h>
#include <stdlib.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>

class Measurement {
public:
  Measurement();
  inline  uint64         getCycles();
  inline double          getCPUfreq();
  inline unsigned long   getClockPeriod();
  inline double          start();
  inline double          stop();
  inline double          getElapsed();
  inline void            print();

  // Implemented!
  Measurement( const Measurement& src );
  const Measurement& operator= ( const Measurement& src );

protected:
  static double          cpu freq;
  static unsigned long   clock_period;

private:
  double         elapsed;
  _uint64        start_time;
   uint64        stop time;
   clockperiod   clkper;
};

#include "Measurement.inl"
#endif
```

4.6 Rwlock

Description:

The Read/Write Lock class can be used to have multiple readers accessing a given resource or a single writer writing into that resource. It is more common to find the implementation of pthread_mutex than pthread_rwlock on most UNIX platforms, so it is more portable to base our RWLock implementation from our Mutex class instead of using those pthread_rwlock functions. It is also simpler to code.

List 7: Rwlock.hpp

```
#ifndef  RWLock HPP
#define RWLock HPP

#include "Mutex.hpp"
```

```
class RWLock : public Mutex  {
public:

 inline RWLock();
 inline virtual ~RWLock();

 void read();
 void write();

 void releaseRead();
 void releaseWrite();

private:
 // -1  = writing
 //  0  = NotUsed (Available)
 //  1+ = reading

 long         readers;
 int          readfail;
 int          writefail;
 int          loop;

// Not implemented or needed.
private:

  // Copy constructor
  inline RWLock( const RWLock& src );

  // Assign operator
  inline const RWLock& operator=( const RWLock& src );
};

// Inlined functions
#include "RWLock.inl"

#endif
```

4.7 Example of a Buffer Producer / Consumer Problem

A bounded buffer is a concrete representation of the abstract idea of a sequence of portions. The sequence is accessible to two programs running in parallel: the first of these (the producer) updates the sequence by appending a new portion at the end; and the second (the consumer) updates it by removing the first portion.

The following is the overall class hierarchy diagram for a producer/consumer problem using a bounded buffer and the class interface discussed earlier.

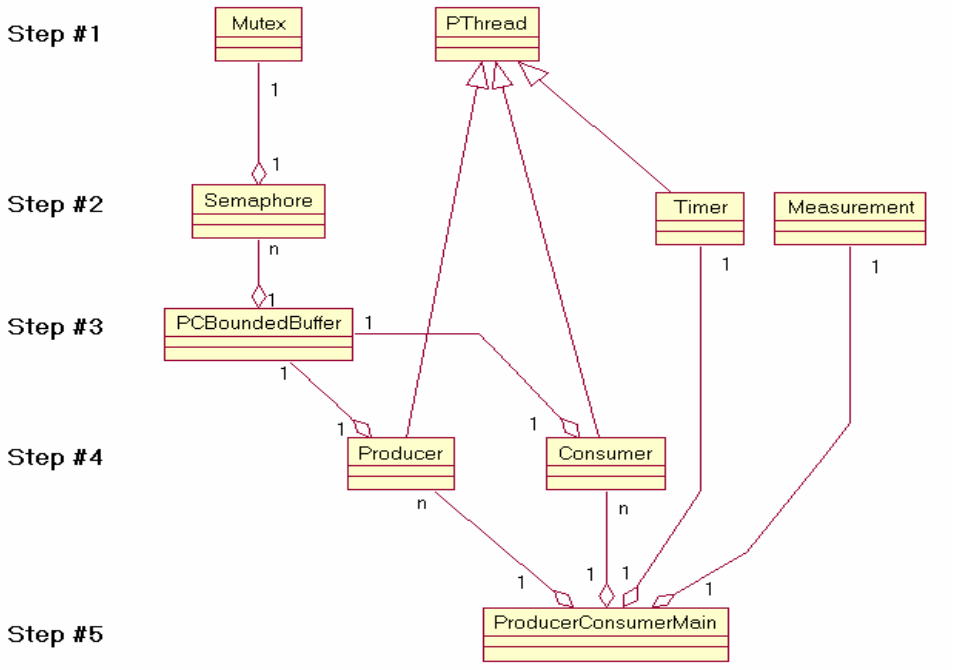


Figure 20: Class hierarchy diagram for a producer/consumer

5 USEFUL LINKS FOR ONLINE REFERENCE

1. To view the pseudo-codes for the classes apart from the ones shown in this manual, use the following path:

QNX labs: /public/j2k/nto/*

2. To view the man pages for the various POSIX functions, use the following link:

http://qdn.qnx.com/support/docs/neutrino_qrp/lib_ref/about.html

3. To learn more about QNX Real Time Programming and for help, go to the following sites:

http://www.qnx.com/
It is the home page of QNX.

http://www.qnx.com/developer/download/free/
Free download of QNX Non-Commercial Edition, training documents and video clips. It is a very good point to start. You can download the free version, see the slides, and listen to the instructions that teach you step by step about the QNX products.

http://www.qnx.com/developer/articles/index.html?article=dec1200b
It is a technical article, “What is Real Time and Why Do I Need It?”

QNX Documentation Roadmap
http://www.qnx.com/developer/docs/momentics621_docs/momentics/index.html
Do not know which book you should start first? Please try this link.

System Administration Guide
http://www.qnx.com/developer/docs/qnx_6.1_docs/sysadmin/docs/wip.html
This *System Administration Guide* shows you how to carry out the day-to-day tasks you need to do to manage your QNX real-time platform (RTP).

QNX Momentics Professional Edition User's Guide
http://www.qnx.com/developer/docs/momentics621_docs/ide_en/user_guide
This *User's Guide* for the Integrated Development Environment (IDE), which is part of the QNX Momentics development suite, accompanies the software and is intended to introduce you to the tools environment and to help you learn about using it effectively to build your QNX Neutrino-based systems.

QNX Momentics Professional Edition Programmer's Guide
http://www.qnx.com/developer/docs/momentics621_docs/neutrino/prog/about.html
The *Programmer's Guide* is intended for developers who are building applications that will run under the QNX Neutrino Real-time Operating System.

Photon Programmer's Guide

http://www.qnx.com/developer/docs/momentics621_docs/photon/prog_guide
The Photon *Programmer's Guide* is intended for developers of Photon applications. It describes how to create applications and the widgets that make up their user interfaces, with and without using the Photon Application Builder (PhAB).

http://www.parse.com/
It is a place that you can get free sample programs on QNX.

www.openqnx.com
It is the QNX Community Portal and BBS system. Have problems? Come here and talk about them with other QNX users.

QNX news groups
news://inn.qnx.com:
QNX Momentics Discussion Groups:
qdn.public.newuser
qdn.public.installation
qdn.public.neutrino
qdn.public.photon
qdn.public.devtools
qdn.public.ddk
qdn.public.bsp
General Discussion Groups:
comp.os.qnx
qdn.public.articles
qdn.public.advocacy
qdn.public.news
qdn.public.porting
qdn.public.qnxjobs
qdn.public.test

6 EXPERIMENT# 1: Hardware Interface

Objectives:

To understand system timer, input/output (I/O) boards and serial port communication using the target Single Board Computer (SBC-GX1), also to develop one small program on the QNX Real Time Programming development system and to run it on SBC-GX1.

Diagram:

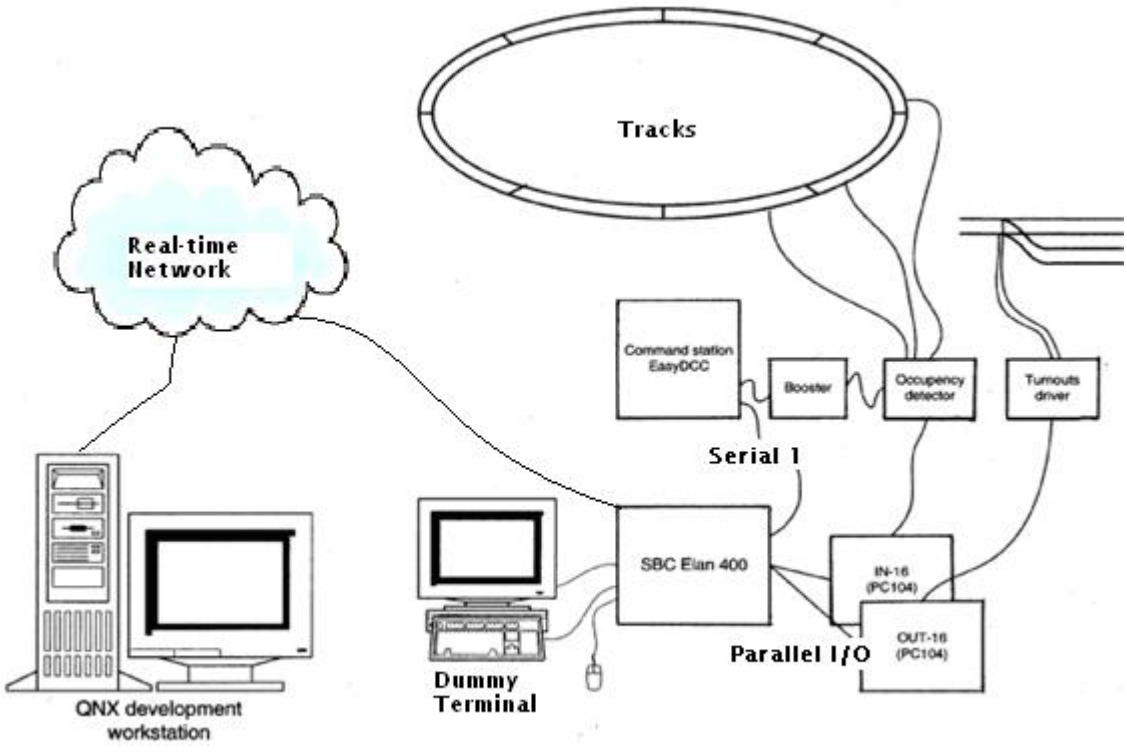


Figure 21: System hardware

Overall Operation of the Train Setup:

In lab #1, you are required to build up one small application that contains two parts:

1) The first part, it provides the basic function of bi-directional serial communication between SBC and Dummy terminal. The SBC is connected to the Dummy with its serial port 2 (COM2). You application shall open, read and write COM2 to communicate with the dummy terminal. Essentially, through one of the two concurrent processes, it reads continuously from COM2, and once a message is received, it will be displayed on the SBC's console. At the same time, the second concurrent process continuously reads from the console for some keystroke. Once it receives a keystroke, it sends the character immediately to COM2.
2) The part two, it can read from the first IN16 board whenever there is a timer event. The IN-16 board is connected to the occupancy detector circuit (refer to 3.9) on the

board; thereby it helps in processing the location of the train on the track. The trains can be controlled by the EasyDCC through its command control station. Please refer to 3.5 EasyDCC Command Control System about how to control a train.

Suggested Class Hierarchy Diagram:

The overall class hierarchy diagram for this experiment would look like as follows:

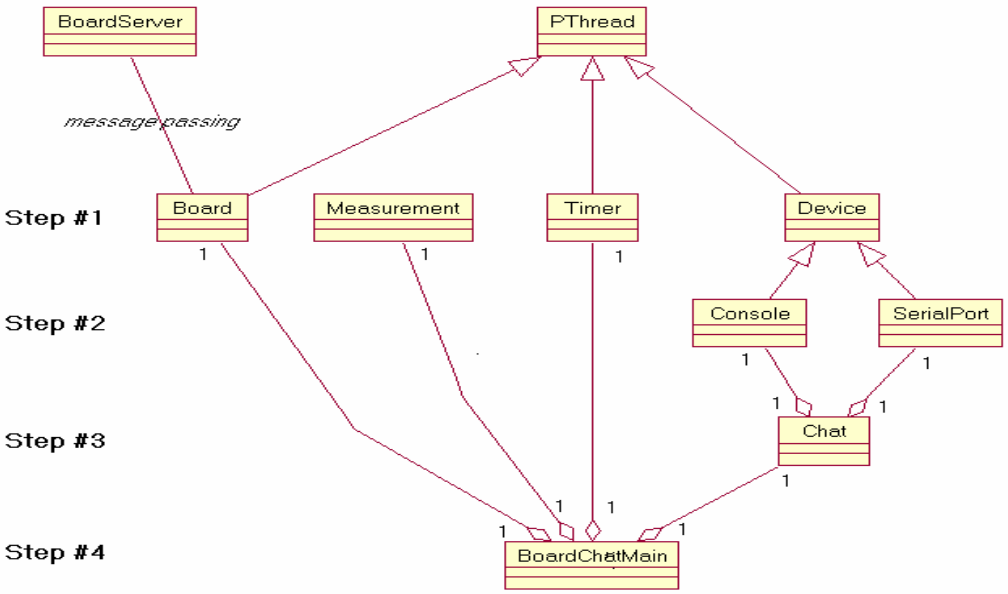


Figure 22: Overall class hierarchy diagram for experiment #1

- Pthread: this class wraps the POSIX thread functions.
- Timer: this class is used to notify when a given amount of time has elapsed. The timer could be either periodic or non-periodic depending on a specific requirement.
- Board: this class provides an interface to read from/write to the Arcom IN16/OUT16 ports.
- Device: this class simulates the operations performed by the serial and console devices.
- Console: this class is used to receive acknowledgements from the EasyDCC, which confirms that the message that was sent was received correctly.
- Serial Port: this class is used to write to the serial port, which is connected to the EasyDCC.
- Measurement: this class is used to precisely calculate, in terms of clock cycles, an elapsed amount of time to cover a track segment(s).

Suggested Design:

- One of the serial port **'/dev/ser2'** is plugged in to a dumb terminal which is being used for observing communication activities taking place via the serial port of SBC-GX1. It will basically be used to display any data (ASCII characters) being sent to the

serial port2; as a two-way communication, you can also send data to the serial port by simply typing on the dumb terminal keyboard.

- In order to work with I/O and serial ports, in this experiment, you are required to implement three threads, and a timer utilizing reusable libraries that you've developed in programming assignment #1.
- The first thread (console class) is needed to read continuously from the serial portw(dummy terminal), and once a message is received, it should display the message AS IS on the SBC's dummy terminal.
- The second thread will continually read from the SBC keyboard waiting for some keys to be pressed. Once it receives a keystroke, it should send it immediately to the serial port2. The "enter" key must be pressed at the SBC keyboard to send a message from the SBC to the dummy terminal.
- The third thread with use a timer to poll the IN-16 board. The reading from the IN-16 board will indicate which segment is occupied by the train. There are 16 segments are corresponding to the 16 bits read from the IN-16 board. If you put one train on the track, you should see only one bit is set to 1.
- In order to be able to read track status without any data loss, you need to adjust the timer interval (probably in the range of 1-500ms) according to the train's speed. **Hint:** Start from 500 ms, and decrement the timer interval in a step of 50ms.
- The call back function in a derived class of Timer class should poll the "IN16" module through a modified version of the I/O board client, which interacts with the I/O board server. It should then record those input and display them in the Screen with a readable format.
- The software designed should be portable and reusable for future labs. You should make sure that in your coding, each class has the following:
    a. Default constructor
    b. Virtual destructor
    c. Copy constructor
    d. Assign operator

Note: The readings obtained from the IN16 are in binary, while the tracd id within the code are integers. Hence it is important to convert the track ids into binary. This is achieved by using the `b_track = 1<<track_id;`. The value from the IN16 and the track id converted into binary, can then be compared by doing `if ((track & IN16) == track)`

The following pseudo codes could be used whiles creating various classes in this lab.

List 8: Bitpattern.hpp

