UNIVERSITY OF CALIFORNIA

SANTA CRUZ

BTS 2: A ROBUST, LOW-COST, REAL-TIME BUS TRACKING SYSTEM

A project submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Kerry Veenstra

March 2017

The Project of Kerry Veenstra
is approved:

Professor Katia Obraczka, Chair

Professor Roberto Manduchi

Table of Contents

List of Figures viii

List of Tables x

Abstract xi

Acknowledgments xii

1 Introduction **1**

1.1 Related Work 3

1.1.1 Example Commercial Systems 3

NextBus 4

TransLoc 5

1.1.2 Characteristics of Academic Systems 5

Adding Testbeds 5

Partnering to Share Costs 6

1.2 BTS 2 Compared to BTS 7

1.2.1 Updated Design Goals 7

Automatic Route Identification 7

Vehicle Arrival-Time Predictions 9

Additional Communications-Error Detection 9

Evaluation of Sensors for Improving Localization Accuracy 10

Improved Hardware Reliability 12
1.2.2 BTS 2 System Components 14
Front Route Sign 14
Route Selector 17
Route Programmer 18
Base Station 18
Database Server 19
Web Server 20

2 Design of the Bus Node **21**

2.1 Front Route Sign 21
2.1.1 Enclosure 21
2.1.2 Fiberglass Mounting Panels 22
2.1.3 LED PCBs 23
Design of the LED PCB 24
Layout of the LED PCB 24
Power Routing 28
Computer-Aided Design 29
2.1.4 Control PCB 29
Design of the Control PCB 29
Power Supply 29
Microcontroller 32
General-Purpose I/O 33
Connectors 35
LED-PCB Connectors 35
GPS-Receiver Connector 36

900-MHz–Radio Connector 37
JTAG Connector 37
Route-Selector Connector 37
Side-Sign Connector 38
Testbed-Processor Connector 39
Sensors 39
EEPROM 41
Layout of the Control PCB 41
Computer-Aided Design 43
2.1.5 BTS 2 Front-Sign Firmware 43
Requirements 43
BTS 2 Real-Time Task Management 45
2.1.6 Wiring Harness 47
2.2 Route Selector 47
2.2.1 Enclosure 47
2.2.2 Route-Selector PCB 48
Design of the Route-Selector PCB 48
Power Supply 48
Microcontroller 48
General-Purpose I/O 48
Connectors 50
JTAG Connector 50
Route-Sign Connector 50
Route-Programmer Connector 51
EEPROM 51
Layout of the Route-Selector PCB 51

Computer-Aided Design 53
Firmware of the Route-Selector PCB 53
2.2.3 Route-Selector Cable 54
2.3 Route Programmer 54
2.3.1 Enclosure 55
2.3.2 Route-Programmer PCB 55
Firmware of Route-Programmer PCB 55
2.3.3 Route-Programmer Cable 55
2.3.4 Wall Power Adapter 55

3 Base Stations and Servers **57**
3.1 Processing on the Base Station 59
3.1.1 Evaluation of the Original BTS 59
3.1.2 Improvements for BTS 2 60
3.2 Servers 61
3.2.1 Evaluation of the Original BTS 61
3.2.2 BTS 2 Server 61

4 Testbed Management **63**
4.1 Test-run Management 65
4.1.1 nodels 65
4.1.2 nodediff 66
4.1.3 noderun 66
4.1.4 nodecron 67
4.2 Data Collection 68
4.2.1 nodegs 68
4.3 Implementation 68

4.3.1 Client-Server Interaction 68
4.3.2 Client-Server Communication Protocols 69

5 Results **71**

5.1 Comparing SCORPION and DOME 71
5.2 PCB Design 73
5.3 Base-Station Design 74
5.4 Server Design 74

A Design Computations **75**

A.1 MAX6495 75
A.2 LED-Sign Power Routing 79
A.3 Ribbon-cable Termination Resistors 82

Supplemental Files **84**

Bibliography **85**

List of Figures

1.1 GPS Repeatability 11

1.2 Number of available BTS nodes vs. time 13

1.3 BTS 2 System Block diagram 15

1.4 LED PCB 16

1.5 Sign controller 16

1.6 Testbed CPU board 17

1.7 Route selector 18

1.8 Base station 19

2.1 Block diagram for bus 22

2.2 New Front Route Sign: rear view, exploded diagram 23

2.3 LED PCB Connections 24

2.4 Pinout of LED-PCB ribbon-cable connector 25

2.5 One Column of the LED PCB Logic 26

2.6 Control PCB Connections 30

2.7 Block diagram of power supply 31

2.8 Pinout of Control PCB's microcontroller 33

2.9 Ports PA and PB of the Sign Controller 34

2.10 Pinout of ribbon-cable connectors J9 and J10 35

2.11 Pinout of connector to GPS receiver 36
2.12 GPS cable 36
2.13 Pinout of Control-PCB connector for 900-MHz radio 37
2.14 Pinout of connector to Route Selector 38
2.15 Pinout of connector to Side Sign 38
2.16 Pinout of connector to testbed processor 39
2.17 Control PCB 44
2.18 Pinout of Route Selector's Microcontroller 49
2.19 GPIO of the Route Selector 49
2.20 Rotary-Encoder Timing Diagram 49
2.21 Pinout of connector to Route Sign 50
2.22 Pinout of connector to Route Programmer 51
2.23 Route selector PCB 53
2.24 Route Programmer Firmware-Development Prototype 56

4.1 Client-server model used by SCORPION 69

A.1 MAX 6495 Design Worksheet 78
A.2 Effect of termination on ribbon-cable signal integrity 83
A.3 Termination for ribbon cables 83

List of Tables

1.1 BTS 2 goals 8

2.1 Power Supply Requirements 30

2.2 Port PA and PB Bit Functions 34

2.3 Accelerometer connections 40

2.4 Controller Task Comparison 43

2.5 GPIO bit assignments of Route-Selector PCB 50

A.1 Explanation of Selected MAX6495 Design Values 77

Abstract

BTS 2: A ROBUST, LOW-COST, REAL-TIME BUS TRACKING SYSTEM

by

Kerry Veenstra

This report describes the goals, design, and deployment of the prototype for the BTS 2 bus-tracking system. It notes where the BTS 2 production system differs from the BTS 2 prototype and where it differs from the original BTS system.

The author created the bus nodes and base stations of the BTS 2 prototype and managed the creation of the production bus nodes. Tracking data from the BTS 2 production system now is available to UCSC campus users through web and smartphone apps written by others.

Acknowledgements

Portions of section 1.1.2 and Chapter 4 appear in an expanded form in unpublished conference-paper submissions by the author. James Koshimoto and Matt Bromage created the original BTS system which inspired this project. Undergraduate researcher Ben Cizdziel wrote the initial version of the firmware for the BTS 2 Route Selector under my direction. University professors at UCSC assisted me in learning the syllabuses of courses that were used in this theses. Finally, I'd like to acknowledge an unexpected source: lynda.com, whose videos taught me to use Adobe Illustrator's time-saving *Live Trace* button, which gives my hand-drawn pen sketches a pseudo-professional facade.

Chapter 1

Introduction

Imagine a cold and wet night on campus. A young student stands at a bus stop, quite aware that there is warmth and safety in the building behind her. She is unsure when the bus will arrive, and she knows that when it does the driver must see her waiting at the road or else he may drive by. She would like to wait indoors *and* be certain of catching the bus, but she cannot do both without real-time knowledge of the bus's position.

We see this scenario every year, as students wait at the campus's bus-stops. However, the 2009–2010 school year at UCSC was different: on a web-browser-based map of the campus, students could see markers representing campus shuttles moving in real-time. The system that generated those maps was BTS[16].

Although the public face of the BTS system was its real-time transit map, its creators had other motivations: the nodes of the BTS system served as part of the SCORPION testbed for research in wireless networking[7]. The public service provided by the BTS nodes was a beneficial side effect made possible by erecting five base stations around UCSC and mounting small Linux boxes on the frames of about 20 transit shuttles.

BTS 2: The original BTS system met its initial goals, but unfortunately it had shortcomings that prevented long-term deployment. This project report lists those problems and describes the design and implementation of a new bus-tracking system, BTS 2, that addresses the problems and provides more reliable transit tracking and testbed features at the UCSC campus.

I have organized this report into five chapters. Chapter 1 introduces the subject of the report, discusses related work, and states the goals of BTS 2 and how they differ from those of BTS. Chapter 2 presents the design and implementation of the new BTS 2 bus nodes. These nodes are entirely original and are the heart the project's contribution. Chapter 3 presents the design of the five campus base stations that receive data from the bus nodes and the design of the server that stores their data. Chapter 4 focuses on using the nodes to make a MANET testbed. Since testbed-management requirements are unchanged from BTS, Chapter 4 merely documents the testbed-management programs that are shared by both BTS and BTS 2. Finally, Chapter 5 reports results.

I created all of the BTS 2 prototype myself, but some parts of the prototype are modifications of original BTS designs by others. Here are my specific contributions:

- All hardware design and construction (route sign, route selector, route programmer, base station)
- Firmware design and implementation (route sign)
- Firmware *redesign* and implementation (route selector—original firmware by Ben Cizdziel)
- Software design and implementation (arrival-prediction)

- Software redesign and implementation (base stations and backend-server prototype—original BTS software by James Koshimoto and Matt Bromage)
- Database design
- Web app design and implementation

In addition to creating the BTS 2 prototype, I managed the deployment of the BTS 2 production system. The production system is almost identical to the prototype, but it integrates these new features which were made by others:

- New web app (Kevin Abas)
- Android app and new back-end server (Wade "Simba" Khadder)
- iOS app (Sterling Dreyer)
- Firmware for the route programmer—in progress (Christopher Villalpando)

1.1 Related Work

BTS 2 provides features that are similar to those of other tracking systems. Below I describe other academic systems and some commercial systems that provide tracking but lack testbeds. Since there are differences between commercial systems and academic systems, I discuss these two kinds separately. I describe the original BTS in section 1.2, where BTS 2 is defined.

1.1.1 Example Commercial Systems

Two well-known commercial systems are NextBus and TransLoc.

NextBus

NextBus tracks buses of a transit system using GPS receivers. The system determines bus locations and predicted arrival times and displays them on web sites, on mobile-phone apps, and on digital signs at bus stops[38]. To help riders decide which of the approaching buses to take, some NextBus installations include on each bus a Passenger Load Sensor System (PLSS) which estimates the number of passengers on the bus[28]. NextBus applications receive data updates at a rate that depends on the installation: from as often as every 10 seconds to as infrequent as every 2 minutes. The NextBus installations that have faster data-update rates can support real-time location maps.

A full-featured NextBus system is expensive to install and maintain. For example, the Washington Metropolitan Area Transit Authority (WMATA) reports that installing NextBus on the Metrobus in the Washington, D.C. area added $3,000,000 in capital investments. In addition, WMATA spends $223,000 annually in NextBus operating costs[37]. Since Metrobus has a fleet of 1,480 buses[42], a rough per-bus cost estimate for NextBus is $2,000 per bus for installation and $150 per year for maintenance. (This estimate amortizes the costs of any purchased server equipment over the costs of the buses.) Reference [37] does not state whether all of the buses had route signs at the time of system installation, but likely they already did. WMATA bus stops do not have electronic message signs, and so the capital investments did not include such signs.

Checking the NextBus web site, one finds that most NextBus customers run entire metropolitan transit systems, like that of WMATA.

TransLoc

Another commercial bus-tracking system, TransLoc, provides basic features of bus tracking and arrival prediction[35].

TransLoc installations appear to be less expensive and less extensive than those of NextBus. A TransLoc installation at Louisiana State University (LSU) cost $20,000 for GPS equipment and was installed in a day[36]. Since the LSU Tiger Trails system has 23 buses (18 active plus 5 spares[41]), the per-bus installation cost was about $870. The annual maintenance cost was not reported.

Checking the TransLoc web site, one finds that most TransLoc customers run small, campus-based transit systems, like that of LSU.

1.1.2 Characteristics of Academic Systems

The primary difference between commercial systems and academic systems is that academic systems sometimes limit tracking features but compensate by supporting research testbeds.

Adding Testbeds

Network researchers can evaluate their algorithms on simulators, testbeds, and complete deployments. Simulators, such as QualNet[26], ns-2[23], and ns-3[24], test algorithms in controlled artificial environments. Using a simulator, a researcher can test specific circumstances with manually or algorithmically generated stimuli, can improve test coverage with pseudorandom stimuli, or can model operation under real-world conditions with network traces. Also a researcher can repeat prior stimuli in subsequent simulator runs (to compare different algorithms) and can design stimuli to include all system states. Simulators are best for making

detailed observations and for subjecting the system under study to a wide variety of conditions, but they have weaknesses.

While simulators can model real-world environments through network traces, the use of traces presumes that a network's responses will be unaffected by the modeled algorithm. Instead of using a simulator, researchers who want to measure the effects of algorithms on network activity can deploy their algorithms to testbeds and drive them with real-time stimuli, as in GENI[2], Planetlab[25], and DOME[31]. Such testbeds may sacrifice repeatability and may limit observability, but they can show the effects of real-world activity that may not have been revealed by artificially generated stimuli. Using testbeds is even more important in the case of *wireless* networks where the physical world plays a definitive role and is much harder to capture than using mathematical abstractions and models.

Another benefit of using testbeds is the possibility of *long-term* experimental runs[31]. Assuming that the nodes of the testbed are deployed reliably and are available for use, a researcher can design an experiment that runs for weeks or months. Observations collected during such longitudinal studies can help reveal long-term trends.

Partnering to Share Costs

Researchers track the mobile nodes of testbeds because node mobility affects experimental results, however once one *has* tracking data, it can be used for other purposes that may allow sharing the costs of system installation and maintenance. Mobile wireless testbeds need vehicles for node mobility, and of course vehicles cost money. Researchers can reduce these costs by partnering with transit authorities to deploy nodes on existing vehicles, providing vehicle-tracking features to transit passengers and transit managers as motivation to provide access to ve-

hicles. Based on my own experience, which is consistent with that of DOME[31], I believe that such a partnership is more likely to be successful when researchers and transportation authorities find the resulting system jointly beneficial.

1.2 BTS 2 Compared to BTS

During the summer of 2009, UCSC's original Bus Tracking System and Wireless Networking Testbed was successfully deployed to about 20 campus transit shuttles. Over the 2009/2010 school year the system was evaluated, and several improvements were identified. The following summer, when most of the campus transit shuttles were retired and replaced with larger, more fuel-efficient vehicles, the BTS hardware was removed from the retiring vehicles and stored.

Based on the BTS evaluation, and using feedback from that system's users (TAPS employees, wireless-networking researchers, and shuttle riders), I have designed an improved BTS 2.

1.2.1 Updated Design Goals

My overall goals for BTS 2 remain the same as those for BTS: to create a combination transit-vehicle tracking system and ad hoc wireless-network testbed. However, there are several specific improvements. I summarize these in Table 1.1 and detail them below.

Automatic Route Identification

Although the real-time location map of the BTS system identified and located each of UCSC's transit shuttles accurately, the route indicators displayed on the map were incorrect. For example, regardless of the route that a vehicle followed,

its indicator on the map was set permanently to "Loop" or "Night Core" or any one of the other published route names. The cause of this problem was a user-interface shortcoming rather than a programming error. The system as originally designed required vehicle route assignments to be entered manually by a TAPS shift supervisor using a web-browser-based GUI. Unfortunately, vehicles changed routes frequently enough to make updating the GUI more of a hassle than route-indicator accuracy was worth, and so TAPS stopped using this feature of the system. For passengers, the lack of accurate route indicators reduced the map's value.

To address this problem with BTS 2, I note that each transit-shuttle driver is responsible for setting the message on the shuttle's route sign correctly. The BTS 2 bus node reads this setting from the route sign to identify vehicle-route assignments automatically. This setting is sent, along with the shuttle's GPS location, from the BTS 2 bus node to BTS 2 base stations for storage in the tracking database.