```
#ifndef BITPATTERN HPP
#define BITPATTERN_HPP

/*********************************************************************
      BIT patterns that you may find useful.
*********************************************************************/
```

```
// Useful for Board

#define BIT0                  0x01
#define BIT1                  0x02
#define BIT2                  0x04
#define BIT3                  0x08

#define BIT4                  0x10
#define BIT5                  0x20
#define BIT6                  0x40
#define BIT7                  0x80

#define BIT8                 0x100
#define BIT9                 0x200
#define BIT10                0x400
#define BIT11                0x800

#define BIT12               0x1000
#define BIT13               0x2000
#define BIT14               0x4000
#define BIT15               0x8000

// Used for Unsigned Long ONLY(32-bit)
#define BIT16              0x10000
#define BIT17              0x20000
#define BIT18              0x40000
#define BIT19              0x80000

#define BIT20             0x100000
#define BIT21             0x200000
#define BIT22             0x400000
#define BIT23             0x800000

#define BIT24            0x1000000
#define BIT25            0x2000000
#define BIT26            0x4000000
#define BIT27            0x8000000

#define BIT28           0x10000000
#define BIT29           0x20000000
#define BIT30           0x40000000
#define BIT31           0x80000000

//Used for 48-bit ONLY
#define BIT32          0x100000000
#define BIT33          0x200000000
#define BIT34          0x400000000
#define BIT35          0x800000000

#define BIT36         0x1000000000
#define BIT37         0x2000000000
#define BIT38         0x4000000000
#define BIT39         0x8000000000

#define BIT40        0x10000000000
#define BIT41        0x20000000000
#define BIT42        0x40000000000
#define BIT43        0x80000000000

#define BIT44       0x100000000000
#define BIT45       0x200000000000
#define BIT46       0x400000000000
#define BIT47       0x800000000000

#define BIT48      0x1000000000000
#define BIT49      0x2000000000000
#define BIT50      0x4000000000000
```

```
        #define BIT51      0x8000000000000

        #define BIT52     0x10000000000000
        #define BIT53     0x20000000000000
        #define BIT54     0x40000000000000
        #define BIT55     0x80000000000000

        #define BIT56    0x100000000000000
        #define BIT57    0x200000000000000
        #define BIT58    0x400000000000000
        #define BIT59    0x800000000000000

        #define BIT60   0x1000000000000000
        #define BIT61   0x2000000000000000
        #define BIT62   0x4000000000000000
        #define BIT63   0x8000000000000000
#endif
```

List 9: BoardClient.cpp

```
#ifndef BOARDCLIENT CPP
#define BOARDCLIENT CPP

#include "BoardDef.hpp"

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

THIS FILE IS AN EXAMPLE OF HOW TO WRITE A CLIENT FOR THE BOARDSERVER.

*****************************************************************/
/*** Client Side of the code ***/
int client() {
    board data t msg, rmsg;

    register int fd = name open( ATTACH POINT, NAME FLAG ATTACH GLOBAL );
    if ( fd == -1 )
    {
      printf( "Couldn't find the Board Server !\n" );
        return EXIT FAILURE;
    }

    /* We would have pre-defined data to stuff here */
    msg.hdr.type    = 0x00;
    msg.hdr.subtype = 0x00;
    msg.serialMsgID = BOARD SerialMsgID;

    /* Do whatever work you wanted with server connection */
    for (msg.data=0; msg.data < 5; msg.data++)
    {
        memset(  msg.string, 0, 80 );
        memset( rmsg.string, 0, 80 );
        sprintf( msg.string, "Hi %d", msg.data );

        if ( MsgSend(fd, &msg, sizeof(msg), &rmsg, sizeof(rmsg) ) == -1)
        {
            break;
        }
            // 'rmsg' now has the message returned by the server
    }

    /* Close the connection */
    name close( fd );
    return EXIT SUCCESS;
}

int main(int argc, char **argv) {
  return client();
```

```
}
#endif
```

List 10: Boarddef.hpp

```
#ifndef BoardDef HPP
#define BoardDef HPP

#include "Standard.hpp"
/*****************************************************************
THIS FILE HAS SOME DEFINITIONS THAT ARE IMPORTANT, AND OTHER DEFINITIONS
THAT YOU MAY FIND USEFUL.
*****************************************************************/
#include
#include
#include

#define   BOARD IN16 ADDR     0x188
#define   BOARD IN16 ADDR2    0x18C //Address of second IN16 Board
#define   BOARD OUT16 ADDR    0x180
#define   BOARD CONTROL OFFSET   2
#define   BOARD OUT16 ENABLE  0x03
#define   VAL MAX             0xFF   // 255 (%1111 1111)

#define   BOARD SerialMsgID   ( 0xE10b0a4d )
#define   ATTACH POINT       "elan104-io"

// Normal Command

#define   BOARD IN16 READ     0x01
#define   BOARD OUT16 WRITE   0x02
#define   BOARD NORMAL RW     0x03

// Extra Command

#define   BOARD CONTROL IN    0x04
#define   BOARD CONTROL OUT   0x08
#define   BOARD CONTROL       0x0C
#define   BOARD ALL           0x0F
#define   BOARD REPLY DONE    0x80
#define   BOARD GET STATE     0x11

// Problem solving Command
#define   BOARD TEST SERVER   0x20
#define   BOARD KILL SERVER   0x40

// Switches definition
#define   SWITCH NORMAL      0xFF
#define   SWITCH 1           3
#define   SWITCH 2           2
#define   SWITCH 3           1
#define   SWITCH 4           0
#define   TRACK INSIDE       0
#define   TRACK OUTSIDE      1
#define   TRACK CURVED       0
#define   TRACK STRAIGHT     1
#define   SWITCH 4 CURVED    254
#define   SWITCH 4 STRAIGHT  253
#define   SWITCH 3 CURVED    251
#define   SWITCH 3 STRAIGHT  247
#define   SWITCH 2 CURVED    239
#define   SWITCH 2 STRAIGHT  223
#define   SWITCH 1 CURVED    191
#define   SWITCH 1 STRAIGHT  127
#define   SWITCH 4 INSIDE    SWITCH 4 STRAIGHT
#define   SWITCH_4_OUTSIDE   SWITCH_4_CURVED
```

```
#define   SWITCH 3 INSIDE    SWITCH 3 CURVED
#define   SWITCH 3 OUTSIDE   SWITCH 3 STRAIGHT
#define   SWITCH 2 INSIDE    SWITCH 2 CURVED
#define   SWITCH 2 OUTSIDE   SWITCH 2 STRAIGHT
#define   SWITCH 1 INSIDE    SWITCH 1 STRAIGHT
#define   SWITCH 1 OUTSIDE   SWITCH 1 CURVED

/* We specify the header as being at least a pulse */

typedef struct  pulse msg header t;

/* Our real data comes after the header */

typedef struct  board data

{

    msg header t hdr;
    unsigned long  serialMsgID;     // Message verification
    unsigned short input;     // 16-bit from IN16  board
    unsigned short output;    // 16-bit from OUT16 board
    unsigned char control out;      // Control setting for Out16
    unsigned char control in; // Control setting for In16
    unsigned char  command;     // Command to be processed.
    size t  data;       // Data packet sent
    char string[90-sizeof(unsigned short )-sizeof(unsigned char)];
      // Message [ 1 line max ]
    unsigned short input2;    // 16-bit from IN16  board
    unsigned char control in2;      // Control setting for In16

} board data t;

#endif
```

List 11: BoardServer.cpp

```
/*************************************************************************
DO NOT COMPILE AND RUN THIS FILE. IT IS ALREADY RUNNING AS A DAEMON. YOU
JUST NEED TO UNDERSTAND WHAT MESSAGES THIS BOARDSERVER UNDERSTANDS SO THAT
YOU CAN WRITE A CLIENT FOR IT.
*************************************************************************/
#ifndef BoardServer CPP
#define BoardServer_CPP__