	Project	
Common Goals	**BTS**	**BTS 2**
Real-time bus-location maps	●	●
Testbed for ad hoc network research	●	●
New Goals		
Automatic route identification	○	●
Vehicle arrival-time predictions	○	●
Additional communications-error detection	○	●
Improved hardware reliability	○	●

Table 1.1: BST 2 goals.

Vehicle Arrival-Time Predictions

Imagine standing alone at a shuttle stop at midday, when all of a sudden students pour out of dorms, cell phones in-hand, just as the shuttle approaches. This scenario is not fiction. It happens every weekday during the school term at UC San Diego, and it can happen at UCSC, too.

Although the real-time map on the BTS web site provided accurate geographic locations of each shuttle, the map's use as a forecasting tool was limited to those riders who already were familiar with the routes and speeds of shuttles. Consequently, the BTS 2 web-site adds predictions of shuttle arrival times[1]. In addition, the web server provides the prediction data in an XML file that can be displayed by mobile apps. While this project does not provide a mobile app, the predictions can be used by students who want to make their own mobile app.

Additional Communications-Error Detection

BTS used Aerocomm CL4490 data radios, which support packet-error detection and packet retransmission[1]. The support of error detection and retransmission suggests that tracking data would be transferred from shuttles to base stations error-free, but paradoxically the BTS database shows occasional errors, such as records with an invalid shuttle ID of -122 (the integer portion of longitudes on the UCSC campus!). Analysis of the BTS system design reveals the likely cause of these errors.

BTS bus nodes created variable-length data frames, but since the CL4490 ra-

[1]This report describes the BTS 2 prototype. The prototype used the INRG lab's "skynet" server for database, web-server, and arrival-prediction services. The production BTS 2 system migrates the BTS 2 database and web-server functions to a VM that is managed by UCSC IT. The arrival-prediction functionality provided by "skynet" has not yet been migrated, and since "skynet" suffered a recent hardware failure, arrival-prediction services are not currently running.

dios send fixed-length packets, at least occasionally a BTS frame was fragmented across a pair of radio packets. Although the radio's error detection and retransmission of individual fragments should have guaranteed correct transmission of complete frames, a *mobile* bus node could leave a base station's coverage area *between* fragment transmissions. Such a situation (the loss of the second fragment of a BTS frame) causes a specific problem.

The CL4490 does not reassemble packet fragments into a single packet before sending the packet to the base station. Instead, it sends each fragment to the base station as it receives it. Also, the radio attempts only four retransmissions of each fragment, and so if all of the second fragment's transmissions are lost, the base station's input buffer will contain just the first packet fragment. This fragment will act as a "prefix" to the next full frame that is received, creating a corrupt frame in the base station's input buffer. The original BTS base-station code lacks error detection, and so the situation outlined above leads to invalid records being written to the database.

While the scenario mentioned above explains the record corruption that is seen, I cannot directly verify that there are occasional corrupt frames in base-station buffers because nodes of the BTS system no longer are deployed. Nonetheless, this explanation is the best that I have, and so while BTS 2 continues to use CL4490 radios, it also adds a frame checksum so that invalid frames can be detected and discarded. (Invalid frames are discarded rather than retransmitted because fresh BTS 2 frames are transmitted every three seconds.)

Evaluation of Sensors for Improving Localization Accuracy

BTS nodes use GPS receivers to identify each bus's position (longitude and latitude). Although GPS receivers have legendary accuracy, archive BTS data show

that bus positions reported along Heller Dr. between Porter College and the Core West parking structure often do not lie along the actual location of the road (see Figure 1.1). Instead the GPS data from this area sometimes shows offset errors more than 50 meters. These errors can confuse BTS map users who see a bus's icon traveling through the forest!

The cause of such offset errors likely is the forest canopy of the northern devel-

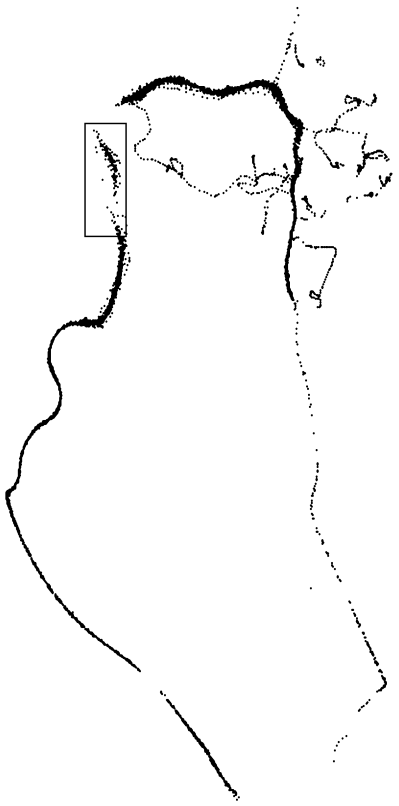


Figure 1.1: GPS repeatability demonstrated by BTS data that was collected at UCSC on August 9, 2010. The rectangle in the upper-left corner marks a region that often shows large GPS offset errors along Heller Dr. between Porter College and the Core West parking structure. While Section 2.1.5 discusses an additional cause of GPS error, the map clearly shows a lack of location repeatability in forested areas of the campus.

oped portion of the UCSC campus. While the U.S. Department of Defense reports that civilian GPS receivers provide 3-meter accuracy 95% of the time, such accuracy requires reception of 19-cm–wavelength signals from four GPS satellites simultaneously[13]. GPS-receiver manufacturer Trimble Navigation explains that materials with high water content—such as leaves of trees—become signal attenuators, and that consequently, "Forest canopy is one of the most limiting factors in using the global positioning system for positioning and mapping"[22].

To help improve the accuracy of positioning data, the BTS 2 project adds two sensors, an accelerometer and a magnetometer (magnetic compass). The data from these sensors are stored in the database to improve localization in the future; this project does not use the data to augment position data.[2]

Improved Hardware Reliability

The hardware for each BTS node was based on a Mini-ITX motherboard with daughter cards for ad hoc wireless communication. Although such motherboards are found in amateur automotive-computing applications, deployed bus nodes showed more early failures than expected. Figure 1.2 shows the number of available BTS bus nodes over time.

Briefcase nodes of the SCORPION testbed use similar motherboards, and I have discovered that those nodes often crash when their electromechanical connections are flexed. Based on this experience, I am reluctant to redeploy the original BTS node hardware in BTS 2. Instead, bus nodes for BTS 2 adopt construction techniques more like those used in industrial and automotive electronics, elim-

[2]While the original prototype node has accelerometer and magnetometer features, evaluation of its archived data showed us that neither the accelerometer nor the magnetometer provide sufficiently precise data to disambiguate GPS offset errors. Consequently, the production nodes save cost by omitting these hardware modules. That said, each production bus node has sockets for these two sensors should they be wanted in the future.

inating electromechanical connections whenever possible. Consequently, BTS 2 nodes are constructed using custom printed-circuit boards (PCBs) and through-hole and gull-wing surface-mount device packages. The leads of such packages absorb the stresses that PCBs experience due to thermal and mechanical shocks thereby avoiding connection disturbances.

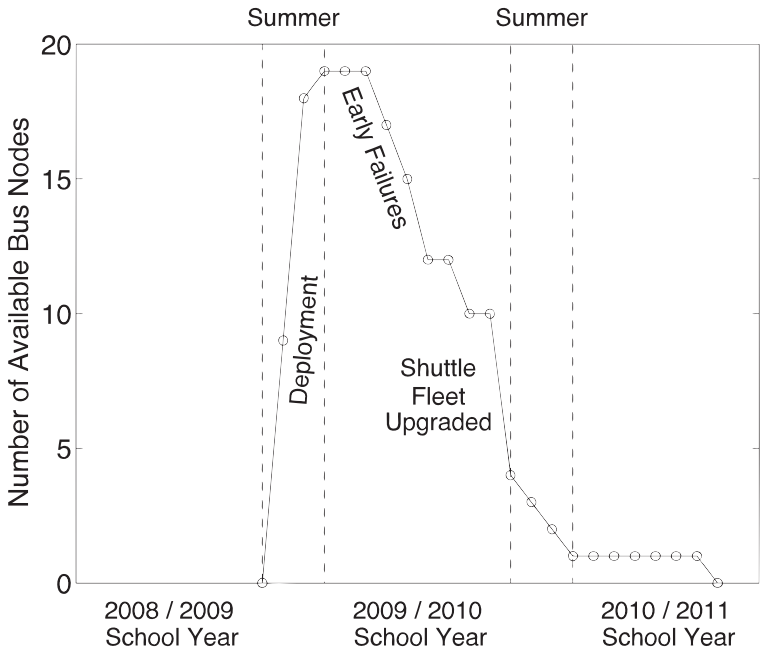


Figure 1.2: Number of available BTS bus nodes vs. time. Each individual node is continuously *available* from its first appearance in the tracking database through its last. All nodes were deployed during the summer of 2009. Most had been removed from retiring shuttles by the following summer.

1.2.2 BTS 2 System Components

These are the most significant differences betweens the BTS and BTS 2 implementations:

- The BTS 2 bus node adds a route sign and a route selector.
- The BTS bus node has a single CPU which performs all tasks, but tasks of the BTS 2 bus node are performed on three separate CPUs.

The rest of this section describes the BTS 2 system components.

As shown in Figure 1.3, each bus has a Front Route Sign and a Route Selector[3]. The Front Route Sign sends data wirelessly to nearby base stations. Base stations store any received data in the database. The Web Server retrieves information from the database and provides it to riders' web browsers and smart phones. At any time, TAPS can reprogram a bus's Route Selector with new messages by attaching a Route Programmer to it (not shown).

Front Route Sign

The front route sign contains the following electronics:

- 13×64 **LED message display.** (Figure 1.4) The buses' old route signs are 16×112 flip-disc displays. While it would have been possible to replace these signs with identically sized LED signs, I know that usually only the central regions of the signs are used. To reduce the cost of the new signs, I chose a smaller array. Although the smaller LED array displays shorter static messages, longer messages can be displayed through right-to-left scrolling.

[3]The optional Side Route Sign and optional Rear Route Sign are not part of this project, but a connector on the Front Route Sign allows daisy chaining data signals to these signs in the future.

- **Sign Controller** (Figure 1.5) with a 900-MHz Aerocomm wireless radio, a GPS receiver, an accelerometer (optional), and a digital magnetometer (optional). The microcontroller retrieves GPS-position coordinates and sensor data and sends it along with the bus's numerical ID, its route name, and a checksum to any listening Base Station.

- **Testbed CPU** (Figure 1.6, optional) with 2.4-GHz ad hoc wireless link. The testbed CPU can be programmed over the 2.4-GHz link.

The Front Route Sign receives power from the vehicle (and the Route Selector receives power from the Front Route Sign). The optional testbed CPU receives the

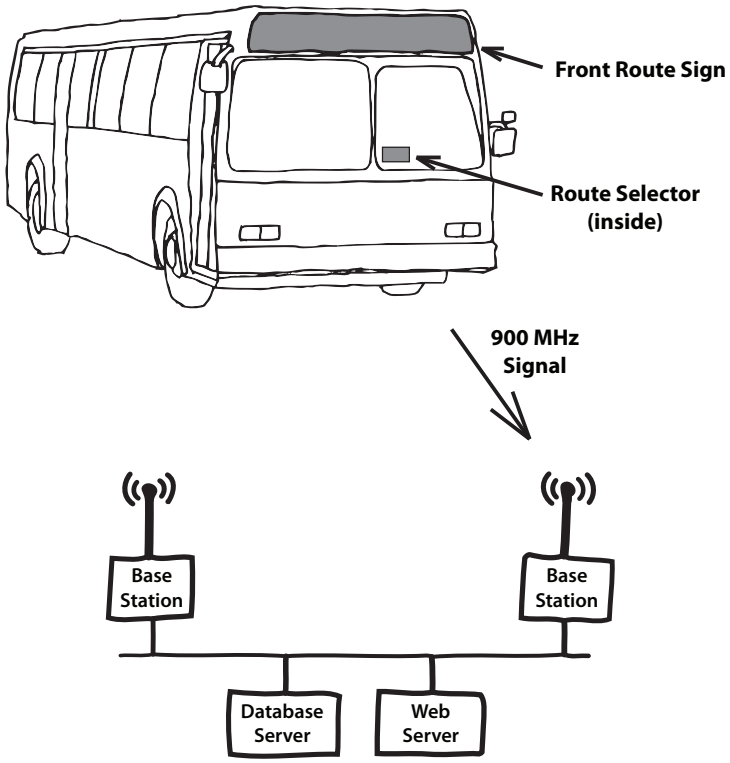


Figure 1.3: BTS 2 System Block Diagram. Only two of the system's five base stations are shown.

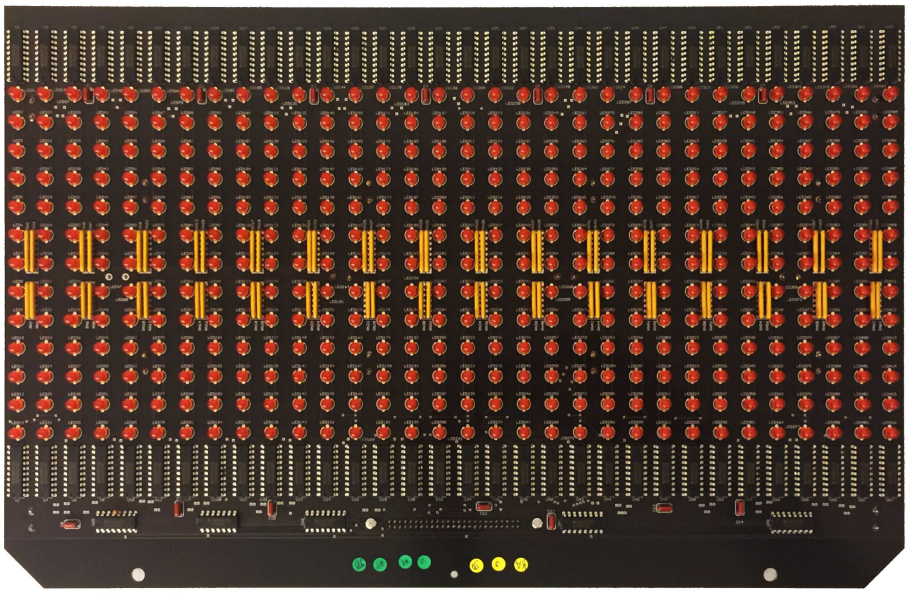


Figure 1.4: LED PCB. One of two identical PCBs in the LED Message Display. These are mounted in the sign enclosure with the LEDs facing forward.

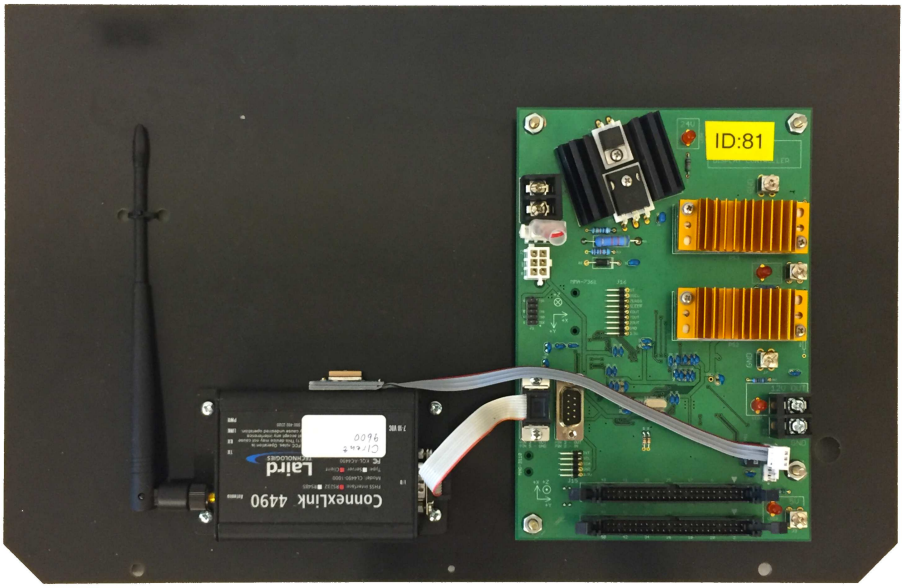


Figure 1.5: Sign controller. This board is mounted in the sign enclosure with its back facing forward so that the mounted components are not visible through the sign window.

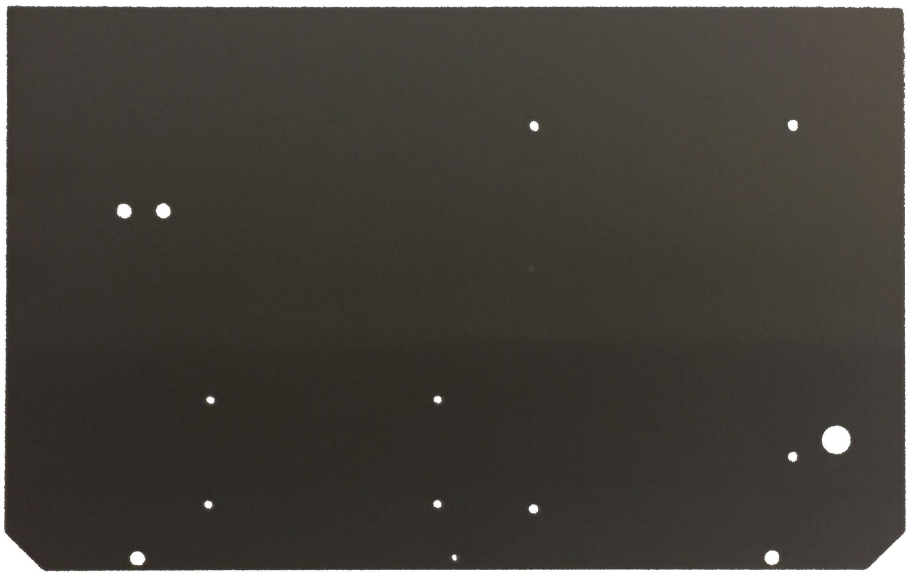


Figure 1.6: The Testbed CPU board of the route sign provides mounting holes for a future testbed CPU. (The Testbed CPU board is just an unpopulated Sign Controller board.

most recent GPS coordinates from the supervisor microcontroller and in exchange sends its status updates to the supervisor microcontroller.

Route Selector

The Route Selector (Figure 1.7) is mounted within reach of the seated bus driver. It has a mind-numbingly simple interface: twist a knob to change the route message. The selected message always is shown on the Route Selector's display and on the Front Route Sign. Using non-volatile memory the Route Selector remembers the message that was most recently displayed and will cause the Front Route Sign to redisplay that message after power is cycled.

Route Programmer

The Route Programmer is a sort of "master" Route Selector: TAPS fleet management can edit messages that are on the Route Programmer, and then at any point, TAPS can copy the new messages to any Route Selector by plugging the Route Programmer into the Route Selector using a short cable.

Base Station

BTS 2 Base Stations (Figure 1.8) use the same radios and roof antennas as the BTS base stations, but they refresh the compute hardware from hard-drive-based Mini-ITX computers to Raspberry Pis with solid-state memory cards. In addition, base-station software has been completely updated.

Figure 1.7: Route selector.

Database Server

The BTS 2 database contains a table that records the current location of each bus. Compared to the BTS database, this table adds columns for acceleration, compass heading, and for the bus's route name[4]. Other tables that define the bus routes and the expected inter-bus-stop delays were added to support arrival predictions.

This database is updated by the five on-campus base stations, and it is queried

[4]Columns for acceleration and the compass heading were used by the prototype, but since the production nodes omit the accelerometer and the magnetic compass, these two columns now are ignored.

Figure 1.8: Base station.

by a daemon that creates an XML file of shuttle locations[5].

Web Server

We can use the BTS Web Server nearly unchanged. In the prototype, arrival predictions are static, based on a simple table. The database archives all information so that a future project can compute predictions dynamically.[6]

[5]In the production version of BTS 2, a JSON file is created instead of an XML file.

[6]The arrival-prediction algorithms have not yet been migrated from the old "skynet" server, which suffered a hardware failure, to the new VM-based UCSC IT-maintained server.

Chapter 2

Design of the Bus Node

This chapter details the design of the vehicle portion of the BTS 2 system, in particular, the Front Route Sign, the Route Selector, and the Route Programmer (Figure 2.1). Each of these parts uses its own firmware-controlled microcontroller, and so in the sections that follow I discuss both the hardware design and the firmware organization.

2.1 Front Route Sign

The Front Route Sign is constructed using three custom printed-circuit boards (PCBs), two custom fiberglass panels, a 900-MHz radio, a GPS receiver, and an optional testbed processor. The sign reuses the aluminum enclosure from the bus's old route sign. See Figure 2.2.

2.1.1 Enclosure

(Figure 2.2, item 1.) Each of the TAPS buses already has a flip-disc[40] route sign. But since these signs are difficult to program and nearly unreadable at

night, TAPS plans to replace them. To control the project's cost, BTS 2 reuses the old signs' aluminum enclosures. In reusing the enclosures, the BTS 2 project replaces each of the old sign's four flip-disc modules with four replacement PCBs that have the same dimensions but improved or different functions. I describe the replacement PCBs next, in Sections 2.1.2 and 2.1.3.

2.1.2 Fiberglass Mounting Panels

The two outermost flip-disc modules of the old route sign are replaced with bare fiberglass panels (Figure 2.2, item 2). These panels have no circuit traces. Instead they support other parts of the sign. Viewed from the rear, the leftmost panel supports the Control PCB, the 900-MHz radio, and the GPS receiver. The rightmost panel supports the optional testbed processor. Pairs of these panels are fabricated by a PCB company by etching all of the copper foil from both

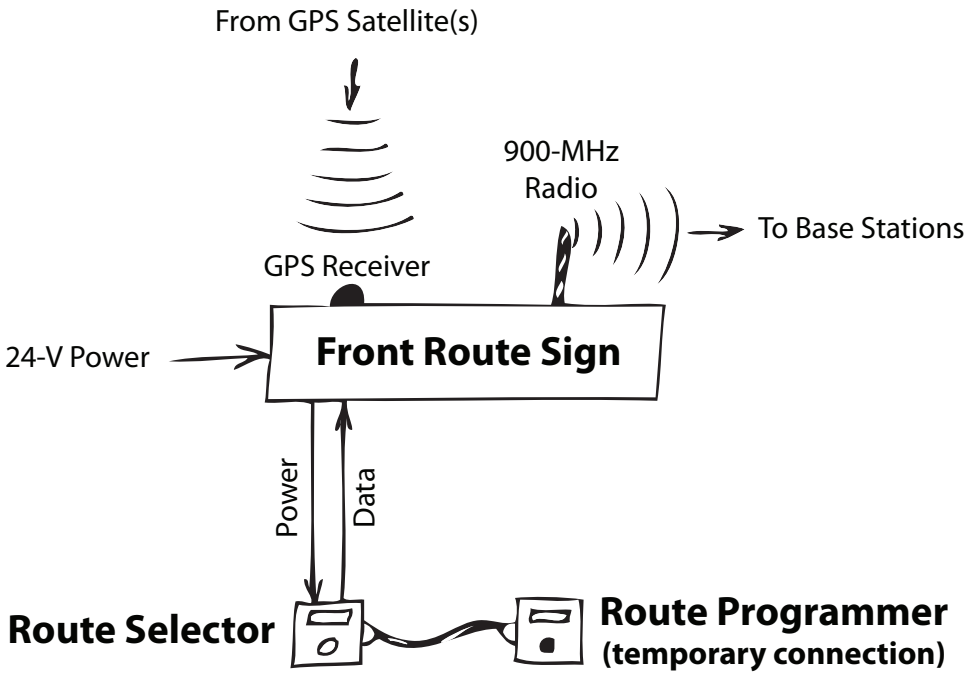


Figure 2.1: Block diagram for bus.

sides of a raw 18-in × 24-in × 0.093-in PCB panel and routing it into the two 16.8 in × 10.85 in bare fiberglass mounting panels. While it would have cost less to use unetched copper-clad fiberglass panels, leaving the copper in place would have attenuated the radio signals of the sign (900 MHz, GPS, and WiFi).

2.1.3 LED PCBs

The two innermost flip-disc modules of the old route sign are replaced with LED PCBs (Figure 2.2, item 5). These PCBs together form a 13 × 64 regular array of LEDs for displaying route names. This part of the sign is "dumb": it's merely a peripheral and lacks its own CPU. Another part, the Control PCB, provides the sign's intelligence. The Control PCB is described later in Section 2.1.4.

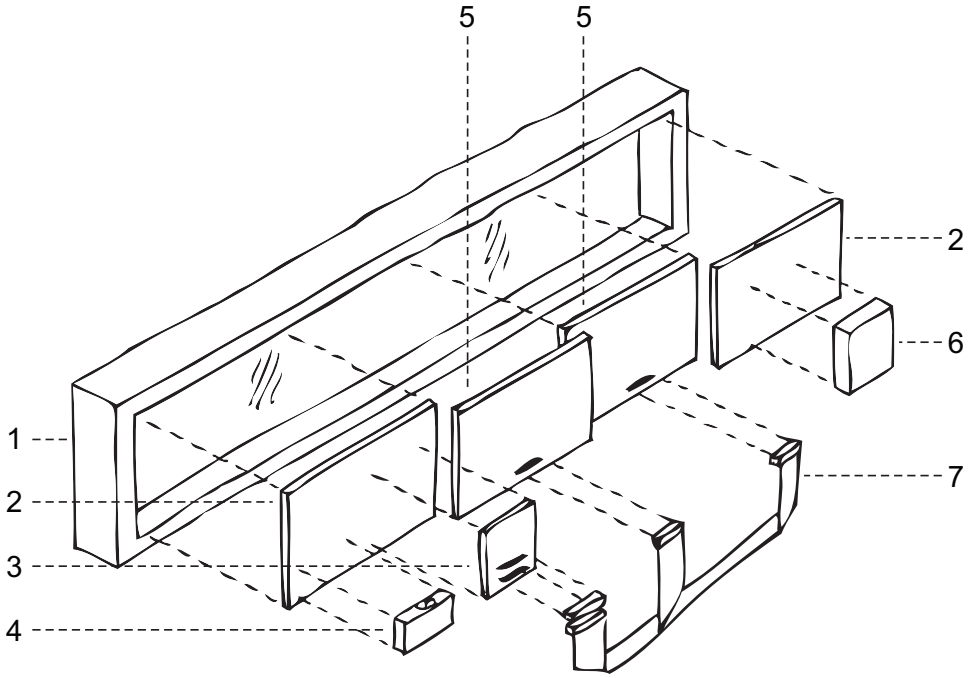


Figure 2.2: New Front Route Sign: rear view, exploded diagram. (1) Aluminum enclosure with transparent window. (2) Fiberglass mounting panel (2×). (3) Control PCB. (4) 900-MHz radio and GPS receiver. (5) LED PCB (2×). (6) Testbed Processor. (7) Wiring harness.

Design of the LED PCB

Each LED PCB holds 416 LEDs, organized into 13 rows and 32 columns. Aside from power and ground, the only connections to the LED PCB are through a 50-pin ribbon-cable connector.

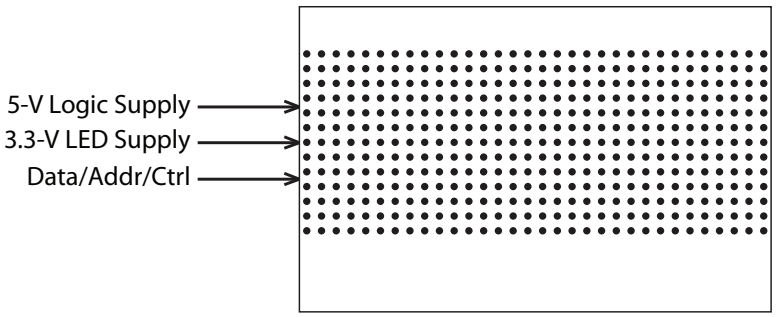


Figure 2.3: LED PCB connections.

The PCB's control interface is simple: change any column of LEDs on the board by providing the column's data and address and strobing a latch-enable signal. Figure 2.4 defines the pinout of the ribbon-cable connector, and Figure 2.5 describes the PCB's basic logic. (PDFs of the LED PCB's full logic schematic can be found in the Supplemental Files listed in the table on page 84.)

Layout of the LED PCB

The LED PCBs have an unusual layout. While most PCBs have only a few physical constraints, such as the overall dimensions and the locations of mounting holes and connectors, the LED PCBs fix the positions of almost all of the components because the LEDs must form a visible, regular matrix. These are the constraints that the LED PCBs must meet:

1. **Proper physical dimensions, including thickness.** To fit in the aluminum enclosure, the LED PCBs must have the same dimensions as the

PCBs of the flip-disc modules, which are 16.8 in × 10.85 in × 0.093 in.

2. **Maximum PCB size.** All PCBs are fabricated in batches by arraying them on standard-sized raw PCB panels. Comparing the sizes of panels and PCBs, it appears that two LED PCBs can be arrayed on a standard 18-in × 24-in raw panel, but there is further consideration. Panel-fabrication steps require leaving a 0.5-in border on a two-layer panel or a 1-in border on a multi-layer panel[6]. And assuming that fabrication uses a 0.1-in router bit, the two PCBs on the raw panel are separated by 0.1 in. These constraints limit the maximum size of a multi-layer LED PCB to 16.00 in × 10.95 in (because $1 + 16 + 1 = 18$ and $1 + 10.95 + 0.1 + 10.95 + 1 = 24$). For a two-layer LED PCB, the maximum size is 17.00 in × 11.45 in (because $0.5 + 17 + 0.5 = 18$ and $0.5 + 11.45 + 0.1 + 11.45 + 0.5 = 24$).

3. **Cost-conscious fabrication.** Since it has fewer fabrication steps, a two-layer PCB costs less than a multi-layer PCB of the same size[5, 6]. Consequently, as long as the dimensions of a PCB are fixed, using a two-layer PCB

Pins 9, 11, 17, 19, 21	`A[4:0]`	Column addr
Pins 7, 5, 13, 3, 15, 1, 23, 49, 25, 47, 27, 45, 35	`D[12:0]`	Column data
Pin 33	`ALE`	Latch enable
Pin 29	`OE`	Output enable
Pin 31	`OEN`	$\overline{\text{Output enable}}$
Pins 2, 4, 6, ..., 50	`GND`	Signal ground

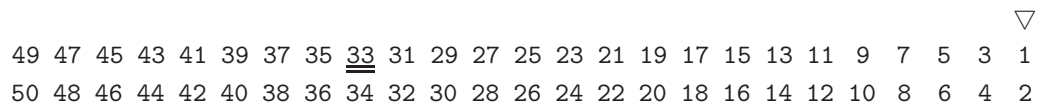


Figure 2.4: Pinout of the LED-PCB ribbon-cable connector. The ▽ symbol on the connector denotes pin 1. Every other conductor of the ribbon cable is connected to GND.