#include
#include
#include
#include

#include "BoardDef-Kev.hpp"

/*****************************
- Default Address of ARCOM IN16 board is set to 180h
  in the lab they are set to 188h.  Second board
  is addresed at 18Ch.
- Define the address in BoardDef-Kev.hpp
- You will also need static unitptr t to contain the
  state of the of the Input Channels and Control register.
- May also have to define appropriate constants in Board-
  Def-Kev.hpp for purposes of message passing between
  server and client.  eg. BOARD_IN16_READ2 blah blah blah
- To compile do #g++ BoardServer-Kev.cpp -fno-builtin

*****************************/
```

```
#include "Standard.hpp"

uintptr t setHardware( uint64 t addr )
{
  ThreadCtl(  NTO TCTL IO, 0 );
  uintptr t ptr = mmap device io( 2, addr );
}

static uintptr t       ib state = NULL;
static uintptr t       ib ctrl  = NULL;
static uintptr t       ob state = NULL;
static uintptr t       ob ctrl  = NULL;

/*Adding Second IN16 */
static uintptr t        ib state2   = NULL;
static uintptr t        ib ctrl2    = NULL;
static USHORT                ib val2           = 0xFFFF;
/*********************/

static name attach t*  attach   = (name attach t*)(~0);

static USHORT          ob val = 0xFFFF;
static USHORT          ib val = 0xFFFF;
static int             done   = 0;

void SetupIO()
{
  ob state = setHardware( BOARD OUT16 ADDR );
  ib state = setHardware( BOARD IN16 ADDR  );
  ib state2 = setHardware( BOARD IN16 ADDR2  );

  ob ctrl  = setHardware( BOARD OUT16 ADDR + BOARD CONTROL OFFSET );
  ib ctrl  = setHardware( BOARD IN16 ADDR  + BOARD CONTROL OFFSET );
  ib ctrl2 = setHardware( BOARD IN16 ADDR2  + BOARD CONTROL OFFSET );

  out16( ob state, ob val );         // Set to default
  out8(    ob ctrl,    BOARD OUT16 ENABLE  );  //  Set  Array  0  &  1  ON
(Controllable)
  ib val = in16( ib state );
  ib val2 = in16( ib state2 );
}

void CleanUp()
{
  /* Remove the name from the space */
  printf( "Removing the attach name from the local name space.\n" );
  name detach(attach, 0);
}

/*** Server Side of the code ***/
int server()
{
    board data t msg, rmsg;
    int rcvid;

    /* Create a local name (/dev/name/local/...) */
    attach = name attach( NULL, ATTACH POINT, NAME FLAG ATTACH GLOBAL );
    if ( attach == NULL )
    {
      printf( "Could not attach a name to the local name space.\n" );
        return EXIT FAILURE;
    }
      printf( "Server Attached\n" );
    SetupIO();

    /* Do your MsgReceive's here now with the chid */
```

```
            //commented out memset cuz is causing problems
            //during compile and runtime

    while( !done )
    {
        memset(  msg.string, 0, 80 );
        memset( rmsg.string, 0, 80 );
        rcvid = MsgReceive(attach->chid, &msg, sizeof(msg), NULL);

        if (rcvid == -1)   /* Error condition, exit */
        {
            break;
        }

        if (rcvid == 0)    /* Pulse received */
        {
          printf( "Pulse received.\n" );
            switch (msg.hdr.code) {
            case  PULSE CODE DISCONNECT:
                /*
                 * A client disconnected all its connections (called
                 * name close() for each name open() of our name) or
                 * terminated
                 */
            printf( "Client disconnected.\n" );
                fflush( stdout );
                ConnectDetach(msg.hdr.scoid);
                break;

            case  PULSE CODE UNBLOCK:
                /*
                 * REPLY blocked client wants to unblock (was hit by
                 * a signal or timed out).  It's up to you if you
                 * reply now or later.
                 */
            printf( "Client wants to unblock.\n");
                break;
            default:
                /* A pulse sent by one of your processes? */
            printf( "Unknown pulse.\n" );
                fflush( stdout );
                break;
            }
            continue;
        }

        /* A QNX IO message received, reject */
        if (msg.hdr.type >=  IO BASE && msg.hdr.type <=  IO MAX) {
            MsgError( rcvid, ENOSYS );
            printf( "Rejected.\n" );
            continue;
        }

        if ( msg.serialMsgID != BOARD SerialMsgID ) {
        printf( "Wrong Serial ID...\n" );
        continue;
        }

        /* A message (presumable ours) received, handle */
        memcpy( &rmsg, &msg, sizeof( msg ) );

      register UCHAR cmd = msg.command;


      /* For both Board1 and Board 2 */
      if ( (cmd & BOARD CONTROL IN) == BOARD CONTROL IN )
{
```

```
        out8( ib ctrl, msg.control in );
        out8( ib ctrl2, msg.control in2 );
        }


      if ( (cmd & BOARD CONTROL OUT) == BOARD CONTROL OUT )
        out8( ob ctrl, msg.control out );

      if ( (cmd & BOARD IN16 READ ) == BOARD IN16 READ ) {
        ib val = in16( ib state );
        rmsg.input = ~ib val;
        ib val2 = in16( ib state2 );
        rmsg.input2 = ~ib val2;
      }

      if ( (cmd & BOARD OUT16 WRITE ) == BOARD OUT16 WRITE ) {
        ob val = ~( msg.output );
        out16( ob state, ob val );
      }
      if ( (cmd & BOARD GET STATE) == BOARD GET STATE ) {
        rmsg.control in  = in8( ib ctrl );
        rmsg.control in2  = in8( ib ctrl2 );
        //Get Board2 in the same time
        rmsg.control out = in8( ob ctrl );
        rmsg.output      = ~ob val;
        // input processed already.
      }
      if ( (cmd & BOARD TEST SERVER) == BOARD TEST SERVER ) {
        memcpy( &rmsg, &msg, sizeof( board data t ) );
      }
      if ( (cmd & BOARD KILL SERVER) == BOARD KILL SERVER ) {
        done++;
      }
      rmsg.command = BOARD REPLY DONE;

        rmsg.data++;
        strcpy( rmsg.string, "ACK" );
        MsgReply(rcvid, EOK, &rmsg, sizeof( rmsg ) );
    }
    CleanUp();

    return EXIT SUCCESS;
}

int main(int argc, char **argv) {
    return server();
}
#endif
```

List 12: Chat.cpp

```
#include "Serial.hpp"
#include "Console.hpp"

main() {

      Serial serial;
      Console console;

      // read from console (i.e. SBC keyboard) ->
      // write to serial port (i.e. dummy terminal screen)
      serial.attachDevice(&console);

      // read from serial (i.e. dummy terminal keyboard) ->
      // write to console (SBC screen)
      console.attachDevice(&serial);
```

```
	console.start();
	serial.start();

	console.join();
	serial.join();
}

/* in Device.cpp file, you can implement the attachDevice as follows:

void Device::attachDevice(Device* source) {
	// The source (of characters) is the "other" device object
	// that will be sending characters to this device
	this->source = source;
}

// since Device has been implemented as a Thread,
// a run method is needed to be implemented
Device::run() {
	while(1) {
		// whatever is read from the source is written to this device
		source->read(buffer, size);
		this->write(buffer, size);
		// you may have to overwrite this run method in your
		// Console class in order to echo back the characters
		// sent from the dummy terminal to the SBC
	}
}
*/
```

Hints:

Parallel I/O

Your programs run with user privileges. In order to access the IN16 and OUT16 boards, you need supervisor privileges. You can get around this problem by implementing a proxy that runs as a daemon process with supervisor privileges. You send a message to the proxy (called BoardServer.cpp), and the proxy, writes your message to the OUT16 board. In addition, the proxy may read a message from the IN16 board, and return a message back to the client (i.e. your program).

BoardServer is implemented already, and it runs in the background. You will notice this program starting when you reboot the SBC. All you have to do is write a client for this BoardServer.

You do not have to worry about reading/writing to the boards, just understand the protocol that the BoardServer understands and send it messages to tell it what action to carry out. The trick here is to set-up the connection between the client (your program) and the server (the proxy).

Refer to the following references, and the sample pseudocode that was given above:
www.qnx.com/developer/docs/momentics621_docs/neutrino/lib_ref/m/msgsend.html
www.qnx.com/developer/docs/momentics621_docs/neutrino/lib_ref/n/name_open.html
www.qnx.com/developer/docs/momentics621_docs/neutrino/lib_ref/n/name_attach.html

You should encapsulate all this “C style” stuff into appropriate C++ classes that will be easy to build on in future labs.

Serial I/O

Serial I/O on UNIX is quite similar to File I/O. In Lab 1, the "/dev/con1" and "/dev/ser1" and "/dev/ser2" devices would be accessed. Try to open, close, read, and write on these devices similar to these actions on files.

Refer to these references:
http://www.qnx.com/developer/docs/momentics621_docs/neutrino/lib_ref/o/open.html
http://www.qnx.com/developer/docs/momentics621_docs/neutrino/lib_ref/r/read.html
http://www.qnx.com/developer/docs/momentics621_docs/neutrino/lib_ref/w/write.html
http://www.qnx.com/developer/docs/momentics621_docs/neutrino/lib_ref/c/close.html

Answer the following questions:

1. At which timer speed does your program won't work anymore?
2. Why it won't work at that speed? Due to what phenomena?
3. What kind of information does the QNX message header part contain?
4. Would it be a good idea to skip the QNX message header and simply send data as a message? Analyse for both cases.
5. Would it be a good idea to skip the coding block that checks for the message header in order to gain some speed efficiency for our program? Explain for both cases and give the assumptions that you've made.

Checkpoints: Your performance in this lab will be evaluated based on the following operations properly being implemented.

1. The timer is polling the IN16 at a reasonable rate to detect input changes via the speed change of the train.
2. The output from IN-16 board must be displayed on the screen in a readable format.
3. The chat program works perfectly on both sides (bi-directional):

   - Text typed on the serial port terminal is received on the PC and displayed as it is on the screen.
   - Text typed on the PC is received on the serial port terminal and displayed as it is on the terminal screen.

7 EXPERIMENT# 2: Train Controller

Objectives:

- To be familiar with the train set-up.
- To understand how the train controller works in order to change the speed and direction of each train and also to change the light status of each train.
- To be able to detect where the train is on the track system.
- To measure accurately the speed of each train for each given speed states.
- To be able to do "X" number of laps in a given time (+/-).

Track Diagram:

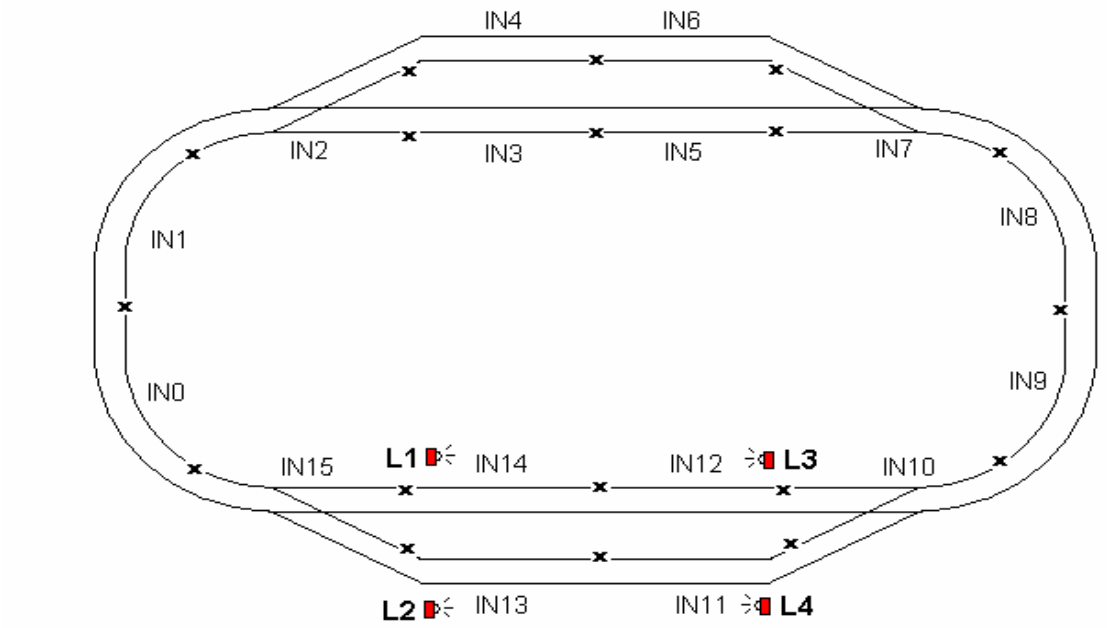


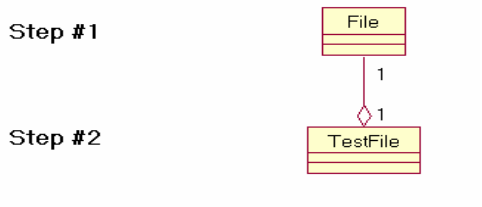
Figure 23: Track layout for experiment #2

Note: In this lab, we are only using segment 0-1-2-3-5-7-8-9-10-12-14-15. That means all of switches should be located straight so as to let the train use the straight lines rather than the curve lines. You should just ignore the segment 4,6,11,13, which will be used in lab3 and lab4. All of switches and traffic lights should be ignored too.
There are three types segments are involved in the lab:

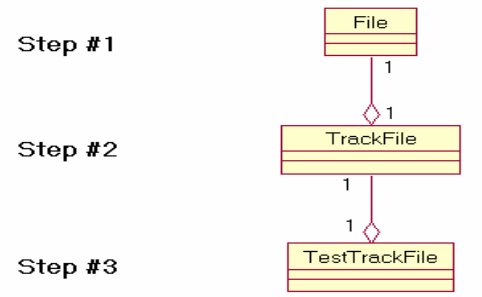
1) 0,1,8,9: Curved segments
2) 2,7,10,15: Straight switched segments
3) 3,5,12,14: Straight segments

Suggested Classes:

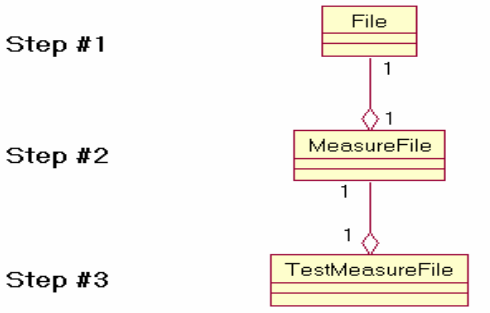
STAGE—1: Create a file class and a test program for that class.



STAGE—2: Create a track file class that will read and parse a *.trk file and store the corresponding track layout data structure representation in memory.
Provide a test program to ensure the correct behavior for this class.



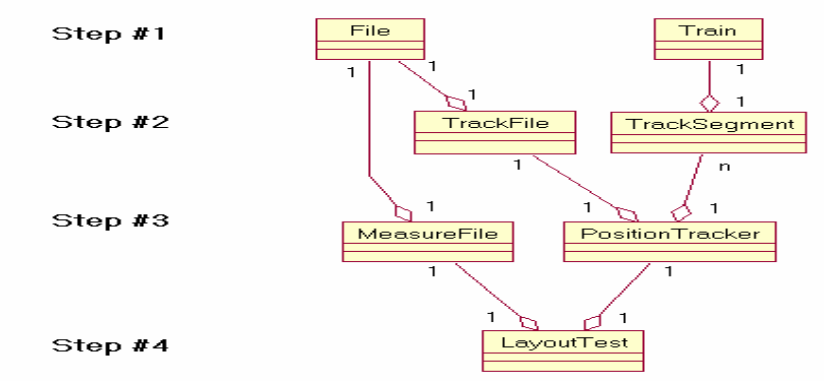
STAGE—3: Create a measure file class that will store or retrieve the elapsed time, speed and performance of each train on linear or curved tracks for future retrieval.
Provide a test program to ensure the correct behavior for this class.



STAGE—4: Each track segment entry in the track file should be stored inside a track segment class, that will contain all the respective information about that small segment, including the presence or the absence of a train on such a segment.

The position tracker will take care of putting together all the segments in some sort of data structure, in order to deal with the big picture, which is the complete track layout and manage which trains go where, at what speed and when.

Each single class can now be assembled together to create a small test program that will ensure the correctness of these classes.



Suggested Class Hierarchy Diagram:

The overall class hierarchy diagram for this experiment would look like as follows:

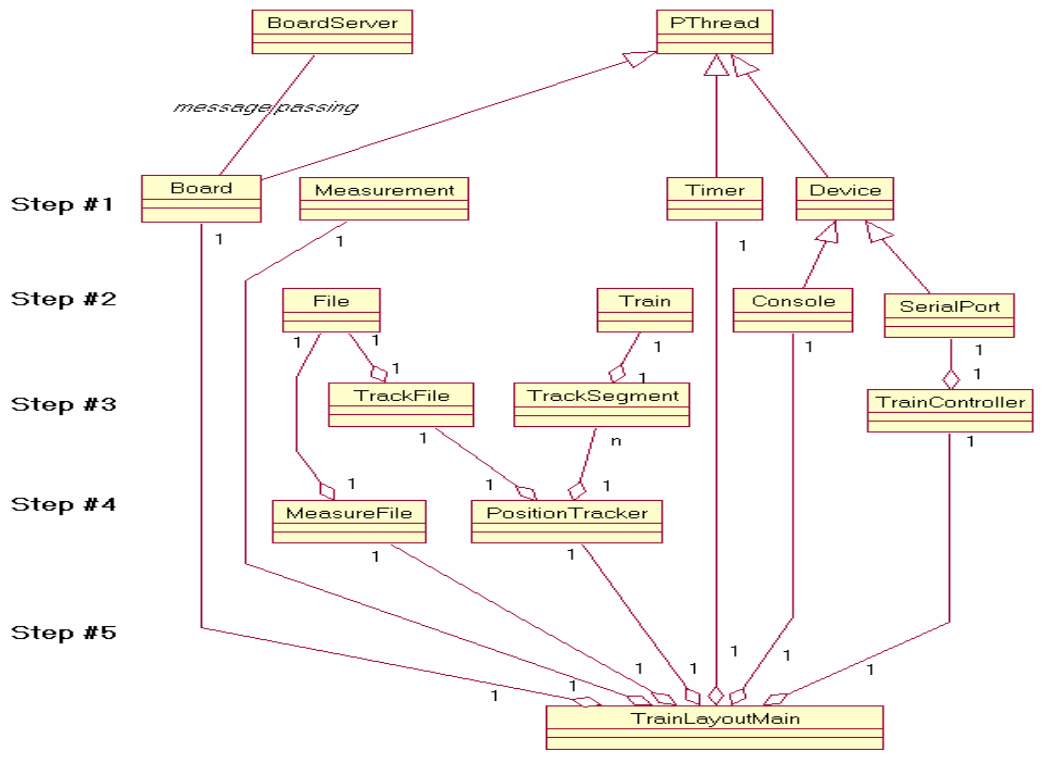


Figure 24: Overall class hierarchy diagram for experiment #2

The figures for lab 2, 3, 4 and 5 are the same. This is due to the fact that the classes implemented in experiment 2 will be reused for all the other experiments. The only thing that evolves is the complexity of the algorithm in some of those classes. The association with the other classes must be done in experiment 2, but the real implementation of the code, if not used, in that experiment will be done in the forthcoming experiments. For instance, the class Track Segment contains an object called Semaphore. So, all the code related to the semaphore can be only implemented in experiment 3, since this is the only experiment where such a feature is really useful.

Lab Description:

Part(A):

In order to know where the train will be you should start by developing a Position Tracker class that will be used to read the track configuration file and to have the track set-up in memory in order to exactly know what's happening.

Part(B):

Modify your code from the previous experiment in order to meet the serial port Easy DCC specification, in order to be able to set the train speed and attributes on the track set-up. Try to get the train running at a given desired speed. Measure the time it takes to do a complete loop at each speed.

Part(C):

Write some sort of Train Manager class that will be able to control the speed of a given train so it can stop precisely at a given location. Meet each of the stated objectives in order to complete this experiment.

Suggested Design:

You need to design a class that will represent the track layout utilizing a data structure. The data structure will essentially capture the connectivity of different tracks or zones of a given layout.

Train Position Tracker

Create a class that represents the track layout inside any one of the following data structure:
2 (CCW and CW) x Array of single link-list
2 (CCW and CW) x Array of arrays (Boolean: Immediate path exist or not)
2 (CCW and CW) x 2D array (adjancy matrix)
Array of double link-list (prev/next = CCW/CW)
Array of prev/next (CCW/CW) arrays

For example, if you have 12 tracks, circular model, and you choose the array of prev/next arrays, basically you have:

CCW		CW
Prev	Curr	Next
[15]	← [0]	→ [1]
[0]	← [1]	→ [2]
[1]	← [2]	→ [3]
[2]	← [3]	→ [5]
[3]	← [5]	→ [7]
[5]	← [7]	→ [8]
[7]	← [8]	→ [9]
[8]	← [9]	→ [10]
[9]	← [10]	→ [12]
[10]	← [12]	→ [14]
[12]	← [14]	→ [15]
[14]	← [15]	→ [0]

So, basically, if you are in segment 5 and you ask yourself what are the possible paths to go CW, your first choice is 6 and that's also your only choice.

For example, if you have a track setting like this:

[4] ← [5] → [6,7,8]

Your first choice would be 6, second choice would be 7, and third choice would be 8.
Normally, you would have to find the shortest path avoiding collisions, but since all layouts are simple, this is not an issue discussed in this lab.

Modeling:

You should come up with a physical modeling that approximates the train behaviour,

- Straight track(3,5,12,14) (14 steps)
- Straight switched track(2,7,10,15) (14 steps)
- Curved track(0,1,8,9) (14 steps)

On a pseudo-linear path, assume that if the speed is gradually increased and decreased at a fixed rate, we can model it linearly.

The point is that even if the track is curved, the velocity is still linear since the train must follow the path. In case of small deceleration/ acceleration, some small measurements can be made to "adjust" the model to linearize it for curved tracks.

The main objective is to get a near "reality" model that can be used for track reservation and scheduling in future experiments.

Even if your model is not perfect due to some timing issues or latency reading, it can be updated/corrected/modified to follow the reality. For instance, if your model says that train A should be on track 2 now, but it is not there yet, you can revise the position/speed in order to fix the timing issue, so that train A will follow the model once again as it touches track 2, few milliseconds later.

Table and Measurements:

In order to have a complete model, you must make few measurements on your own. In fact, use your Measurement class of programming assignment #1, and calculate how much time it takes at a given speed to travel on track 0,1,2,3,,5,7,8,9,10,12,14,15.

That is, once you touch track "0", you start "Measure" and you stop "Measure" as soon as you touch track "1".

In order to do this, create a data-structure and save those results into a file, so that you can recover it later on, or load it on the fly.