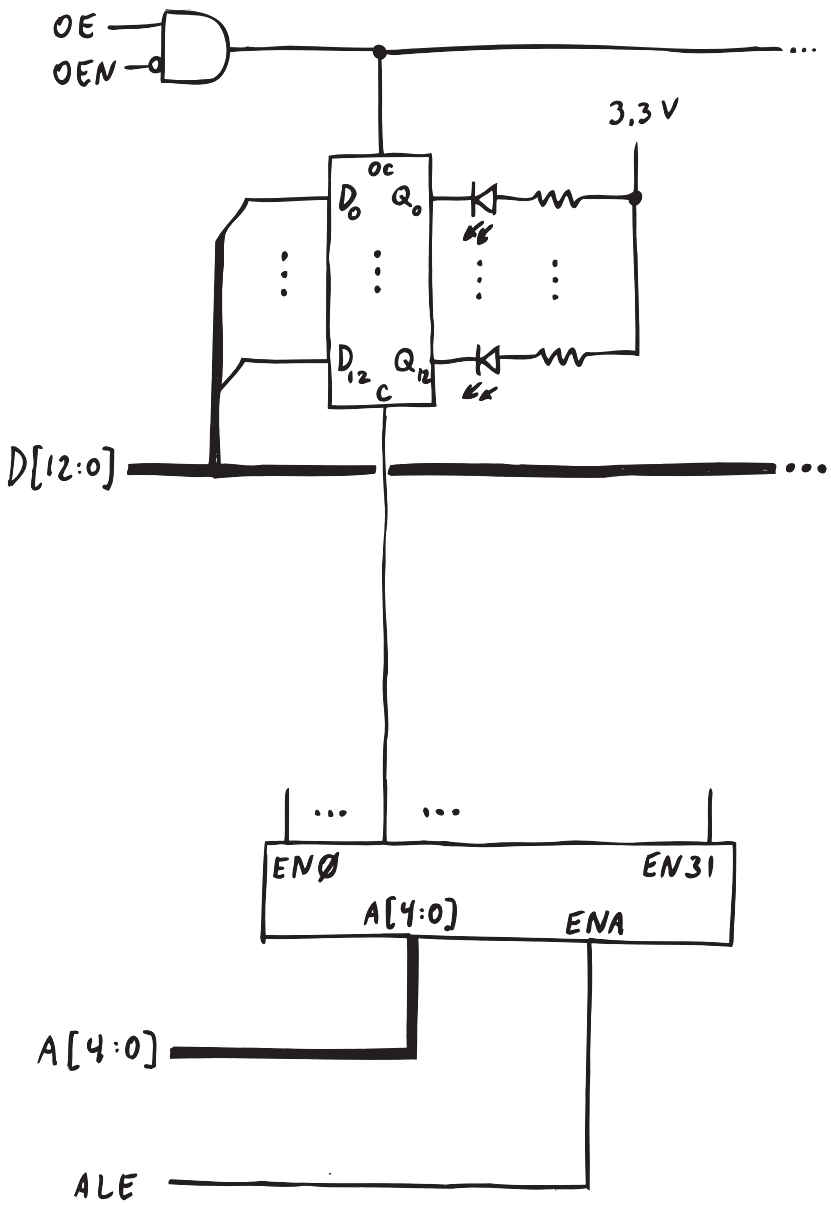


Figure 2.5: One column of the LED PCB logic. While holding the latch enable `ALE` inactive, write the column address to `A[4:0]` and the column data to `D[12:0]`. Then strobe `ALE` active momentarily to write the column data. The LEDs will display the data immediately. Note that since the driver is connected to the LED's cathodes, a data bit value of `0` turns the LED on, and a data bit value of `1` turns the LED off. Immediately after power up, be sure to set the global signal `OE` = 1 and `OEN` = 0 to enable all drivers.

is preferred to reduce cost. In addition, since PCB fabrication costs depend on the number of PCBs that can be arrayed on a raw panel, if possible, it is best to choose PCB sizes that array fully.

4. **Proper physical spacing of LEDs.** Since there are 32 LEDs per row on the LED PCBs, the physical pitch of the LEDs in each row must be 16.8 in $\div$ 32 = 0.525 in. It is desired—but not necessary—that the physical pitch in each column also is 0.525 in.

5. **Adequate trace size and spacing.** PCB-fabrication yield is affected by the sizes of the PCB's traces and the amount of space between them. Finer features are more susceptible to fabrication errors due to contamination. Therefore, using larger traces and spaces can improve PCB-fabrication yields[44]. Unnecessarily fine features should be avoided.

6. **Device packaging that tolerates temperature extremes.** The LED PCBs are mounted in an outdoor windowed enclosure, and so they will be "cooked" in the summer and "frozen" in the winter! For PCB-layer stability, I use through-hole packages because thermal expansion of surface-mount packages can sheer the printed traces off of the PCB[1]. (I am aware that gull-wing surface-mount packages do not exhibit sheer problems, but the PCB layout appears to benefit from the large lead spacing of through-hole packages.)

7. **Adequate power routing.** Each LED PCB can draw as much as 8.32 A (416 LEDs $\times$ 20 mA/LED). Power routing on the board must be sufficient to avoid large voltage drops that would cause variations in LED brightness

[1] Actually, it's *differential* thermal expansion that is important: the difference in the expansion of the package and the PCB that it is attached to.

or malfunctions of the driver chips. One can simplify the design of power routing using dedicated power planes, whose use requires four or more layers in a PCB, or one can add external sheet-metal power strips.

Considering all of these constraints, I conclude the following. Constraints 1 and 2 imply that a standard 18-in × 24-in raw panel can provide just *one* multi-layer PCB or *two*, two-layer PCBs. So to limit costs (constraint 3) I **must use a two-layer PCB**. Constraints 4 and 5 eliminate room between the LEDs and force the drivers to the top and bottom edges of the PCB. Constraint 6 leads to using through-hole devices.

The consequences of constraints 1 through 6 are straightforward, but then I am left with constraint 7. When all of the through-hole devices are connected by adequately sized PCB traces, there is not much room in the PCB layout for power routing.

Power Routing Three options are available for routing power in the LED PCB. Here they are presented, and the choice is justified. The first option is to route power using 1-oz copper traces. With this option, to minimize total voltage drop, we would drive the power traces from the middle of the board and drive the ground traces from the ends. Then the PCB areas farthest from the power source are closest to the ground source, and vice versa, equalizing each LED's applied voltage. The second option is the same as the first, but instead use 2-oz copper foil, which has half of the $R_{\square}$ ("square resistance") or 1-oz copper foil. The third option is to route power through sheet-metal strips with multiple connections along the PCB.

Computations that justify my power-routing decision are in Section A.2 of the Appendix. Using 1-oz copper traces is inadequate. Using 2-oz copper traces would

be adequate, but the boards would cost more. Instead, I prefer using copper strips because power cables need to be connected between the PCB's anyway.

Computer-Aided Design I created the LED PCB design using EAGLE 6.2.0 Professional Edition schematic and PCB-layout software. I note that since most components require exact physical locations on the PCB, I placed nearly all of the design's components algorithmically before routing the traces. My approach was to write an AWK program that generates a text script file for EAGLE which places and rotates each component correctly. Then I used EAGLE's Follow-me router to route the nets. Although EAGLE has a full Autorouter, I was not able to successfully use it on this project with the amount of time that I devoted to the effort (without manual configuration, the Autorouter was unable to route the board completely). Using the Follow-me router actually was sufficient.

PDFs of the LED PCB's logic schematic and PCB layout can be found in the supplemental files on page 84.

2.1.4 Control PCB

Below I detail the design and implementation of the route sign's Control PCB.

Design of the Control PCB

The Control PCB has the overall connections shown in Figure 2.6 below. It provides the system's power supply, it's main microcontroller, and connections to other PCBs and peripherals.

Power Supply The Control PCB has a custom power-supply circuit. Table 2.1 summarizes the route sign's power supply's requirements. The power supply draws

from the bus's 24-V unregulated electrical system and generates three regulated voltages. The sign's 832 LEDs use the 3.3-V supply, with each illuminated LED drawing 20 mA, for a maximum LED current of 16.64A. (Although the sign's firmware never should illuminate all LEDs of the sign simultaneously, the power supply is designed to support this state.) The system's low-power CMOS digital circuitry draws an insignificant amount from the 3.3 V supply as well. The gates of the CMOS LED drivers use 5.0 V. Finally, the 900 MHz radio and the testbed processor require 12.0 V.

Considering the power supply's input voltage, understand that an automotive electrical environment is unexpectedly harsh. While a designer can ensure that a DC-to-DC power converter will operate properly over a range of battery voltages (from a low voltage when the battery is cranking the starter motor to a high voltage when the battery is being charged), such a battery-centric view of the design space fails to reveal worse circumstances. For example, the 24-volt battery of a bus could become disconnected while the engine is running. Then the bus's

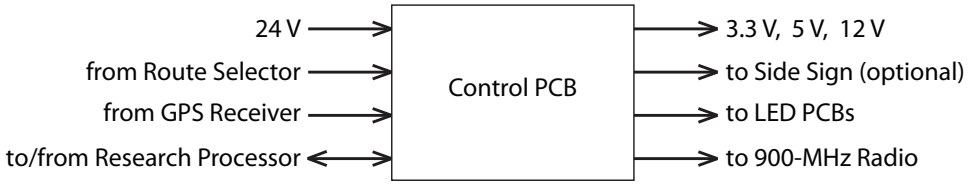


Figure 2.6: Control PCB connections.

Input*	24 V	@	3	A
Output	3.3 V	@	17	A
Output	5.0 V	@	200	mA
Output	12.0 V	@	6	A

Table 2.1: Power Supply Requirements. *Note: the 24-V specification is nominal. The power supply should accept voltage from a normally operating 24-V automotive electrical system of a commercial vehicle.

electrical system would be driven by the unloaded alternator, causing the system voltage to exceed that of a charging battery. Or worse, consider the consequences of a short circuit that blows a fuse. The long supply wire that leads to the short circuit is *not* a perfect conductor: it's an *inductor*. The opening of the circuit's fuse causes a rapid drop in current which induces a huge voltage along the inductive supply wire. I hold that all of a vehicle's powered accessories, including this project's route sign, should be designed to survive exposure to voltage transients such as these.

The International Organization for Standardization publishes standard ISO 7637-2, which covers transients along power lines in automotive vehicles[10]. The Control PCB's power supply is designed for immunity to the worst-case transients of this standard: −600 V and +227 V. (See Test Pulse 1 and Test Pulse 3b in [10].)

To meet this specification, the power supply is split into three separate parts, each of which provides a specific function: *under-voltage protection*, *over-voltage protection*, and *voltage conversion/regulation*. See Figure 2.7.

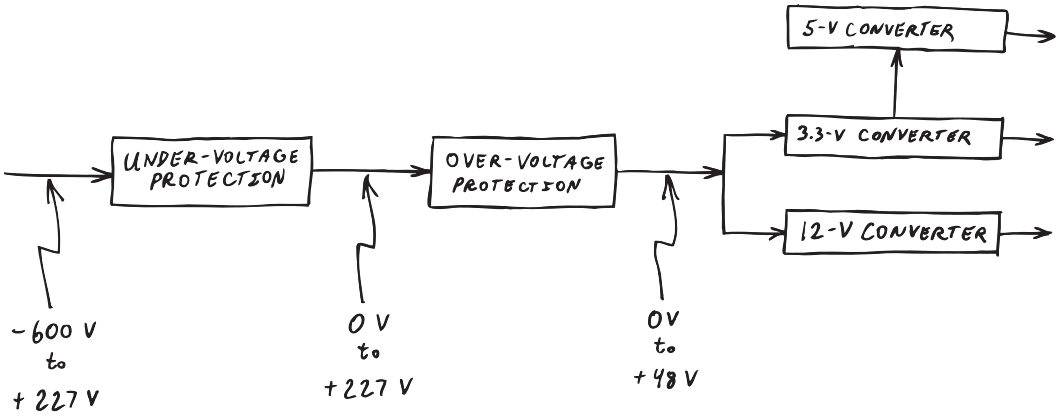


Figure 2.7: Block diagram of power supply.

The under-voltage protection (UVP) circuit simply blocks negative voltages. This circuit is a series-connected 600-V, 12-A Schottky diode. Mounted to a heat

sink, the diode can conduct current sufficient to power the route sign. The UVP circuit feeds the over-voltage protection (OVP) circuit.

The OVP circuit protects the voltage converter/regulator from voltages that exceed the upper limit of the converter's allowed input range. The OVP circuit is based on the MAX6495 integrated circuit by Maxim Integrated Products[21]. The MAX6495 controls an N-channel MOSFET based on the voltage presented to a simple resistor-divider network. When the OVP circuit's monitored input voltage is within the range allowed by design, the MAX6495 pumps the MOSFET's gate to a voltage sufficient to turn it on strongly. If the OVP circuit's input voltage exceeds the circuit's design limit, the MAX6495 immediately turns off the MOSFET by pulling its gate to 0 V.

Computations for the OVP circuit design are summarized in Section A.1 of the Appendix. (A full Excel file for the computations is included in the Supplemental Files listed in the table on page 84.)

The OVP circuit is connected to a set of commercially available DC-to-DC voltage converters/regulators. The converter network functions when the output of the OVP is at least 18 V [18] and [19]. Since the actual voltage of a "24-V" vehicle battery is 27 V and the UVP-circuit diode and the OVP-circuit MOSFET together drop voltage only slightly (less than 2 V total), during normal operation, the power supply's input voltage is more than sufficient to power circuitry on the route sign and all other PCBs.

Microcontroller The route sign must perform five tasks: accept instructions from the route selector, read sensor data, control the sign's LED display, communicate with the testbed processor, and send radio transmissions. The Control PCB performs all of these tasks using a microcontroller and a number of peripherals.

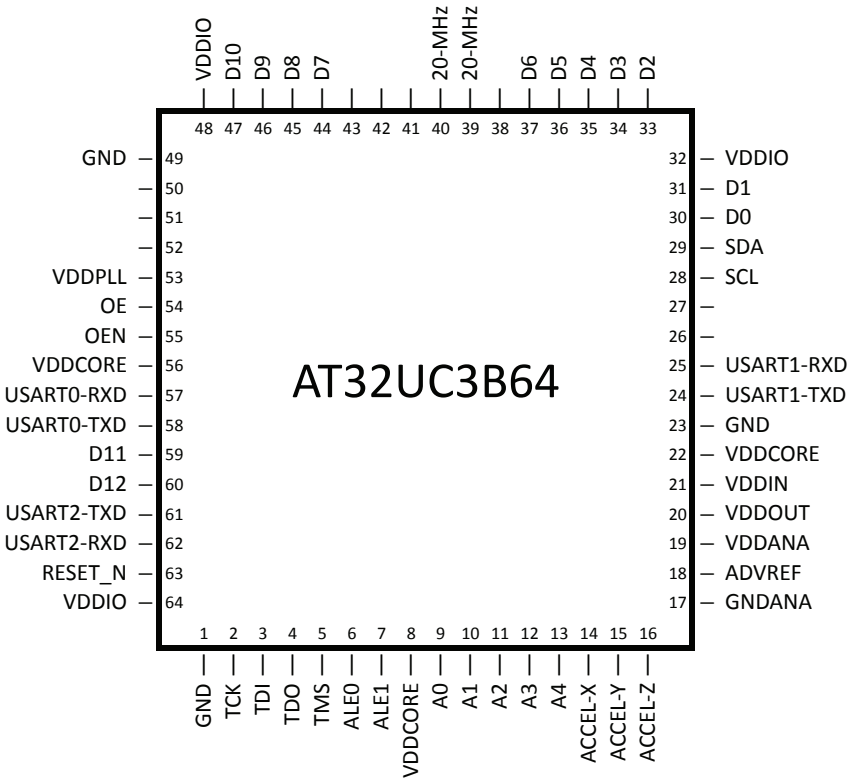


Figure 2.8: Pinout of the Control PCB's microcontroller.