```
typedef struct  speed data
{
  UCHAR   track No;
  UCHAR   speed step;
  double  elapsed time;
} speed_data_t;
```

Basically you use read and write function to load/save those data inside a "speed.dat" file.

```
 void DataLog::SaveData( UCHAR trk, UCHAR spd, double time )
{
  speed data t data;

  data.track No       = trk;

  data.speed step     = spd;
  data.elapsed time   = time;
  int err             = write( fd, data, sizeof( data ) );
  MC OnError( err, Error, "Cannot write speed table data." )
}
```

And you do the opposite for read.

The next aspect is that you must use your Board class to link the Sensors on IN16 with your data-structure and model, updating everything in real-time.

Ideally, you should update only if the state has changed, else return to your Board Timer class.

Note: If a train switches from segment 0 to segment 1, the IN16 reading may flicker between the following possible values:

- 0x00 // train lost contact with the track,
- 0x01 // train is on segment 0,
- 0x11 // train is between segment 0 and segment 1,
- 0x10 // train is on segment 1.

The following pseudo-codes could be used while creating various classes in this lab:

List 13: EasyDCC.cpp

```
#include "EasyDCC.h"
```

```
// Train speed functions
//  -1  Emergency stop
//   0  Stop
//   1   1st speed step:  1/14 * MAX
//   2   2nd speed step:  2/14 * MAX
//   3   3rd speed step:  3/14 * MAX
//   4   4th speed step:  4/14 * MAX
//   5   5th speed step:  5/14 * MAX
//   6   6th speed step:  6/14 * MAX
//   7   7th speed step:  7/14 * MAX
//   8   8th speed step:  8/14 * MAX
//   9   9th speed step:  9/14 * MAX
//  10  10th speed step: 10/14 * MAX
//  11  11th speed step: 11/14 * MAX
//  12  12th speed step: 12/14 * MAX
//  13  13th speed step: 13/14 * MAX
//  14  14th speed step: 14/14 * MAX
// EasyDCC::getHexSpeed
// 0x00 = Stop
// 0x01 = Emergency Stop
// 0x11 = Emergency Stop
// 0x02 =  1/14
// 0x12 =  2/14
// 0x03 =  3/14
// 0x13 =  4/14
// 0x04 =  5/14
// 0x14 =  6/14
// 0x05 =  7/14
// 0x15 =  8/14
// 0x06 =  9/14
```

```
// 0x16 = 10/14
// 0x07 = 11/14
// 0x17 = 12/14
// 0x08 = 13/14
// 0x18 = 14/14
UCHAR EasyDCC::getSpeedHex( int spd )
{
  //
  // Emergency stop
  //
  if ( spd <  0 ) return 0x01;
  // Stop the train
  if ( spd == 0 ) return 0x00;
  assert( spd <= 28 );
   // if ( spd ) { even = 0x00; } else { even = 0x10; }
  UCHAR even = ( (spd & 0x01) ? 0x00 : 0x10 );

   // Same as: val = (spd+1)/2  + 1;
  UCHAR val  = spd;
  val++;
  val >>= 1;  // Shift Right by 1   OR   Divide by 2
  val++;

  return even + val;
}

void  EasyDCC::sendMsg(  UCHAR  trainid,  int  speed,  bool  clockwise,  bool
lights )
{
      // send your message as if you were sending a
      //string out the serial port for lab1
}

// EasyDCC::Msg2Str
// This function encodes trainID, speed, direction (clockwise = true,
// counter-clockwise = false), and light on/off,
// into a string that is sent to
// the EasyDCC takes the string and then sends information to
// control the behaviour (i.e. speed, direction, lights) of the train
// identified by trainID.

char*  EasyDCC::Msg2Str(  UCHAR  trainid,  int  speed,  bool  clockwise,  bool
lights )
{

  char buf[13];
  memset( buf, 0, 13 );

  register UCHAR checksum = trainid;
  register UCHAR command  = 0x40;      // Set Speed 14 speed steps

  if ( clockwise ) command += 0x20;

  if ( lights    ) command += 0x10;
  command += getSpeedHex( speed );

  // XOR the parameters to get checksum = trainid ^ command.
  checksum ^= command;

  sprintf( buf,"Q %02X %02X %02X%c%c",trainid, command, checksum,13,0);

 /* printf( "Train #%d is going at speed %d ", trainid, speed );

  if ( clockwise ) {
    printf( "in the forward direction " );
  } else {
    printf( "in the reverse direction " );
```

```
  }
  if ( lights ) {
    printf( "with the lights ON \n" );
  } else {
    printf( "with the lights OFF \n" );
  }
*/
  // printf( "%s", buf );

  return strdup( buf );
}
```

List 14: EasyDCC.h

```
#ifndef EASYDCC H
#define EASYDCC H
#include "Device.h"
class EasyDCC : public Device {

      public:
            EasyDCC();
            virtual ~EasyDCC();
            void sendMsg( UCHAR trainid, int speed, bool clockwise, bool
lights);
      private:
            EasyDCC(const EasyDCC& );
            const EasyDCC& operator=(const EasyDCC&);
            UCHAR getSpeedHex( int spd );
            char* Msg2Str( UCHAR trainid, int speed, bool clockwise, bool
lights);
};
#endif
```

Hints:

Position Tracker

In order to "track", or monitor the positions of trains as they move along the track, you first need: (1) a model of the track, and (2) an interface to the sensors that will tell you if a train is present on a segment or not.

The model, or layout, of the track should be read from a file. You describe the layout any way you wish in your file. You then need to parse your file and store info into a data structure.

As for the data structure, use whatever type you see fit (linked-lists, arrays, vectors, etc). It is important to remember that one segment may be connected to a switch, which will give the option of going to one or two connected segments.

For example, say your class for storing track information is called Track, and you have a method called getNextSegment(...) to determine the next connected segments, as follows:

```
// constructor reads track layout from file called ovalTrackLayout.trk
Track track("ovalTrackLayout.trk");
int currSegmentID = 0;
int direction = CCW; // or left, or right, or whatever label you wish

// ... eventually you say:

??? = track.getNextSegment(currSegmentID, direction);
```

What does this method return, given the direction of train movement, and the current segment? What if you have only one connected segment? What if you have two connected segments because there is a switch up ahead? What to do? This is for you to decide. Talk about these types of design issues in your report.

Once you have a model of the track, you read the IN16 board at a regular interval (refer to Lab1, parallel I/O with Timer::tick()). The IN16 board is your interface to the track sensors. Each one of those bits corresponds to a track segment. Match the bit to your model of the track and you will be able to figure out that "there is a train on track segment 4 since BIT4 was 1 and all others were 0", for example. Remember that you will NOT know what train is on track segment 4 immediately. It is the responsibility of the PositionTracker to "figure out" what train that might be, given some initial conditions. In other words, you can read BITS from the track, but not "which-train-on-which-track". That's for the Position Tracker to determine.

Here are some VERY IMPORTANT cases to consider:
- What if train is on two segments at the same time?
- What if no "1" is picked up when you read IN16 has my train vanished?
- What is my polling interval with respect to my train speed?

NOTE: You can test the IN16 board with the following test program:

Type /public/testIn16.x

As your train moves along the oval track, you should see one or more "1" on the SBC monitor. This "1" corresponds to the presence of a train detected on a segment. You will see a "0" for each segment that does not have a train on it. For example IN16:
0000000000010000: means train detected on segment 4 (i.e. BIT4, starting from right, which is BIT0)

Serial Communication with EasyDCC

Serial communication with the EasyDCC (controller for trains) is easy if you got your "chat" working in lab1. The methods to encode your messages for the EasyDCC are given in EasyDCC.h/.cpp. You take these messages and send them out to the serial port1.

Here are some more VERY IMPORTANT things to consider:
- What if my message gets lost somewhere or my train never gets the message? Is there some sort of acknowledgment message sent back?
- How long will it take for my train to respond to my message (queuing delay in EasyDCC, transmission time, etc) ... are these relevant? How many messages can I send to the EasyDCC in a given amount of time without flooding it (congestion)?

NOTE: You can test the serial communication equipment with

```
/public/testEasyDCC.x
```

Put a train on the oval track. Run the program and enter trainID, speed, and direction. To stop the train, press the RESET button on the EasyDCC or run the program again, and set the speed to 0 (or -1 for emergency stop). The program sends ONE message and then terminates.

Measurements & Control

This part should be done in two phases:

1) Take Measurements
- move train to a particular segment
- set speed of train to X, direction to Y, do one turn around track
- for each new segment encountered by the train, stop the old measurement timer, start a new measurement timer. With the elapsed time that it took for the train to traverse the last segment at speed X, calculate the speed in m/s (Length of segment is in your Track file).
- write trainID, trainSpeed, direction, SegmentID, m/s, time, whatever you think necessary to a file.
- loop until all speeds (or at least odd speeds) have been measured, in both directions.

2) Use Measurements to make predictions
- based on the measurements taken above, read the results from a file and try to make predictions of how long it will take to get to another segment. For example, you may have printouts that look like:
Train 6 on segment 4 going forward: It will take approximately 2.085 seconds to get to segment 6 at speed 17 (0.311 m/s).

Extra points that would help develop a better code:

1) An easy way of getting a Position Tracker, without having to deal with link list, is to use the STL vector class, to get an easy 3D array.

```
vector<  int  >                          vect_1D( x, 0 );

vector<  vector< int >  >                vect_2D( y, vect_1D );

vector<  vector<  vector< int > > >      vect_3D( z, vect_2D );
```

Don't put the >>> together, keep the spaces.
Don't write vector<vector<int>> since this is interpreted as operator>>
This will give you a table array with the following range:
Table [0][0][0] to table[z-1][y-1][x-1] filled with zeros.
e.g. x = 2, y = 2, z = 16 gives you:
table[0][0][0] to table [15][1][1] filled with zeros.
Basically, you should set:
- z maps to the segment ID
- y maps to the direction CW/CCW
- x maps to the possibilities from 0..n

where possibilities are all the alternative paths that the train can follow at that segment.

e.g. Y junction path

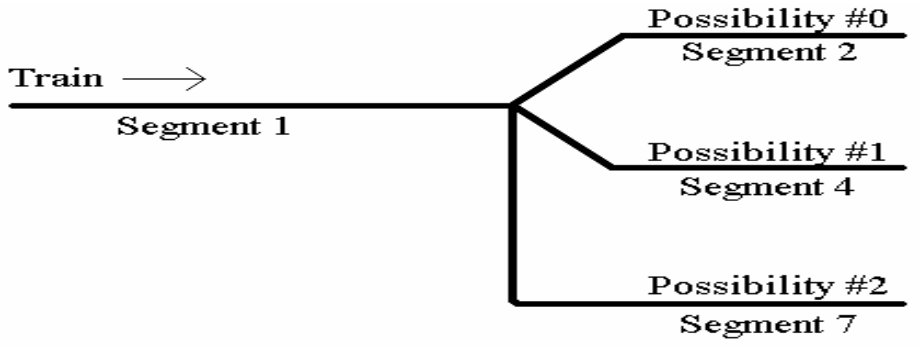


Figure 25: Train path possibilities

z value in the vector<> constructor should be the maximum segment plus 1 which is 8.

y is two, CW/CCW.
x is the maximum number of possibilities as indicated in the % initialization line.

e.g. Where can the train go from segment 4 in CW direction ?
Possibility #0: table [4][0][0]
Possibility #1: table [4][0][1]
Possibility #2: table [4][0][2]

e.g. Where can the train go from segment 6 in CCW direction ?
Possibility #0: table [6][1][0]
Possibility #1: table [6][1][1]
Possibility #2: table [6][1][2]

If a possibility does not exist, set it to -1, since 0 represents a possibility to go directly to segment [0].

For instance, if you only have one choice, the possibility #0 should indicate that choice, the possibility #1 should be -1, the possibility #2 should be -1, etc.

2) A track is consisting of several segments. So, the train layout for this lab has 12 segments. Semgent has the following variables:

int indexHW = Segment Name: Segment bit on In16 board.

Train *train = Pointer to the train on that segment, NULL else.

Sempahore reserved = Like for a Bounded Buffer of size 1, reserved the segment.

Semaphore lock = Lock the Segment variables

isEmpty()/isFull() = Read the semaphore value

For the controller, look for the speed pseudo-code file in the directory mentioned in the various links section in this manual and it should use Device class developed for Lab1.

There is only one thread, which is for the confirmation from the Easy DCC.

Model is simply:

speed or velocity = Length of a segment / DeltaT

speed_Curved = average(0.4425 meters / DeltaT)

speed_Straight = average(0.3720 meters / DetlaT)

DeltaT = Measurement.elapsed() = Measurement.stop() - Measurement.start()

3) circle.trk contains all the information about the current track set-up. It is given as follows:

```
#!./FileReader
%0-15,1,1;
#
# External curved segment 0,1,8,9
@C0=0.3630;
#
# Internal straight segment 3,5,12,14
@S1=0.2830;
#
# Straight switch segment
@S2=0.1250;
#
#

$0:(C0),1,1,L(1),R(15),S(-1);
$1:(C0),1,1,L(2),R(0),S(-1);
#
$2:(S2),1,1,L(3),R(1),S(-1);
$3:(S1),1,1,L(5),R(2),S(-1);
#
$5:(S1),1,1,L(7),R(3),S(-1);
$7:(S2),1,1,L(8),R(5),S(-1);
#
$8:(C0),1,1,L(9),R(7),S(-1);
$9:(C0),1,1,L(10),R(8),S(-1);
#
$10:(S2),1,1,L(12),R(9),S(-1);
$12:(S1),1,1,L(14),R(10),S(-1);
#
$14:(S1),1,1,L(15),R(12),S(-1);
$15:(S2),1,1,L(0),R(14),S(-1);

# A straight  = 1
# A curved   = 2
# B straight  = 4
# B curved   = 8
# E straight  = 16
# E curved   = 32
# F straight  = 64
# F curved   = 128

#!./FileReader
==============
```

This line is a comment, since it starts with the token '#'.
It's also used so that the OS thinks it's a script where the content is readable by a program called FileReader in the local directory.

%0-15,1,1;
=========
This line says that the segment/track starts at bit 0 and ends at bit 15.
Also, it indicates that the maximum number of paths CW = Left is 1 and the maximum number of paths CCW = Right is 1 too.
Therefore:
Segment [0] = %0000 0000 0000 0001

Segment [1] = %0000 0000 0000 0010
Segment [2] = %0000 0000 0000 0100
Segment [3] = %0000 0000 0000 1000
Segment [5] = %0000 0000 0010 0000
Segment [7] = %0000 0000 1000 0000
Segment [8] = %0000 0001 0000 0000
Segment [9] = %0000 0010 0000 0000
Segment [10] = %0000 0100 0000 0000
Segment [12] = %0001 0000 0000 0000
Segment [14] = %0100 0000 0000 0000
Segment [15] = %1000 0000 0000 0000

```
@C0=0.3630;
==========
```

This line says that the curved segment labelled #0 measures 0.3630 meters. There might be many curved segments of different length, labelled C2, C3, C4, etc.

```
@S1=0.2830;
==========
```

This line says that the straight segment labelled #1 measures `0.2830` meters. There might be many curved segments of different length, labelled S2, S3, S4, etc.

```
@S2=0.1250;
==========
```

This line says that the straight switch segment labelled #2 measures `0.1250` meters. There might be many curved segments of different length, labelled S2, S3, S4, etc.

```
$1:(C0),1,1,L(2),R(0),S(-1);

==================
```

Segment [1] is bit 1, curved of length C0, 1 path left, 1 path right, left segment is 2, right segment is 0. There is no switch associated with this track segment.

```
$3:(S1),1,1,L(5),R(2),S(-1);
==================
```

Segment [3] is bit 3, straight of length S1, 1 path left, 1 path right, left segment is 5, right segment is 2. There is no switch associated with this track segment.

```
$10:(S2),1,1,L(12),R(9),S(-1);
```

Segment [10] is bit 10, straight switch path of length S2, 1 path left, 1 path right, left segment is 12, right segment is 9. There is no switch associated with this track segment.

$7:(S2,C2),1,2,L(8),R(5,6),S(1,2);

This format is going to be used in the lab 3,4 and 5 where the switch is used. It is not used in this lab.

Segment [7] is bit 7, the straight path is of length S2, the curved path is of length C2. To go from left (8) to right (5), the straight path, you must set Switch (1) which is bit 1. To go from left (8) to right (6), the curved path, you must set Switch (2) which is bit 2.

5) The screen output should look like this:

Initializing Position tracker from circle.trk... Done!
Logging result in file: circle.spd
Taking measurements for Forward Speed Step #1:
Waiting until train #3 in position...

Logging started for Track [0].
Track [0] average speed = xxx.xxxx m/s
Track [1] average speed = xxx.xxxx m/s
Track [2] average speed = xxx.xxxx m/s
Track [3] average speed = xxx.xxxx m/s
Track [5] average speed = xxx.xxxx m/s
Track [7] average speed = xxx.xxxx m/s
Track [8] average speed = xxx.xxxx m/s
Track [9] average speed = xxx.xxxx m/s
Track [10] average speed = xxx.xxxx m/s
Track [12] average speed = xxx.xxxx m/s
Track [14] average speed = xxx.xxxx m/s
Track [15] average speed = xxx.xxxx m/s

Average speed at Forward Speed Step #1:
Straight track average speed = xxx.xxxx m/s
Straight switch track average speed = xxx.xxxx m/s
Curved track average speed = xxx.xxxx m/s
A complete circle trip took xxx.xxxx ms to travel x.xxxxx meters.
Data logged.
Taking measurements for Forward Speed Step #2. ……
Taking measurements for Reverse Speed Step #1. ……

Part B: Model testing
- Run the train at speed X asked by user.
- Show your time prevision and real-time elapsed time
- Update in real-time the time prevision as needed.

Prevision for Speed Step #8:
Track [0->1] at t = xxx.xxxx ms.
Track [1->2] at t = xxx.xxxx ms.
Track [2->3] at t = xxx.xxxx ms.
Track [3->5] at t = xxx.xxxx ms.
Track [5->7] at t = xxx.xxxx ms.
Track [7->8] at t = xxx.xxxx ms.
Track [8->9] at t = xxx.xxxx ms.
Track [9->10] at t = xxx.xxxx ms.
Track [10->12] at t = xxx.xxxx ms.
Track [12->14] at t = xxx.xxxx ms.
Track [14->15] at t = xxx.xxxx ms
Track [15->0] at t = xxx.xxxx ms

Real-Time data:
Track [0->1] at t = xxx.xxxx ms.
Track [1->2] at t = xxx.xxxx ms.
Track [2->3] at t = xxx.xxxx ms.
Track [3->5] at t = xxx.xxxx ms.
Track [5->7] at t = xxx.xxxx ms.
Track [7->8] at t = xxx.xxxx ms.
Track [8->9] at t = xxx.xxxx ms.
Track [9->10] at t = xxx.xxxx ms.
Track [10->12] at t = xxx.xxxx ms.
Track [12->14] at t = xxx.xxxx ms.
Track [14->15] at t = xxx.xxxx ms
Track [15.5] at t = xxx.xxxx ms. (stop the train)

Waiting for transition.....None in less than xxx.xxxx ms.
Train stopped on Track #15.

Checkpoints: Your performance in this lab will be evaluated based on the following operations properly being implemented.

- Proper and meaningful measurement.
- The actual train speed is proportional to the desired speed.
- The calculated train speed is proportional to the desired speed.
- Measurements are taken from the head of the train touching track x to the head of the train touching the track x+1.
- The train stops within reasonable time.

8 EXPERIMENT# 3: Manoeuvre of Two Trains with Switch

Objectives:

In this lab experiment, the software interface being developed is expected to control the movement of two trains, which are running in the opposite direction to each other. It is important to ensure that there are no collision and/or deadlock conditions. One of the trains can stop momentarily or slow down to let the other train pass by. All of them are achieved by controlling the switches.

You will be making an extensive use of all software components being developed in earlier lab experiments. In particular, your position tracker and semaphore classes will be quite helpful.

Track Diagram:

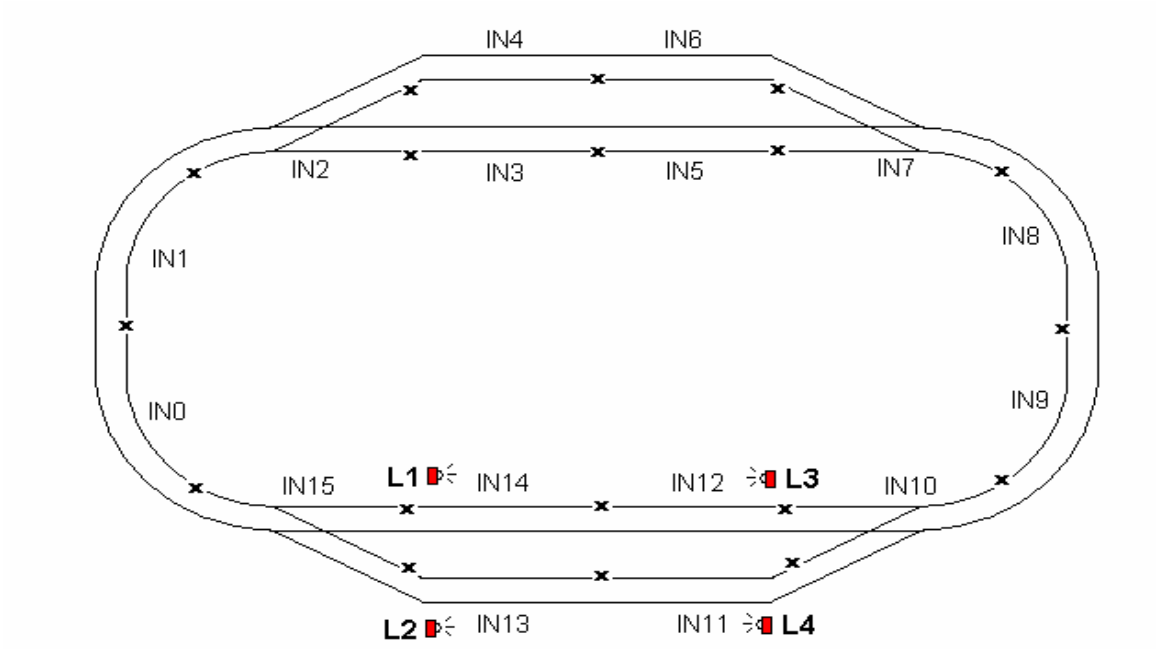


Figure 26: Track layout for experiment #3

Note: In this lab, the lights are not considered. Only the switches will be used.

Suggested Class Hierarchy Diagram:

The overall class hierarchy diagram for this experiment would look like as follows:

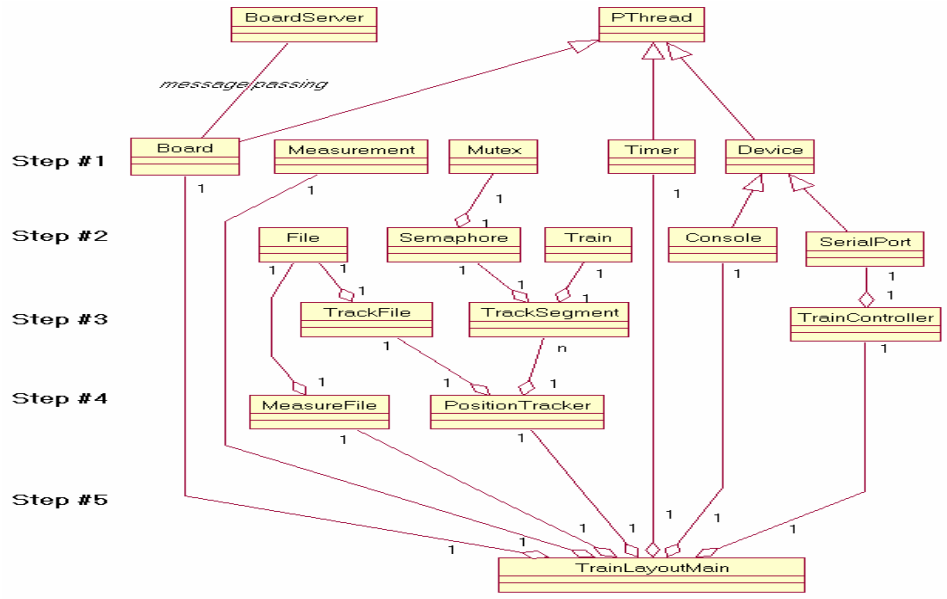


Figure 27: Overall class hierarchy diagram for experiment #3

Lab Description:

Consider Train A and B going in clockwise and anti-clockwise direction, respectively. If Train A is on a track segment 15-1, it must attempt to acquire track segment 2-3-5-7 or 2-4-6-7. If Train A can acquire it, it goes ahead and enters the selected bypass zone. If it fails to acquire it, that is, Train B has already reserved one of the track segments in the bypass zone, then Train A must wait on either 15 or 1 track segment. Train B, after entering the bypass zone (say 4-6), will now attempt to acquire track segment 1-15, and would fail because 1-15 is currently held by Train A. Train B must now release switch 2 and 7. This would allow Train A to acquire track segment 2-3-5- 7 and cross the bypass zone. As Train A enters track segment 8-9, Train B can acquire switch 2 and segment 1-15 and moves ahead.

Refer to the figure shown above for the following discussion. A new track file called "oval.trk" will be provided.

Switch Related Issues:

Notice that there are two electro-magnetic coils or solenoids per switch for each direction. A table describing configuration of the bit to be set on the Outl6 board for each switch direction to connect to the desired track segment is provided in the "BoardDef.hpp" file. The bit must be set for a maximum period of 200ms. Basically, you activate the coil and once the switch is positioned, you turn it off in order to avoid over-heating the coil. Also, there is a hardware protection, which prevents you to turn on again the same "solenoid" within 1sec. However, you could turn the switch on to the other direction within 1sec.

You will be making an extensive use of software components being developed in experiment#2 for this lab, especially position tracker, etc. You must make sure that the measurement part of your experiment#2 works properly so that you can estimate the speed and/or distance to slow down the train in time.

For instance, if you are at x0 and you are going at an average speed of v -avg. You could approximately determine that the train will be at a space [x0 + v-avg * dt] in "dt" time if the speed is maintained at v-avg.

To test the switches (turnouts), you can use:
/public/testSwitch.x
Then you can input the number to set the positions of the switches.

Suggested Design:

- The Switches are active high and the LED's are active low.
- The server complements the state for the LED's, so that, if you send 0x01, the server will send 0xFE, which will light up LED 0.
- The switches are active high, so from the track file, you must send Bit0.
- Send 0x00 to the OUT16 board to turn off the switches.