The route sign is controlled by an Atmel AT32UC3B064 32-bit microcontroller with 64K bytes of Flash memory[3]. This integrated circuit includes not only a CPU and Flash memory but also on-board RAM, several peripherals, and 44 I/O pins. Among the microcontroller's on-board peripherals, the route sign uses the I^2C controller, all three of the USARTs (Universal Synchronous/Asynchronous Receiver/Transmitter serial ports), the timer, and the GPIO controllers (General-purpose Input/Output pin controllers). Figure 2.8 shows the microcontroller's pinout. Various peripherals are connected as described below.

General-Purpose I/O General-purpose I/O pins (GPIOs) can be driven high and low under firmware control. Almost all of the GPIOs control the route sign's pair of LED PCBs through microcontroller ports `PA` and `PB` (see Figure 2.9

and Table 2.2).

The Control PCB drives the LED PCBs through a ribbon-cable wiring harness. To ensure signal integrity, the microcontroller's GPIO pins are *not* connected directly through ribbon cables to the drivers on the LED PCBs. Such a direct connection must be avoided for two reasons. First, the microcontroller generates 3.3-V-level outputs while the LVC CMOS-based LED drivers require 5-V-level inputs (specifically $V_{IH} \geq 3.85$ V). Consequently the design passes the microcontroller's signals through a set of SN74LVC4245A active 3.3-V–to–5-V level shifters[32].

Second, directly connecting the level shifters' 24-mA drivers to ribbon cables

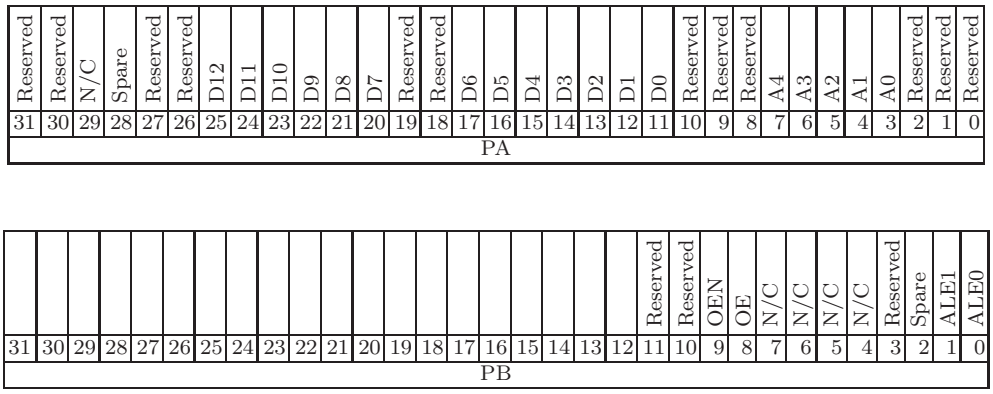


Figure 2.9: Ports PA and PB of the Sign Controller.

`D[12:0]`	Column data. Bit 0 is at the top of each LED PCB.
`A[4:0]`	Column address; word 0 is on the left* edge of each LED PCB.
`ALE0`	Selector for left-hand* LED PCB.
`ALE1`	Selector for right-hand* LED PCB.
`OEN`	Must set to 0 to enable all LED drivers†.
`OE`	Must set to 1 to enable all LED drivers†.

Table 2.2: Port PA and PB bit functions. *Note that the left and right edges of the LED PCB are defined when looking at the LED side of the PCB. †Note that enabling the LED drivers requires *both* `OE` = 1 and `OEN` = 0.

would result in an impedance mismatch and cause the signal waveforms at the LED drivers' inputs to overshoot and undershoot. Since the LED drivers are CMOS devices, their inputs must not be driven above VSS or below GND by more than a diode drop.

It is easy to match the impedance of a driver to that of a ribbon-cable wire using source termination[15] (that is, driving each signal of the ribbon cable through a matching resistor). See Section A.3 of the Appendix for the design computations performed to select the value for the termination-resistors.

Connectors

LED-PCB Connectors The two ribbon-cable connectors of the Control PCB, J9 and J10, share the same pinout, shown in Figure 2.10.

Pins 9, 11, 17, 19, 21	A[4:0]	Column addr
Pins 7, 5, 13, 3, 15, 1, 23, 49, 25, 47, 27, 45, 35	D[12:0]	Column data
Pin 33	ALE0/ALE1	Latch enable
Pin 29	OE	Output enable
Pin 31	EON	Output enable
Pins 2, 4, 6, ..., 50	GND	Signal ground

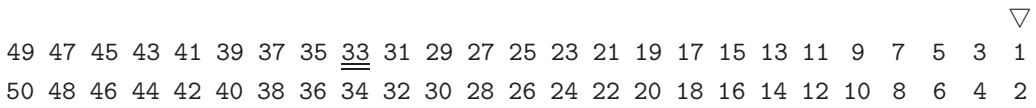


Figure 2.10: Pinout of the two ribbon-cable connectors J9 and J10. The ▽ symbol on the connector denotes pin 1. Every other conductor of the ribbon cable—and hence every even-numbered connector pin—is connected to GND. Note that corresponding signals on both connectors are logically the same except for Pin 33 which is ALE0 on J9 and ALE1 on J10.

GPS-Receiver Connector Localization is performed by a Pharos GPS-500 Global Positioning System receiver. This device generates NMEA 0183 messages over an asynchronous serial data line at 4800 bps[29]. The serial data pin of the GPS receiver drives the microcontroller's USART1 RXD input via device pin 25. (USART1's corresponding TXD output on device pin 24 remains unused, but it is routed on the PCB to a test point labeled `PB2` for potential future use.)

The GPS receiver is powered by the Control PCB through its data connector J13. The connector's pins are defined in Figure 2.11. The GPS receiver's wiring diagram is shown in Figure 2.12.

The microcontroller extracts latitude and longitude from the GPS receiver's GGA message sentences. See [29] for the sentence format.

Connector J13 Pin	Purpose	Wire color
1	3.3-V power to GPS	Red
2	Data from GPS	Blue
3	GND	Black

Figure 2.11: Pinout of the board connector to the GPS receiver.

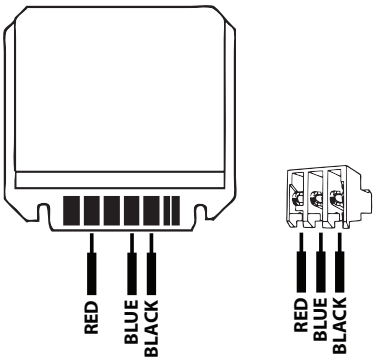


Figure 2.12: Wiring diagram for the custom Pharos GPS-500 cable. Solder three wires to the circuit board of the GPS-500 receiver and insert them in the TE 640441 connector as shown.

900-MHz–Radio Connector Communication with 900-MHz base stations on campus is provided by an AeroComm CL4490-1000 RF transceiver[1]. The microcontroller sends 9,600 bps serial data from its USART2 TXD output on device pin 61, through a MAX3232CSE RS-232 level shifter and a DE-9M PCB-mounted connector, and over a cable to the CL4490-1000 RS-232 serial input. The radio connector's pinout is in Figure 2.13.

Pin 3	TXD to radio
Pin 5	GND

9 8 7 6
5 4 3 2 1

Figure 2.13: Pinout of the Control PCB's RS-232 connector for the 900-MHz radio.

JTAG Connector One can program the microcontroller of the Control PCB over a 2-pin × 5-pin JTAG connector. This connector remains accessible through an access door when the Route Sign is installed in the bus, and so a firmware update, should one become necessary, is possible. The connector follows the pinout that is required by the Atmel *AVR ONE!* device programmer/debugger.

Route-Selector Connector The Route Selector receives 3.3-V power from the Control PCB and returns differential RS-422 serial data that indicates the selected message. Communication and power use connector J2, whose pinout is shown in Figure 2.14.

Pin 1	RS-422 Data IN (+)
Pin 2	3.3 V out
Pin 3	(not connected)
Pin 4	RS-422 Data IN (−)
Pin 5	GND
Pin 6	(not connected)

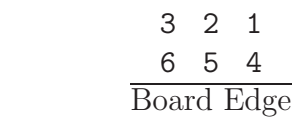


Figure 2.14: Pinout of the connector to the Route Selector.

Side-Sign Connector This project supports future Side and Rear signs. The control data from the Route Selector loops through the Control PCB to a three-pin connector. Either a 100-Ω termination resistor connects the differential data signals of this connector (pins 1 and 2), or the connector drives a cable that sends data to the Side Route Sign. The pinout of the side sign's connector is shown in Figure 2.15.

Pin 3	GND
Pin 2	RS-422 Data OUT (−)
Pin 1	RS-422 Data OUT (+)

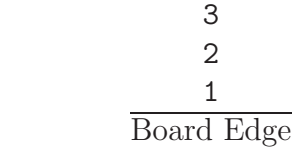


Figure 2.15: Pinout of the connector to the Side Sign.

Testbed-Processor Connector The route sign's microcontroller can communicate with the testbed processor over a full-duplex RS-232 port on USART0. Through TXD the route-sign microcontroller sends the bus's GPS location to the testbed processor, and through RXD the testbed processor returns status messages. RXD is on device pin 57 and TXD is on device pin 58. See the connector pinout in Figure 2.16.

Pin 2	RXD from testbed processor
Pin 3	TXD to testbed processor
Pin 5	GND

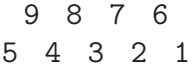


Figure 2.16: Pinout of the RS-232 connector to the testbed processor.

Sensors The Sign Controller includes an option to attach two sensor modules whose data can be sent to the base stations. The data provided by the sensors may improve the accuracy of arrival-time predictions. In addition, one of the sensors provides system temperature readings that can be used during the prototype phase to confirm that the route sign needs no additional thermal management. The inclusion of sensors is motivated by archive data from the BTS project that shows significant GPS offset errors in heavily wooded areas along Heller Drive and McLaughlin Drive between Rachel Carson College and the Baskin School of Engineering. Likely these errors are caused by tall trees attenuating GPS satellite transmissions and preventing position fixes based on the required four satellites.

Since GPS reception cannot be improved, in an earlier project I attempted to compensate for these GPS offset errors using a Kalman filter. I discovered

that changes in the offset error are indistinguishable from bus acceleration when the direction of the offset error is parallel to the road. Consequently, additional information is needed.

To help disambiguate acceleration and changes in offset errors, the first optional sensor is an MMA7361L accelerometer by Freescale Semiconductor[11]. The accelerometer is mounted with its +X axis pointed right, its +Y axis pointed down, and its +Z axis pointed forward. Its data are read through three of the microcontroller's analog inputs as shown in Table 2.3.

Axis	Microcontroller Pin
X	PA8
Y	PA30
Z	PA31

Table 2.3: Accelerometer connections.

Since the MMA7361L maps an acceleration range of $[-1.5\text{ g}, +1.5\text{ g}]$ into a voltage range of $[0\text{ V}, 3.3\text{ V}]$, one obtains accelerations in g's by converting the accelerometer's X, Y, and Z pin voltages using Equation 2.1.

$$\text{g} = \frac{\text{voltage} - 1.65}{1.1} \tag{2.1}$$

The second optional sensor directly determines the orientation of a bus, and therefore its direction of travel. Although one could use changes in a bus's GPS position to determine the bus's orientation, this technique fails when the bus is stopped. Therefore, the Sign Controller includes an optional Freescale Semiconductor Digital Magnetometer[12]. The MAG3110 is mounted with its +X axis pointed up, its +Y axis pointed right, and its +Z axis pointed backward. Its data are accessed through the microcontroller's I^2C interface with 7-bit address `0x0E`.

EEPROM The Control PCB stores one small data value: a unique Bus ID. To store this value, the Control PCB includes a 256-byte AT24HC02B EEPROM by Atmel. The EEPROM is connected to the microcontroller's I^2C interface and uses the 7-bit I^2C address `0x50`.

Layout of the Control PCB

The layout of the Control PCB is entirely conventional. Unlike that of the LED PCBs, where nearly all components of the LED PCBs have exacting physical-placement requirements, the components of the Control PCB have no such constraints, aside from a few power connectors and the corner mounting holes. In addition, while the overall dimensions of the LED PCBs need to be identical to those of the PCBs that they replace in the sign enclosure, the Control PCB has no specific dimensional requirements.

The two relaxed layout constraints mentioned above are items 1 and 4 of the LED PCB's enumerated list. Since the remaining items can apply, I review the list next.

1. **Proper physical dimensions, including thickness.** Does not apply to Control PCB. I am free to use ordinary 62-mil material.

2. **Maximum PCB size.** Given a maximum panel size, the individual PCBs that are cut from the panel have natural sizes themselves, just as the LED PCBs do. Looking at preliminary layouts for the Control PCB, I chose a PCB size that just arrays eight times per panel. Any reduction in size that fails to increase the number of PCBs per panel is unnecessary.

3. **Cost-conscious fabrication.** I am able to use a two-layer PCB instead of a multi-layer PCB. The layout of the PCB is sufficiently loose that there is

room for power routing.

4. **Proper physical spacing of LEDs.** Does not apply to Control PCB.

5. **Adequate trace size and spacing.** As with the LED PCB, I use trace widths and spaces that do not push manufacturing limits.

6. **Device packaging that tolerates temperature extremes.** As with the LED PCBs I use through-hole device packages. I also use several gull-wing surface-mount packages: their lead flexibility should prevent trace-delimitation problems from differential thermal expansion. One device on the PCB is in a non-gull-wing surface-mount package (the MAX6495 OVP circuit controller). This device is available only in a "QFN" package (a small rectangular package with *no* pins—just bottom pads). As I have explained, I avoid such packages for PCB-reliability reasons, but the package's small 3-mm×3-mm size makes the magnitude of any differential thermal expansion small and unlikely to cause problems.

7. **Adequate power routing.** While the Control PCB does not have 416 LEDs, it *does* power them (actually it powers 832 LEDS, as well as several 5-V and 12-V devices). The looseness of the PCB layout allows power routing through "copper pours," which are regions of mostly copper. This layout technique reduces the "squares" of the power-supply traces substantially: for the unregulated 24-V, 3-A power input $N_{\square} < 2$; for the 3.3-V, 17-A LED supply $N_{\square} < 0.4$; and for the 12-V, 6-A supply $N_{\square} < 6$ (which is adequate given that it powers devices with regulated switching supplies themselves).

Computer-Aided Design As with the LED PCB, I created the Control PCB design using EAGLE 6.2.0 Professional Edition schematic and PCB-layout software. All gull-wing surface-mount devices are mounted on the same side of the PCB to avoid needing to apply solder to both sides. As with the LED PCB, I found that it was best to use the Follow-me router to route the PCB traces manually. Figure 2.17 is a photograph of the top of the Control PCB.

2.1.5 BTS 2 Front-Sign Firmware

Requirements

The BTS and BTS 2 bus nodes perform different sets of tasks. While the GNU Linux PC of BTS ran testbed algorithms, the BTS 2 route-sign processor controls the LED display, communicates with the route selector, two sensors, the testbed processor, and the EEPROM memory.

Consequently, just as with the BTS CPU, the BTS 2 sign controller also must run multiple tasks simultaneously. See Table 2.4.

	Controller	
Task	**BTS**	**BTS 2**
Accept data from GPS receiver	●	●
Send data to digital radio	●	●
Run testbed algorithms	●	○
Control LED display	○	●
Accept name changes from Route Selector	○	●
Poll accelerometer and magnetic compass	○	●
Accept status messages from testbed processor	○	●
Update bus-ID of EEPROM when instructed	○	●

Table 2.4: Controller task comparison. BTS bus nodes used a GNU Linux PC, while BTS 2 bus signs are controlled by a microcontroller-based embedded system. In BTS 2, testbed algorithms run on a dedicated testbed processor.

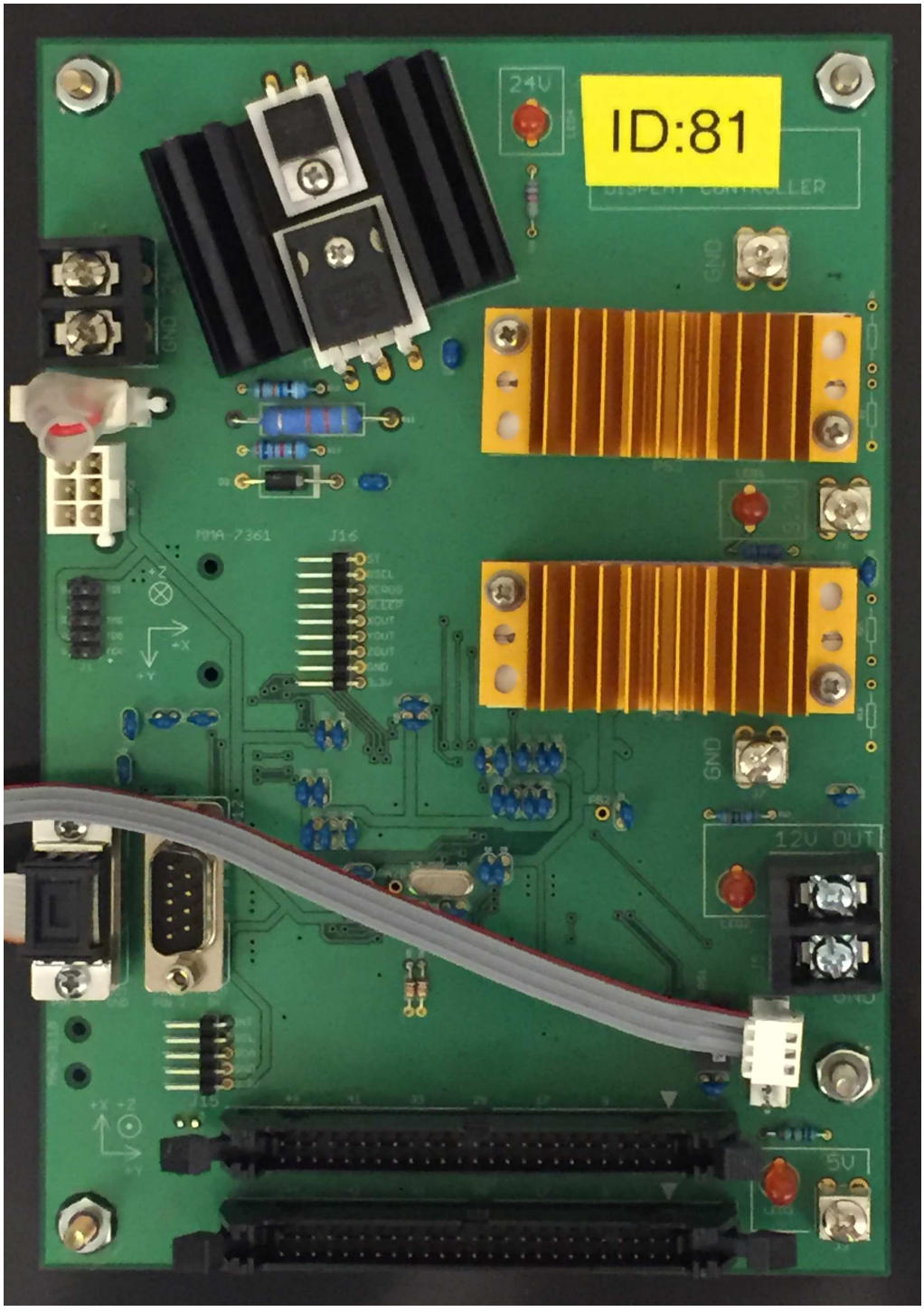


Figure 2.17: Control PCB.

BTS 2 Real-Time Task Management

The BTS 2 route sign's firmware does not use an RTOS (real-time operating system) or scheduler. Instead, it manages its real-time computations using a *cyclic executive*[17, pp. 75–78]. This approach is not unusual. Results of the EE Times 2012 Embedded Market Survey show that a third of embedded systems projects lack an operating system or scheduler of any kind[34, slide 25].

An embedded system like BTS 2 can perform real-time computations without a real-time operating system by following three specific design principles[9].

Principle 1. To ensure that tasks are completed on-time, *one must design for the peak load instead of the average load*[2]. Microcontrollers of the bus node receive periodic input events that have limited arrival rates or that come from sensors whose values are queried at a fixed rate by the microcontroller itself. Consequently, task scheduling can be static rather than dynamic, meaning that several features of RTOSes can be ignored as unnecessary complications.

Principle 2. *Given predictable, periodic events, one can split event servicing into "hard real-time" and "soft real-time" portions.* A hard real-time computation is one whose deadline cannot be missed without causing a catastrophe. Retrieving data from a serial-port peripheral before the buffer overruns is an example of avoiding an embedded-system catastrophe. A soft real-time computation is one whose completion time affects just performance, such as updating the message on the LED display.

Given these two definitions, consider processing characters from an I/O device. Split the processing like this: a small **hard real-time** interrupt routine moves one

[2]This idea is captured by the wry remark, "Then there is the man who drowned while crossing a river with an average depth of six inches." On page 7 of [9] this quote is attributed to the notes of John Stankovic. John Stankovic, and many on the World-Wide Web, attributes it to W.I.E. Gates. But the identify of W.I.E. Gates remains elusive.

character from the single-byte hardware buffer into a longer, multi-byte software buffer, and then a lower priority **soft real-time** routine processes the string of the multi-byte buffer. The only connection between the two portions is that the hard real-time part activates the soft real-time part through the existence of characters in the multi-byte buffer[9, page 16].

Principle 3. The third principle that allows real-time computations without an ROTS is to *ensure processing predictability*. One should avoid hardware features that help the average case at the expense of the worst case, for example DMAs and caches. This recommendation is easier to follow when using a 32-bit microcontroller because such devices lack advanced hardware features but have sufficient performance. Although using a single processor helps predictability, one can use multiple processors (either symmetric or asymmetric) as long as each one performs a predetermined set of tasks (static scheduling). The dynamic scheduling of symmetric multiprocessing (SMP) would cause complications that are better served by an RTOS, and so lacking an RTOS, it is best to avoid SMP.

On the software side, one should avoid dynamic data structures, demand paging, and recursion. One can (and should) prefer loops with a predetermined upper bound on the number of iterations. Also, one needs to ensure that soft real-time tasks are interruptible, and one should reduce interrupt service routines to bare minimums. In addition, one must ensure that the interrupt rate is system-limited and that interrupts are disabled for only a limited time—if at all.

The microcontroller and firmware of the Control PCB follow the principles just described: short interrupt routines service I/O, and tasks are implemented as functions that are called repeatedly from an infinite loop. The microcontroller overprovisions the system with more than ample memory and performance, since a 32-bit microcontroller with moderate capabilities is quite affordable.

2.1.6 Wiring Harness

The Control PCB sends address, data, and control signals to the two LED PCBs over a ribbon-cable wiring harness (Figure 2.2, item 7). Each of the existing route signs has four ribbon cables with compatible connectors, at least two of which appear to have sufficient length to be used in the harness, and so as long as quality is maintained, I reuse the longest of the old ribbon cables in the new route signs.

2.2 Route Selector

The bus driver twists the knob of the Route Selector to choose a new message for the Route Sign. Each click of the knob shows a new message on the Route Selector's display and sends the message to the Route Sign over an RS-422 serial link. (Figure 2.1.)

2.2.1 Enclosure

I have chosen a Hammond Manufacturing 1455J1202BK aluminum enclosure for the Route Selector. The exposed metal of this model is anodized black to reduce the glare of the sun from its surface. When one mounts the enclosure in the cab area of the bus, it should be located for easy access by the driver and oriented so that the driver can read the message on the LCD.

The enclosure has internals slots that align and hold the Route Selector's PCB.

2.2.2 Route-Selector PCB

Design of the Route-Selector PCB

The Route-Selector PCB has a simple interface: power, rotary encoder, serial data out, and serial data in. The 3.3-V input power comes from the Route Sign. The same four-conductor cable that supplies that power also carries serial data back to the sign to control the displayed message. The position of the rotary encoder is determined by the states of the encoder's contacts as sensed by a three-conductor connection to the PCB. For programming, data from the Route Programmer is received as serial input data.

The PCB is controlled by a microcontroller that has several peripherals, as described below.

Power Supply A monolithic power converter creates 5 V for the LCD from the 3.3 V supply.

Microcontroller The Route Selector performs these tasks: determine the position for the rotary encoder, send selected serial messages to the Route Sign, and receive a set of new messages from the Route Programmer.

To perform these tasks, the Route Selector uses the same kind of Atmel microcontroller that is used in the Control PCB of the Route Sign. Among the microcontroller's on-board peripherals, the Route Selector uses the I^2C controller, two of the USARTs, the timer, and the GPIO controllers. Figure 2.18 shows the microcontroller's pinout. Various peripherals are connected as described below.

General-Purpose I/O The microcontroller's GPIOs set the LCD message and read/write other peripherals through ports `PA` and `PB` (Figure 2.19 and Table 2.5).

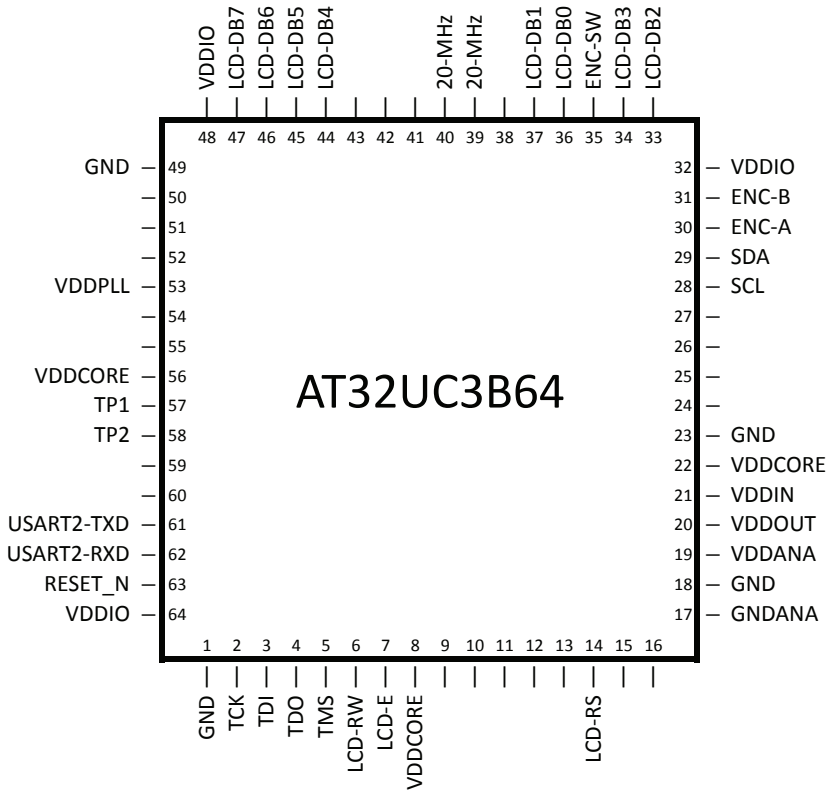


Figure 2.18: Pinout of the Route Selector's microcontroller.

				Reserved	Reserved			LCD-DB7	LCD-DB6	LCD-DB5	LCD-DB4	Reserved	Reserved	LCD-DB1	LCD-DB0	ENC-SW	LCD-DB3	LCD-DB2	ENC-B	ENC-A	Reserved	Reserved	LCD-RS				Reserved	Reserved	Reserved
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PA																															

																				Spare	Spare						LCD-E	LCD-RW
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PB																															

Figure 2.19: GPIO of the Route Selector.

Figure 2.20: Rotary-encoder timing diagram. Clockwise direction.

LCD-DB[7:0]	LCD 8-bit command/data.
LCD-RS	LCD register set: 0 = command, 1 = data.
LCD-RW	LCD read/write: 0 = write, 1 = read.
LCD-E	LCD falling-edge clock.
ENC-A	Rotary-encoder phase A.
ENC-B	Rotary-encoder phase B.
ENC-SW	Optional pushbutton: 0 = pressed, 1 = released.

Table 2.5: GPIO purposes for the Route-Selector PCB.

Connectors

JTAG Connector One can program the microcontroller of the Route Selector over a 2-pin × 5-pin JTAG connector. The connector follows the pinout that is required by the Atmel *AVR ONE!* device programmer/debugger.

Route-Sign Connector The Route Selector receives 3.3-V power from the Route Sign and returns differential RS-422 serial data that indicates the selected message. Communication and power use connector J2, whose pinout is shown in Figure 2.21.

Pin 1	RS-422 Data OUT (+)
Pin 2	3.3 V in
Pin 3	GND
Pin 4	RS-422 Data OUT (−)

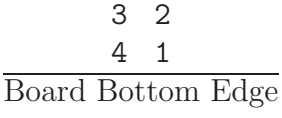


Figure 2.21: Pinout of the connector to the Route Sign.

Route-Programmer Connector The Route Selector drives 3.3-V power to the optional Route Programmer and receives differential RS-422 serial data that specifies new EEPROM programming. Communication and power use connector J3, whose pinout is shown in Figure 2.22.

Pin 1	RS-422 Data IN (+)
Pin 2	3.3 V out
Pin 3	(not connected)
Pin 4	RS-422 Data IN (−)
Pin 5	GND
Pin 6	(not connected)

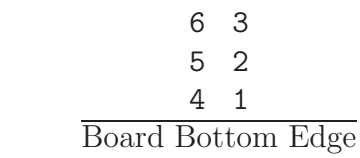


Figure 2.22: Pinout of the connector to the Route Programmer.

EEPROM A 256-byte serial EEPROM stores two data structures. The initial 248 bytes of the memory contain a list of route names. The names are represented as a series of `'\0'`-terminated strings with the final route name of the list followed by a second `'\0'`. The second data structure is the index of the most recently selected route name. It is stored as a 32-bit unsigned int at addresses 248–251 of the EEPROM.

Layout of the Route-Selector PCB

1. **Proper physical dimensions, including thickness.** I chose an enclosure that has internal PCB-alignment slots, which forces the PCB size to be 4.700 in × 2.933 in × 0.062 in. (These dimensions are the size recommended

in the enclosure's product drawing [14] less 0.020 inch in length and width to allow for PCB fabrication tolerances.)

2. **Maximum PCB size.** (not applicable)

3. **Cost-conscious fabrication.** I am able to use a two-layer PCB instead of a multi-layer PCB. The layout of the PCB is sufficiently loose that there is room for power routing.

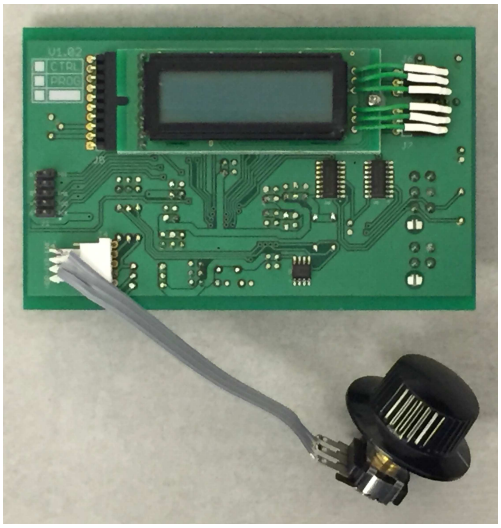
4. **Proper physical placement of components.** Sides of the 0.125-in-deep PCB-alignment slots of the aluminum enclosure could touch the copper-foil traces of the PCB. In the worst case, if power were routed on one side of the PCB and ground were routed on the other, the metal fingers that form the slots could cause a short circuit across the power supply! To avoid any such troubles, all copper is etched back 0.020 inch from any enclosure metal that contacts the PCB. In addition to trace placement, I must be mindful of component headroom. The components on the PCB are placed to avoid any physical obstructions of the enclosure and the rotary encoder that mounts on it. In addition, the LCD display is mounted so that it just touches the inside of the enclosure where a windows is cut in the enclosure.