```
inline void switch( int b )
{
  if ( b < 0 ) return;   // No switch
  board.fastWrite( ~b ); // Activate the coil
  board.fastWrite(  0 ); // Turn it off.
}
```

- The backtracking or moving backward is not allowed.
- You must make sure that two trains are never present in the track segment (2-1-16-15) or (7-8-9-10) at any time.
- If you are in the bypass lane (2-4-6-7, 2-3-5-7, 15-14-12-10, 15-13-11-10), and if you need to stop a train, you must stop the train on either 4 or 6 (and similarly for other segments), and never on 2 or 7 (and similarly for the other side).
- If you are in track segments (2-1-16-15) or (7-8-9-10), the train must stop on track segments l or 16 (8 or 9).
- You must make sure that a train never stops for long time on track segments 1-16 or 8-9.
- In order for you to be able to control these situations, you will need to have six semaphores, one each for the following track segments: 4-6, 3-5, 8-9, 14-12, 13-11, and 1-16. Additionally, you would need four more semaphores for switch zone 2, 7, 10, 15.
- If a train needs to pass the bypass zone, it must acquire both switch zone 2 and 7 (or 10 and 15). On the other hand, if a train needs to stop at a bypass zone, it must stop at track segment 4-6 (or other similar segments), and then must release both switch zone 2 and 7 (or 10 and 15) for the other train to acquire them.
- If a train cannot acquire a track segment (or zone), it must then wait.
- You must also make sure that your "position tracker" is not accessed simultaneously by the two train-threads.
- Use the following file while doing your coding:

List 15: Oval.trk

```
#!./FileReader
%0-15,2,2;
#
# External curved track 0,1,8,9
@C0=0.3630;
#
```

```
# Internal straight track 3,5,12,14
@S1=0.2830;
#
# Internal curved track 4,6,11,13
@C1=0.2855;
#
# Straight switch path
@S2=0.1250;
#
# Curved switch path
@C2=0.1258;
#
#

$0:(C0),1,1,L(1),R(15),S(-1);
$1:(C0),1,1,L(2),R(0),S(-1);
#
$2:(S2,C2),2,1,L(3,4),R(1),S(2,3);
$3:(S1),1,1,L(5),R(2),S(-1,2);
$4:(C1),1,1,L(6),R(2),S(-1,3);
#
$5:(S1),1,1,L(7),R(3),S(5,-1);
$6:(C1),1,1,L(7),R(4),S(6,-1);
$7:(S2,C2),1,2,L(8),R(5,6),S(1,2);
#
$8:(C0),1,1,L(9),R(7),S(-1);
$9:(C0),1,1,L(10),R(8),S(-1);
#
$10:(S2,C2),2,1,L(12,11),R(9),S(6,7);
$11:(C1),1,1,L(13),R(10),S(-1,7);
$12:(S1),1,1,L(14),R(10),S(-1,6);
#
$13:(C1),1,1,L(15),R(11),S(5,-1);
$14:(S1),1,1,L(15),R(12),S(4,-1);
$15:(S2,C2),1,2,L(0),R(14,13),S(4,5);