5. **Adequate trace size and spacing.** As with all other PCBs, I use trace widths and spaces that do not push manufacturing limits.

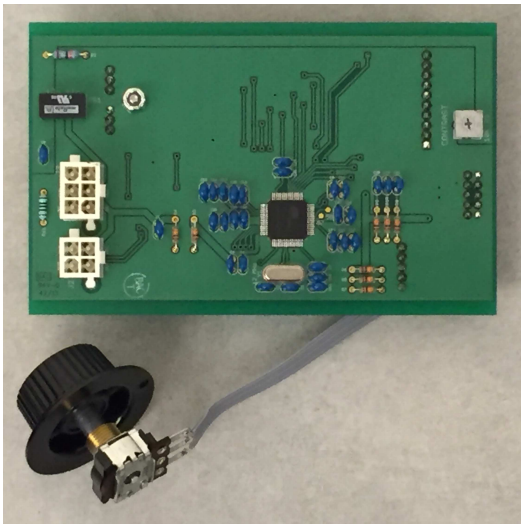
6. **Device packaging that tolerates temperature extremes.** As with all other PCBs of this project, the Route-Selector PCB uses through-hole device packages and gull-wing surface-mount packages for mechanical stability over a wide temperature range.

7. **Adequate power routing.** All components of the PCB use little power. Regardless, copper fills on both sides of the PCB are connected to GND, and all power traces are at least 2 mm wide.

Computer-Aided Design I created the Route Selector PCB using EAGLE 6.2.0 Professional Edition software. Lessons learned from designing this PCB are mentioned in the Chapter 5. Figure 2.23 contains photographs of the resulting PCB.



(a) Top view showing the LCD display, the RS-422 receiver and transmitter chips, the serial EEPROM, and connectors for the JTAG programmer and the rotary encoder.



(b) Bottom view showing the 3.3-V-to-5-V DC-to-DC converter, connectors J2 and J3, the microcontroller and its crystal, and the LCD contrast control.

Figure 2.23: Route selector PCB.

Firmware of the Route-Selector PCB

Undergraduate researcher Ben Cizdziel wrote the initial version of the firmware of the Route Selector's microcontroller. I replaced the first version's interrupt-based design with a cyclic executive when the interrupt-based design was shown unable

to handle pushbutton debouncing.

The cyclic executive runs several tasks:

- Sample and debounce the rotary encoder and determine when a new route name has been chosen.
- Transmit a newly selected route name to the route sign on the serial output port.
- Receive new EEPROM programming from the Route Programmer on the serial input port.
- Show the selected route name on the LCD display and animate the scrolling of long names.

2.2.3 Route-Selector Cable

The pinouts of the cable that connects the Route Sign and the Route Selector can be inferred from Figures 2.14 and 2.21, which show pinouts of the corresponding board connectors.

2.3 Route Programmer

The Route Programmer performs two main functions: (1) let TAPS edit a local copy of the route names and (2) transmit the new route names over a cable to a bus's Route Selector. Since the Route Programmer serves additional functions, its interface has three rotary-encoder knobs instead of one, two pushbuttons, and two LEDs.

2.3.1 Enclosure

I have chosen a Hammond Manufacturing 1455J1202 clear aluminum enclosure for the Route Programmer. Aside from color, this enclosure has the same dimensions and characteristics as those of the Route Selector.

2.3.2 Route-Programmer PCB

The Route Programmer reuses the Route Selector's PCB. The PCB lacks sites for RC networks of two of the rotary encoders and one of the pushbuttons, and it lacks sites for current-limiting resistors for the two LEDs. These components will be soldered directly to the LEDs, encoders, and pushbutton. Figure 2.24 shows the Route Programmer firmware-development prototype in all of its glory.

Firmware of Route-Programmer PCB

The firmware for the Route Programmer is being written by Christopher Villalpando.

2.3.3 Route-Programmer Cable

The pinouts of the cable that connects the Route Programmer to the Route Selector can be inferred from Figures 2.21 and 2.22, which show pinouts of the corresponding board connectors.

2.3.4 Wall Power Adapter

To power the Route Programmer when it is not plugged into a bus's Route Selector a 3.3-V "wall-wart" style power supply is wired to a connector. The power

connections to the connector can be inferred from the corresponding board connector in Figure 2.21. The RS-422 data pins 1 and 4 of the connector are left unconnected.

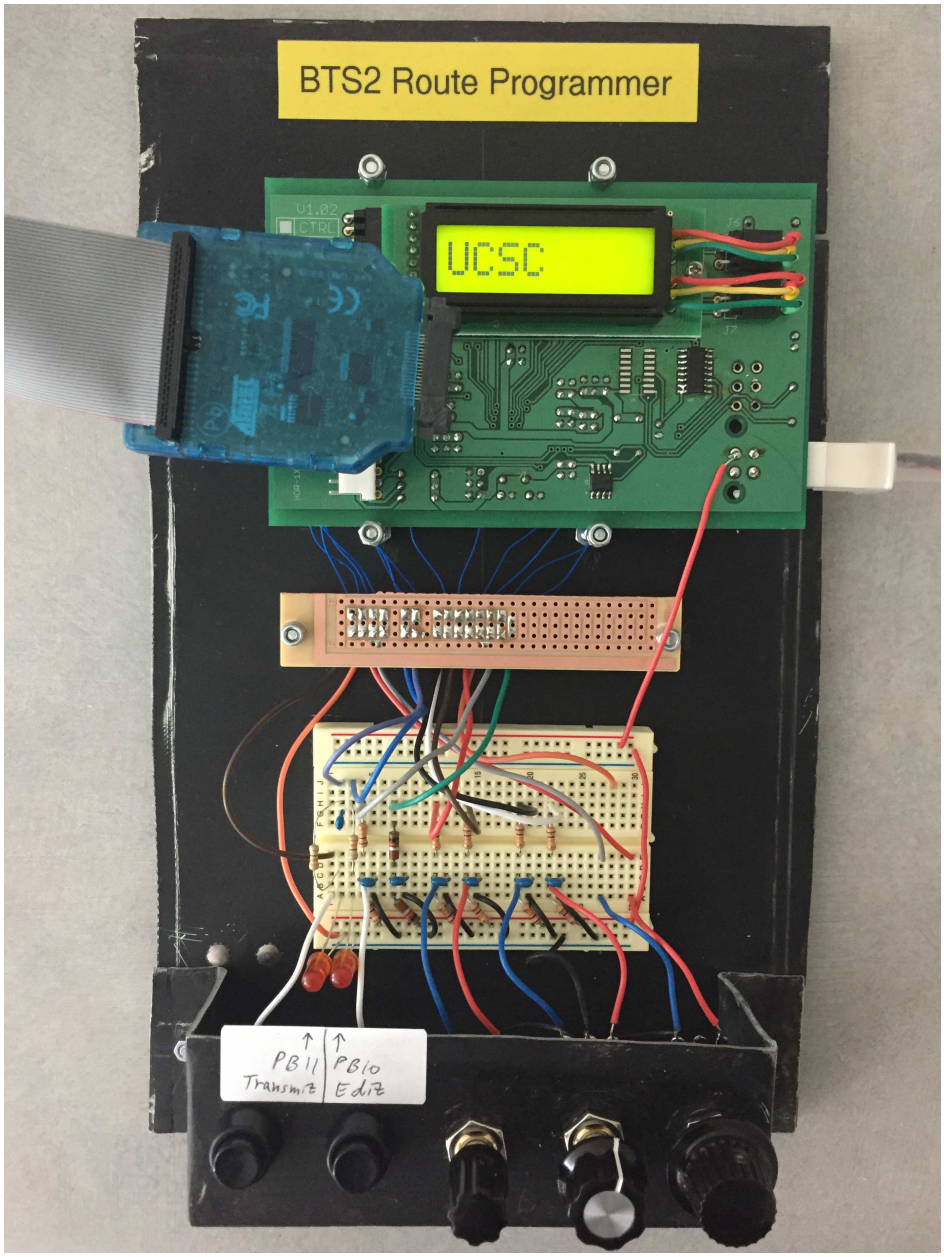


Figure 2.24: Route Programmer firmware-development prototype.

Chapter 3

Base Stations and Servers

Figure 1.3 in Chapter 1 shows that the base stations, the database server, and the web server are connected through a communications network. Information packets, starting with character strings that are sent over a 900-MHz radio link from a bus node to a base station, pass between selected pairs of these computers. This chapter presents the system design and the design and implementation of the programs that run on these computers. Since I influenced the overall design of this portion of the BTS system architecture, and since BTS 2 programs are modifications of the original BTS programs, I present both the BTS and BTS 2 systems here.

I modified this portion of the BTS system for BTS 2 rather than use it unchanged for three reasons. First, I want to improve usability and tracking quality by allowing transmission, storage, and analysis of additional data from the bus nodes (route name, acceleration, and compass heading). Second I want to correct an unchecked buffer overflow error that I discovered in the BTS base-station code. Third, I want the system to detect and reject erroneous packets that pass through the 900-MHz link by adding a badly needed checksum.

In both BTS and BTS 2 systems, packet processing proceeds in four unsynchronized phases which I define below. The foremost goal of this organization is to ensure that the web-server load does not affect the loads of the system's other computers, although my organization isolates load effects in other parts of the system, too. Like some kinds of sensor networks, the system uses aggregation in a hierarchical network to control computational bandwidth as data travels from the bus nodes to the system's "sink" (the web server). That said, I must be careful when using the sensor-network analogy because the system does *not* form an ad hoc network for collection of node-location data. The network diameter is fixed. All data traverses similar paths as it travels through the processing phases:

(Phase 1) *Bus node: store GPS coordinates.* This event occurs once per second on a bus node when the GPS receiver sends location data over its wired 4800-baud link. Only the most recent location is retained; old locations are overwritten. (Phase 2) *Bus node: transmit data to a base station:* The bus node periodically assembles location coordinates into a data string and then transmits the string over a 900-MHz radio link toward a base station. The base station extracts data from the data string and writes it into the BTS 2 system's MySQL database. Such transmissions are limited to a rate of twenty times per minute for each bus node. (Phase 3) *BTS 2 server: periodically query the database to create an XML data file of bus locations.* Once the XML data file is created, it is written to the BTS 2 web server. Creation of the XML file occurs every three seconds. (Phase 4) *BTS 2 web server: serve the XML file to any desktop and mobile clients that demand it.* The rate of this event depends entirely on the number of clients using the bus-location data.[1]

[1]For the BTS 2 project, Kerry updated the XML-file generation algorithm that ran on the INRG lab's "skynet" server. The XML file from this prototype algorithm was served to several smartphone apps and the bts.ucsc.edu web site. Later, to move these computations off of the lab

3.1 Processing on the Base Station

Processing for phase 2 started on a bus node with a data transmission. Phase 2 continues on the base station when the data transmission is received.

3.1.1 Evaluation of the Original BTS

The base-station program

`/trunk/basestation/aero.cpp`

receives the data transmission from the bus node. The program creates a file `UNIT#.TXT` containing the packet data, formatted slightly differently. (Each bus has its own `UNIT#.TXT` file on the base station, where `#` is the bus ID.)

Reading the code, I discovered that the `areo.cpp` program can suffer from a buffer overflow because it does not limit the number of received packet characters that it writes to its buffer. Also, since the packets lack any sort of checksum, the file `UNIT#.TXT` may contain data from more than one source, or it may contain partial data. Archive data in the BTS database's coordinates table shows that it is likely that these kinds of receive errors have occurred.

For example, three records have bus IDs of -122 (valid bus IDs are positive), three records have subversion revisions of -122 (subversion revisions are positive), four records have latitudes < 4 (should be ≈ 37 for Santa Cruz), and three records have longitudes > -25 (should be ≈ -122 for Santa Cruz). All of these records strongly suggest that there has been a misalignment of data during packet processing.

server, undergraduate Wade "Simba" Khadder and graduate Kevin Abas wrote the production algorithm and migrated it to UCSC IT-maintained servers. They also switched from an XML file format to a JSON file format, wrote the Slug Route iPhone and Android apps, and wrote the `bts.ucsc.edu` web app. Since all of the production apps access the JSON file, the prototype XML file generator no longer is running on "skynet".

The separate base-station program

`/trunk/basestation/btsManager.py`

inserts data from the file `UNIT#.TXT` into the MySQL database using functions found in `btsFunc.py`.

A final comment on the original BTS base stations concerns reliability. Occasionally BTS-related daemons on the base stations would terminate. The solution chosen by the BTS team was to schedule a cron job to reboot each base station automatically every day.

3.1.2 Improvements for BTS 2

The base stations for BTS 2 use completely new hardware and extensively updated software. Here are the specific changes made in creating the BTS 2 base stations.

First, to address the BTS base-station reliability problem, I observe that the base station code relied on a daemon compiled from GCC code and a separate daemon that interpreted a Python script. Also, some processing was performed through an `sh` shell script. Since the quality of any system is the product of its subsystems' qualities, it is best to reduce the number of unique subsystems. To that end, I replaced the `sh` script and the Python-based daemon with a completely new GCC-based daemon, thereby eliminating the system's reliance on a Python interpreter and on `sh` scripts. Base-station quality appears to have improved.

I also corrected the BTS base-station's buffer-overflow bug, and I added code to parse each data frame and verify the checksum before writing data to the MySQL database.

3.2 Servers

3.2.1 Evaluation of the Original BTS

In the original BTS system, a daemon on the server (`/etc/init.d/xmlManager`) ran a Python program called `xmlManager.py`. This program queried a MySQL database table (`coordinates_new`) which contained current bus coordinates. The daemon wrote the database information to an XML file (`/var/www/bts/coord.xml`). Then an Apache web server process served this XML file to clients that requested it (as `http://skynet.soe.ucsc.edu/bts/coord.xml`)[2]. The Python program was fairly simple and reliable.

The XML file contained the most recent longitude and latitude of each bus as well as each bus's bus ID. A timestamp helped identify active buses. This information was sufficient to plot active buses on a map.

3.2.2 BTS 2 Server

For BTS 2, the `xmlManager.py` program remains essentially unchanged, but it adds columns for each bus's current route name and its arrival predictions. The route names come directly from each bus node's route sign. The arrival predictions are generated by an additional cron job and script.

To help with arrival predictions, the BTS 2 database adds these new tables:

Table	Description
`busstops`	Latitude and longitude coordinates of the campus's bus stops.
`routes`	Segments of routes and their estimated stop-to-stop durations.
`zones2`	Latitudes and longitudes of rectangular geofence zones that are used by the arrival-prediction algorithm.

[2]The prototype server no longer is running. The production server generates a JSON file at `http://bts.ucsc.edu:8081/location/get` .

Creating arrival predictions is straightforward. A cron job performs these tasks:[3]

1. Identify each bus's location using geofences defined in the `zones2` table.
2. Determine the next bus stop and direction of travel for each bus based on the last three zones that it has passed through.
3. Predict each bus's arrival times for its future bus stops by summing estimated stop-to-stop durations from the `routes` table.
4. Update each bus's MySQL database entry with the new arrival predictions.

All steps except for 3 are performed in the MySQL database using `update` commands. Step 3 is performed by a script.

It is possible in a future student project for arrival predictions to respond dynamically to the time of the day and the day of the week, and whether the campus is open or not.

[3]Arrival predictions used to be provided by "skynet", but since its hardware failure, arrival-prediction services have been offline. Those services will at some point return on the new UCSC IT-maintained servers.