# A straight  = 1
# A curved   = 2
# B straight  = 4
# B curved   = 8
# E straight  = 16
# E curved   = 32
# F straight  = 64
# F curved   = 128
```

a) The last part S() is for the switch bit. $13:(C1),1,1,L(15),R(11),S(5,-1);
b) In order to go from 13->15, switch Bit5 must be on: switch(0x10);
c) In order to go from 13->11, don't switch anything.
   $15:(S2,C2),1,2,L(0),R(14,13),S(4,5);
d) In order to go from 15->14, switch Bit4 must be on: switch(0x08); This is the Straight path of length S2 on the switch.
e) In order to go from 15->13, switch Bit5 must be on: switch(0x10); This is the Curved path of length C2 on the switch.
f) In order to go from 15->0, don't switch anything.

Hints:

- Allocate semaphores/mutex to groups of segments so that they may be reserved (locked) by trains, but the following things should be kept in mind:
  1. You want to have an efficient solution
  2. You want to avoid deadlock situations
- The train must reserve the path ahead, and if it can't either slow down or come to a halt until the path ahead becomes available again.
- The thread controlling a train should never block while the train it controls remains in motion.
- The trains should move as fast as possible, but make sure that you always take into consideration:

1. Stopping distance.
2. Possibility of derailment if they run too fast in turns.

- Taking measurements and using them (i.e. for slowing down, speeding up, determining stopping distance) is STRONGLY encouraged.

Checkpoints: Your performance in this lab will be evaluated based on the following operations properly being implemented.

- Switch-transition to steer the train in the right direction.
- A smooth slow down of the train when needed.
- Ensuring that one of the trains is not stopped at all time.
- Proper use of semaphores.
- No deadlock or collision anywhere on the track.
- Decision procedure to manoeuvre the train properly.

9 EXPERIMENT# 4: Manoeuvre of Two Trains with Lights

Objectives:

In the previous experiment, the software interface developed can control the movement of two trains, which are running in the opposite direction to each other. There are no collision and/or deadlock by steering the switches. In this lab, besides using switches, you will be using lights to add another level of complexity in the train controller.
You will be making an extensive use of all software components being developed in earlier lab experiments. In particular, your position tracker and semaphore classes will be quite helpful.

Track Diagram:

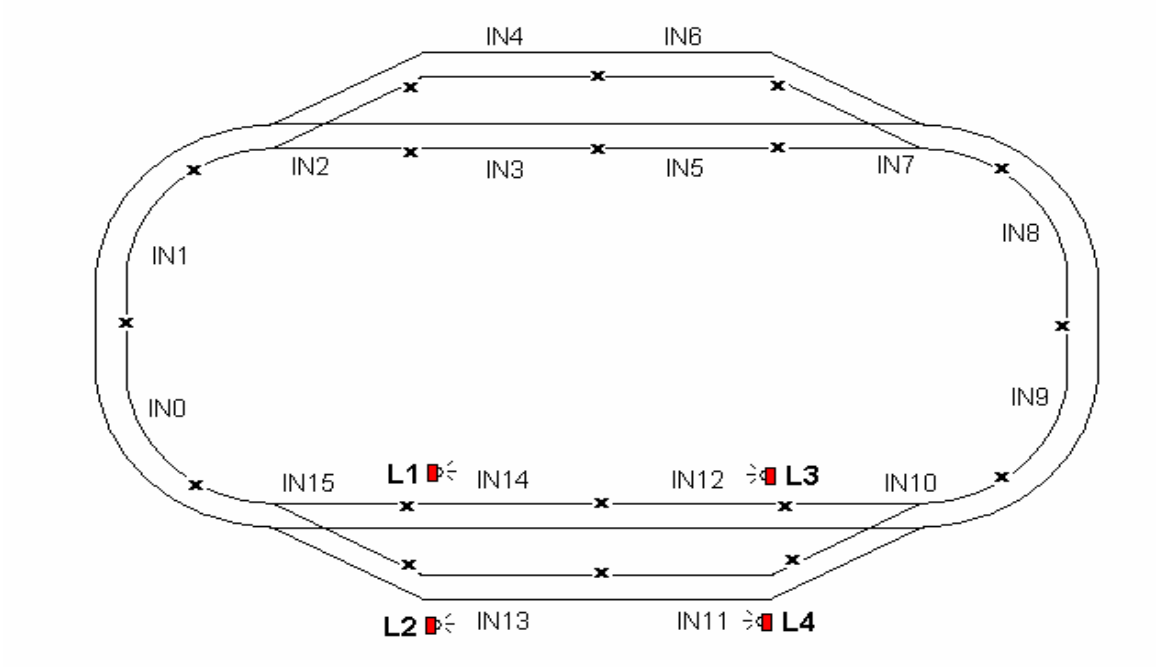


Figure 28: Track layout for experiment #4

Note: In this lab, both of the trains and lights should be used to control the trains.

Suggested Class Hierarchy Diagram:

The overall class hierarchy diagram for this experiment is the same as the previous one and would look like as follows:

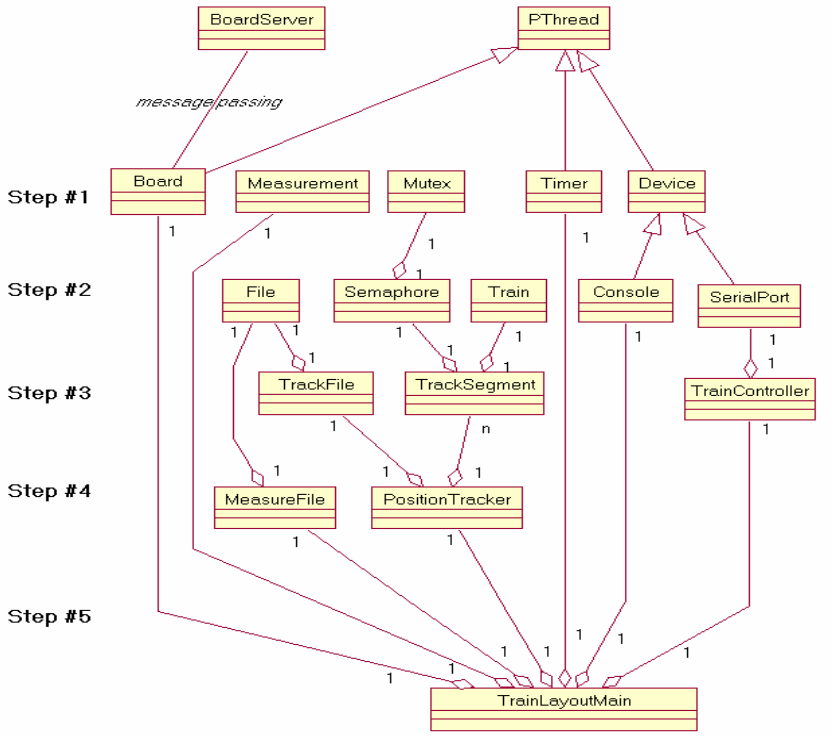


Figure 29: Overall class hierarchy diagram for experiment #4

Lab Description

Traffic lights: The circuit adds another level of complexity in the train control for this experiment and the next one. In addition to controlling the locomotives with respect to each other's position on the layout, you should also take into account an external input that would give the locomotive the right of way or not.

It consists of 4 DPDT switches and 4 traffic light units. Each traffic light unit has 4 LEDs, two green ones and two red ones. The upper two LEDs, a green one and a red one, are connected directly to one port of the DPDT through LED drivers. At the same time, the status of that port can be read from bit 0~bit 3 of a second IN-16 interface board, which has address 0x18C. In other words, you can manually turn on/off the upper two red/green LEDs and SBC can detect the change by reading the IN16 board from address 0x18C.

The lower two LEDs of the traffic light unit are connected to two bits from bit15 to bit 8 of the OUT16 board through LED drivers. The SBC board can automatically turn on/off the lower two LEDs.

Your control program should get the status of the traffic lights and decide if there is need to stop the locomotive(s) and set any of the LEDs belonging to the lower part of each traffic light unit.

Depending on the bit value the locomotive would have to act according to the table below.

Bit 0/1/2/3 read back from IN16	**Signal light unit**		**Locomotive**
	Red	**Green**	**Run/Stop**
1	On	Off	Stop

0	Off	On	Run

On certain tracks, locomotives should respect the traffic lights as shown below:

Tracks	Directions of Locomotives	Lights to be respected	Bits No. on In16
14	Clock-wise	L1	Bit 0
13	Clock-wise	L2	Bit 3
12	Counter-clock-wise	L3	Bit 2
11	Counter-clock-wise	L4	Bit 1

To test the hardware system, you can type:
/public/testio.x
You can see that the SBC controlled LEDs(lower two LEDs) are turned on/off one by one automatically. And if your second IN16 board is okay, you could find the content display for IN16 0x18C change when you change the position of the switch on the traffic light control board, and at the same time, the upper two LEDs will be turned on/off.

Suggested Design:

There are two sets of traffic lights, one set is controlled by SBC's OUT16, and the other set can be switched by hand and can be read back. We can define the two sets with different meanings. For the manually controlled lights, red light means: you better choose other way(s) if possible, while if you can not find other way(s), please enter the track segment and stop in front of me. For the SBC controlled lights, red light means: don't enter, this track is blocked because another train is here.

Checkpoints: Your performance in this lab will be evaluated based on the following operations properly being implemented.

- The locomotives should not pass any red lights. You can stop them by manually turning on the red light(s) when they go though the train station area.
- The SBC controlled traffic lights must properly indicate the track occupation.

10 EXPERIMENT# 5: Manoeuvre of Three Trains

Objectives:

In this experiment we will be controlling three locomotives instead of two in such a way that two of them travel in one direction and the third one in the opposite direction. It is important to ensure that two locomotives are always in movement.

Track Diagram:

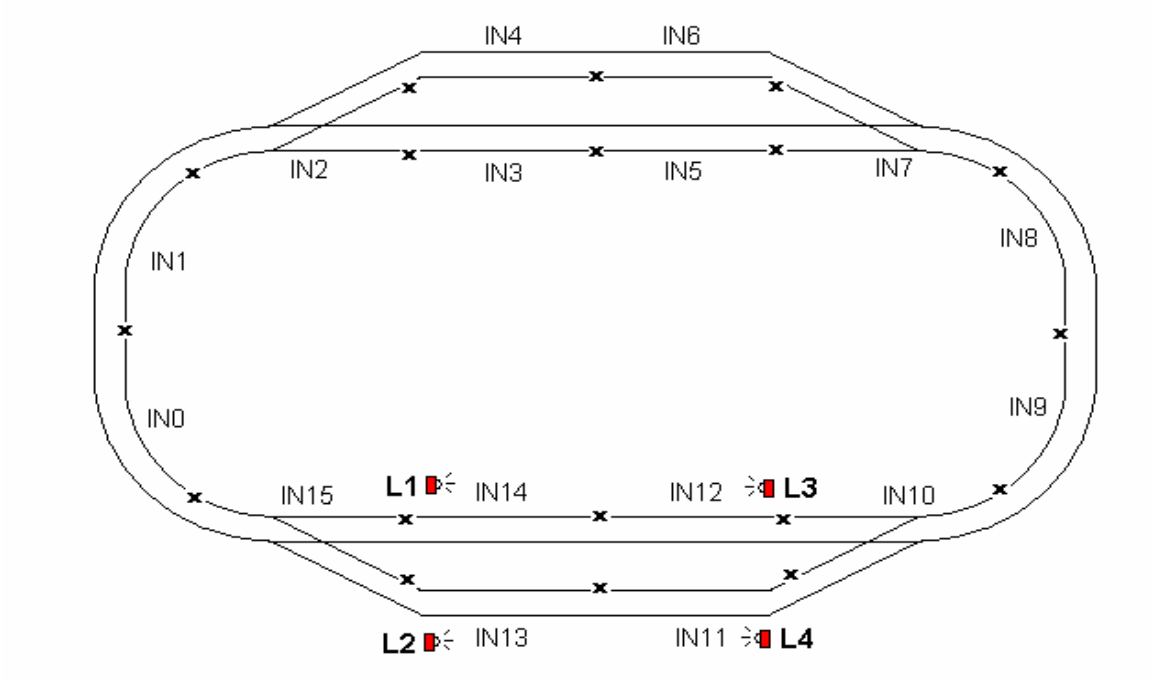


Figure 30: Track layout for experiment #5

Suggested Design:

The description and design are the same as the previous experiment except that there is going to be an extra train in this experiment. The same file oval.trk could be used in this case too and follow the instructions given below:

- To make sure that no collision will ever happen with 3 or more trains, and to insure that all the conditions stated are followed concisely, you must implement Semaphore in the following way:
- Each group of linear track (11 and 13), (12 and 14) must be reserved following a flow direction constraint. Each track can be reserved separately but most follow the same direction. For instance, if train 7 is on the track segment 11 wants to go right (CCW), then a train 3 can only reserve track segment 13, if it also wants to go right (CCW).
- No train can stop on the “outside” curves (0, 1, 8, 9) or on the switch track (15, 2, 7, 10), they can only slow down if needed to a reasonable speed which is not visually “stop”.

Suggested Class Hierarchy Diagram:

The overall class hierarchy diagram for this experiment would look like as follows:

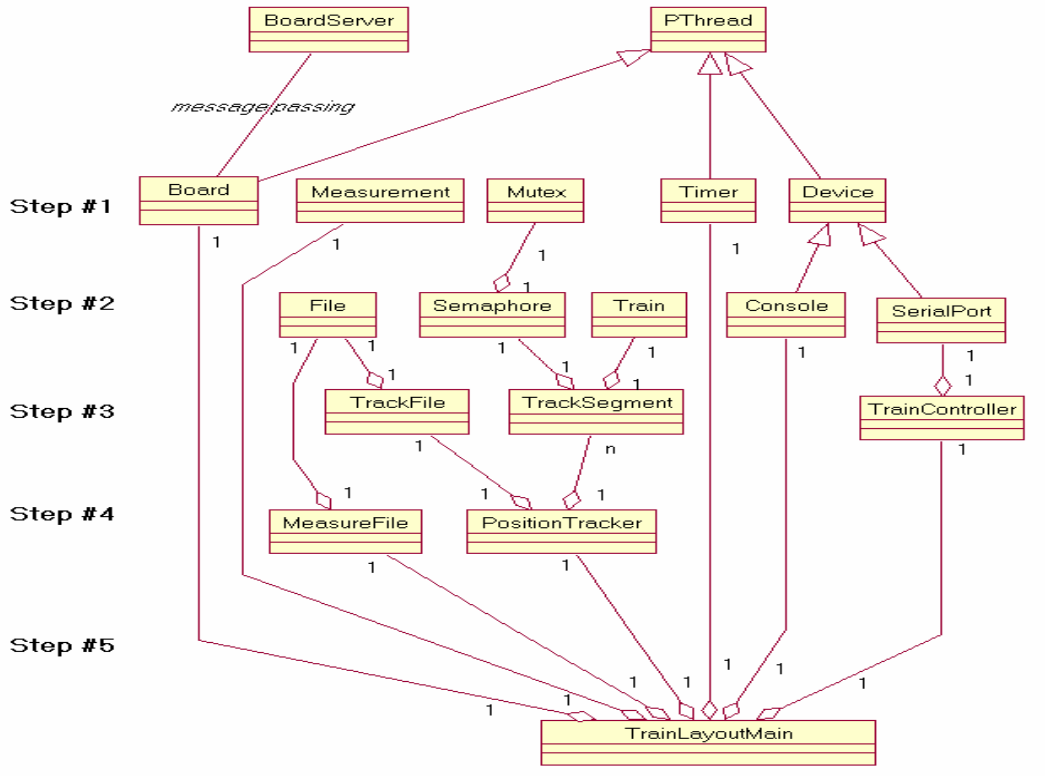


Figure 31: Overall class hierarchy diagram for experiment #5

Checkpoints: Your performance in this lab will be evaluated based on the following operations properly being implemented.

- Switch-transition to steer the train in the right direction.
- A smooth slow down of the train when needed.
- Ensuring that one of the trains is not stopped at all time.
- Proper use of semaphores.
- No deadlock or collision anywhere on the track.
- Decision procedure to manoeuvre the train properly.
- Only one train can be stopped at any given time.
- Train behaviour should be somewhat random and not statically pre-established or hard coded.
- Trains should respect the traffic lights rules, which are same as in experiment #4.