Chapter 4

Testbed Management

This chapter documents work performed by SURF-IT students who worked in the Inter-Networking Research Group on testbed management for SCORPION. The students' work never was published outside of SURF-IT papers and posters. The PI of the iNRG, Katia Obraczka, who also is the chair of this project-report's reading committee, asked me to review the testbed-management code, uncover student presentations, and create a single usage and implementation guide for the SCORPION testbed-management system. On the PI's recommendation, portions of that document are included here, as Chapter 4. I discuss management of an ad hoc network that is formed by optional testbed processors of the bus nodes via WiFi links. The commands described below managed the Mini-ITX version of the testbed. Be aware that that version of the testbed has been retired, and no replacement has been created yet.

Fixed wireless nodes can be controlled continuously since they can remain connected to their network backchannel. This is the case of MoteLab[39] and

ORBIT[27]. To achieve the same level of management with *mobile* wireless nodes (particularly those that test disruption-tolerant networks, or DTNs) testbeds require either a dense network of WiFi access points or nodes with secondary 3G links, as are provided with DieselNet[8][30]. However, including such communications infrastructure in a testbed adds to both deployment costs and maintenance costs. When inclusion of backchannels does not fit into researchers' budgets or plans, DTN testbeds must be designed to operate properly when they are managed through only sporadic connections.

The design of a testbed interface for network researchers needed to confront SCORPION's lack of a continuous administrative backchannel. Although the designers contemplated using the testbed's DTN network itself for administrative communication, they decided that it is prudent not to require perfect operation of an experimental network in order to retrieve the data generated by the networks experiments. Consequently, they created a simple but robust set of commands that do not rely on the DTN working correctly.

Network researchers can reprogram nodes and can control them using the commands described in Sections 4.1 and 4.2. Since all nodes are deployed from a location near a gateway at the beginning of a test and then collected later at the test's end, initial configuration and final data collection tasks assume that nodes are one hop away from a gateway. This single-hop assumption relieves the nodes of the additional task of routing data logs to a sink, since researchers do not want to require the routing protocol under test to operate correctly in order to diagnose its operation! Additionally, they do not want collection of experimental logs to interfere with data transmission carried out by the protocols under test.

Tests using bus nodes require special planning because bus nodes are configured by a gateway near the transportation service's fuel pumps. Network re-

searchers must allocate sufficient time for all bus nodes to be configured before the start time of a test, and they also must wait for a sufficient time after a test for stored data logs to be collected from all buses[1].

4.1 Test-run Management

A researcher configures and initiates tests on the testbed's nodes wirelessly by running a set of four commands on a gateway. Usually the gateway is a laptop computer with a wireless port that has been configured to communicate with the nodes. The utilities, which are named after Linux commands, are simple enough for interactive use, but they can be scripted as well.

4.1.1 nodels

Pronuncation: *node-L-S*. For the purpose of learning the statuses of nearby nodes, this gateway command lists the numbers and characteristics of testbed nodes in range. The listed characteristics include the operating-system kernel version, the time since booted, the latitude and longitude (and whether a GPS fix is available), the subversion revision number of the source-code directory, disk usage, and CPU usage.

The `nodels` command listens for responses from nodes for 1.5 seconds. With the optional `-t` parameter, one can change this duration. The value following `-t` is a floating-point number representing the time to wait in seconds.

[1] Evaluation of the original BTS project revealed that bus drivers turn off bus power while the vehicle is being fueled. Consequently, each bus node's Mini-ITX computer lost power immediately after the bus arrived at the fuel pumps, shortening node-management time. I recommend that testbed nodes of BTS 2 include a small backup battery that will run the testbed CPU for a sufficient period of time for node management. The testbed CPU will need a power-fail detector to trigger a system shutdown once maintenance is complete.

Usage is `nodels` [`-t` *seconds*] .

4.1.2 nodediff

For the purpose of verifying the contents of files on nodes, this gateway command compares a file on the gateway with a specified file on the testbed nodes. To limit communication overhead, the program compares the MD5 sums of the files instead of sending the files themselves. The command prints a sequence of lines, one per responding node. Each line indicates a responding node's number and the result of its MD5 comparison: `Same`, `Different`, or `File not found`.

The optional `-n` parameter lets one specify a comma-separated list of node numbers and/or node-number ranges. The `nodediff` command listens for responses from nodes for 1.5 seconds. With the optional `-t` parameter, one can change this duration. The value following `-t` is a floating-point number representing the time to wait in seconds.

Usage is `nodediff` *localfile remotefile* [`-n` *nodes*][`-t` *seconds*] where

$$
\begin{aligned}
\textit{nodes} &::= \textit{range}(\texttt{,}\textit{range})^{*} \\
\textit{range} &::= \textit{node} \mid \textit{node}\texttt{-}\textit{node} \\
\textit{node} &::= (\texttt{0}|\texttt{1}|\texttt{2}|...|\texttt{9})^{+}
\end{aligned}
$$

4.1.3 noderun

For the purpose of performing arbitrary management tasks, this gateway command runs a remote program or script on one or more testbed nodes. With the `-c` option, the command that runs on the nodes may print text to stdout, and this text will be echoed by `noderun` to stdout on the gateway. One purpose of

using the `-c` option is to retrieve succinct status information from the nodes. Consequently, the format of the `noderun` output appears best when each node's command returns a single line of text. Using the `-c` form of `noderun`, one may add the optional `-n` and `-t` options as described under `nodediff`. Lacking the `-n` option, `noderun` broadcasts the command to all nodes.

Executing `noderun` with the `-s` option, a script on the gateway is transmitted to a node or nodes and then is run there. For each node that runs the script, `noderun` prints "Success" or "Fail" to stdout on the gateway depending on whether the return code of the script is zero or non-zero, respectively. Using the `-s` form of `noderun`, one must include the `-n` option and specify the node or nodes to receive the script. The `-t` option can be added, as described earlier. Including the `-r` option specifies the number of times that the `noderun` command should attempt to establish a connection.

As described below, the implementation uses UDP packets to transmit command results from the node back to the gateway. Consequently, it is possible for a command to be run but its report of success or failure to be lost. In order to allow for regeneration of lost results, one should run only commands that can be re-run harmlessly.

Usage is `noderun -c` *command* [`-n` *nodes*] [`-t` *seconds*]

or `noderun -s` *script* `-n` *nodes* [`-t` *seconds*] [`-r` *retries*] .

4.1.4 nodecron

For the purpose of scheduling experiments, this gateway command uploads a new cron file to one or more testbed nodes. For each node specified, `nodecron` prints "Success" to stdout on the gateway when the cron file is successfully installed and

prints "Fail" otherwise. Command options are identical to those of `noderun -s`.

Usage is `nodecron` *cronfile* `-n` *nodes* [`-t` *seconds*] [`-r` *retries*] .

4.2 Data Collection

4.2.1 nodegs

Pronuncation: *node-G-S*, an abbreviation of "get syslog". A researcher collects logged data from the testbed nodes using the `nodegs` command on a gateway. The `nodegs` command first runs `nodels` to create a list of visible testbed nodes and then copies the log file of each node using `scp`.

To generate a data log, one links one's C program to a library that provides a function called `logInfo()`. The `logInfo()` function works exactly as `printf()` except that it sends its data through `syslog` instead of to `stdout`. The `logInfo()` function formats its output distinctively to allow easy identification within a retrieved system log. The `nodegs` command uses this distinctive format to identify node data within the log.

Usage is `nodegs` .

4.3 Implementation

4.3.1 Client-Server Interaction

The implementation follows the client-server model, where each gateway is a client running `node_client` as commanded, and each node is a server running a background process `node_server`. The `node_client` program is itself run by the commands described in Section 4.1.

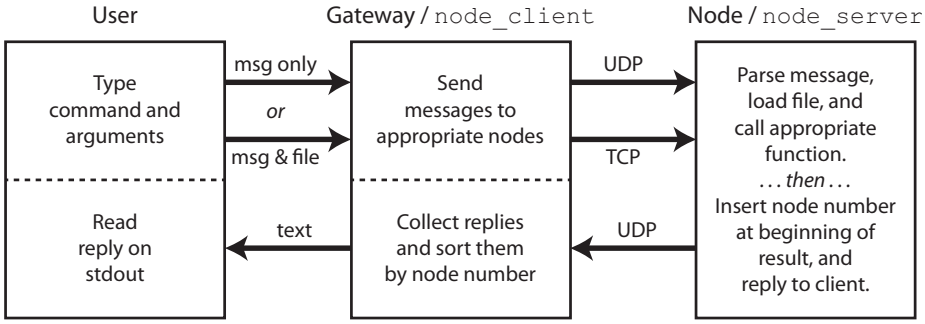


Figure 4.1: The client-server model used by the SCORPION testbed management suite.

When `node_client` is run on a gateway, it communicates with nodes using either TCP or UDP, depending on whether the command that runs it transfers a file. (See Figure 4.1.)

Since `nodels`, `nodediff`, and `noderun -c` are short commands that receive a short text result from each node, the commanded request can be transmittted in a single UDP packet, as can each of the nodes' results. And so `node_client` uses UDP for these cases. However `noderun -s`, `nodecron`, and `nodegs` all transfer files between the client and the node, making UDP inadequate because data of the transferred file may be split into multiple packets, some of which may not arrive error-free or at all. In this case `node_client` uses TCP, which ensures that all packets of a transmitted file will arrive error-free and in-order. The extra time required to establish a TCP connection is a small cost for data integrity.

4.3.2 Client-Server Communication Protocols

`node_client` and `node_server` communicate using a simple request/response protocol. When using UDP, messages are sent as text strings of whitespace-delimited tokens representing a command followed by server arguments. For example, when the gateway executes `nodels -t 5`, which has the *client* argument `-t 5`, it sends

to the node just the message `nodels` (but not the `-t` parameter because `-t` controls the client's retransmission behavior). As a second example, executing the gateway command `noderun -c 'uname -a'` sends the message `noderun -c uname -a` to the node.

When `node_client` and `node_server` communicate using TCP (i.e., when a file is transmitted), the TCP message consists of both length fields and data fields: the first field is one byte long and represents the length of the command string, the next field is three bytes long and encodes the length of the file that is being transmitted, the third field is the command string, and the fourth field is the contents of the file.

Except in the case of `nodegs` (whose "reply" is a syslog file that is obtained using `scp`), a reply from `node_server` is transmitted as a UDP packet that consists of two space-separated fields: the first field is the node's number encoded as a text string, next is the space-character field separator, and the second field is the first line of the result that is returned by the command that was executed.

Chapter 5

Results

This project is an astounding success! Not only is the original prototype node (UCSC bus 7855) still operational, but the node's design, nearly unchanged, has been deployed to all but a couple of the remaining UCSC buses. Buses currently serve route and location data to the project's five base stations.

My production deployment of BTS 2 nodes and base stations supports other parts of the BTS 2 production system which are designed by other students: the production server (by Wade "Simba" Khadder), the web app (by Kevin Abas), the "Slug Route" Android smartphone app (by Wade "Simba" Khadder), and the "Slug Route" iOS smartphone app (by Sterling Dreyer). Firmware for the route programmer is being written by Christopher Villalpando.

5.1 Comparing SCORPION and DOME

When surveying existing wireless network testbeds, DOME [30] is most similar to SCORPION. Although the two systems have similar goals, their selection of different technologies for administrative backchannels necessitates major differences

in system management.

DOME's 3G network provides sufficient bandwidth to distribute configuration images and provides continuous connectivity for node management. Consequently, updates to nodes can be performed at any time, as long as the node is powered up. SCORPION has a much slower 9,600-baud monitoring link for real-time node localization. This link is too slow for image distribution, and so image updates to each SCORPION node must be performed via the node's WiFi link. Although the campus has only a few WiFi access points in buildings close to transit routes, we are fortunate that transit buses must be refueled periodically. Therefore, the BTS project located a WiFi access point in a building near the campus filling station to provide necessary connectivity for updates. A feature of the Linux computers used on the original BTS bus nodes caused them to remain powered for a few minutes after the bus's engine was turned off for fueling, thereby allowing time for image transfers. Future BTS 2 testbed must be designed to have this feature.

Another consequence of managing the testbed over a disconnected network is that researchers must adapt their tasks to the nodes presently nearby. `nodels` helps identify nearby nodes, `nodediff` provides a low-bandwidth means of confirming the contents of files on nodes, `nodecron` lets one schedule experiments quickly by uploading cron files, and `noderun` provides a means of executing Linux commands and scripts to prepare the testbed's nodes for experiments.

Clearly a backchannel network that maintains continuous connectivity would reduce and possibly eliminate the need for management tools as they have been implemented, but using such networks—e.g., 3G—tends to incur large recurring costs. For some research groups these costs are prohibitively high compared to their benefits, and for these groups, a management system more like ours is necessary.

5.2 PCB Design

Since I had designed only one PCB before this project, the experience of designing five new PCBs provided me with a few lessons.

(1) I discovered the benefits of placing surface-mounted devices on just one side of the PCB. The route-selector PCB has surface-mount components on both sides. I can solder the fine-pitched microcontroller on one side using a reflow soldering oven, but then I must hand-solder the remaining "coarse-pitched" surface-mount packages on the other side. The sign-controller PCB, which was designed after learning this lesson, places surface-mount components all on the same side.

(2) I also discovered that the hardcopy plots that I had hand checked should have been printed 1:1 to reveal package/footprint size mismatches. My error was significant enough that I was unable to rework the PCB. I needed to create a second revision of the route-selector PCB before I could solder a working assembly.

(3) I learned the advantage of having all components in-hand during the design process. PCB footprints were checked against measurements of physical components that were obtained using digital calipers. Also, although data sheets have package sizes, it is easy to miss physical obstructions during the design when the packages are not physically present.

All of the PCB designs have performed well. The original prototype, which is mounted in UCSC bus 7855, still functions flawlessly. The only failures discovered in production nodes were caused by failures to hand-solder all through-hole components on a PCB.

5.3 Base-Station Design

The original Mini-ITX BTS base stations occasionally failed. Sometimes a base station needed manual rebooting, and sometimes a hard drive would crash. Aside from a single network-reconfiguration issue in September 2015, the current BTS 2 base stations have been working flawlessly since their installation in February 2015.

The improved quality may be due to any of several changes:

- Completely new hardware (Replace Mini-ITX PC with Raspberry Pi).
- Completely rewritten software (check array bounds in existing C++ daemon and replace interpreted Python program with new daemon written in C++).
- Uninterruptible power supplies were installed on all base stations in November 2015.

5.4 Server Design

The original BTS server, "skynet", was an Apple Mac Pro running GNU Linux. Since this server was located in the E2-311 lab, unreliable UCSC power often brought the server down, and it occasionally needed manual restarting. Generally, the "skynet" server was not production-worthy.

For the BTS 2 prototype, the Mac Pro server remained mostly unchanged because we knew that it would be replaced with a new server that was maintained by UCSC IT. I celebrated when the production apps were cut over from "skynet" to the UCSC IT server because I no longer would need to try and maintain the illusion that "skynet" was a production server.

Appendix A

Design Computations

A.1 MAX6495

Given the harshness of an automotive electrical environments, the Route Sign's over-voltage protection (OVP) circuit is an indispensable part of the system. Although I would have preferred to use a commercial product, none could be found for a 24-V system. My design of the OVP circuit considers guidelines from the MAX6495 data sheet and suggestions from Maxim Application Note 4081. I prototyped the completed design on a modified MAX6495 evaluation kit and verified the threshold of the OVP-circuit. (For reasons of safety and lack of proper equipment, I did not confirm operation at -600 V and $+227$ V.)

The supply pin of the MAX6495 is not connected directly to the UVP circuit. While a MAX6495 itself can tolerate a 72-V input, my power-supply specification requires that the OVP circuit function with a much larger 227-V input. To tolerate the larger voltage, my OVP circuit follows a design idea that is presented in Maxim Application Note 4081 ([20], Figure 2). Instead of powering the MAX6495 directly, this circuit adds a resistor/Zener-diode regulator that limits

the MAX6495's supply voltage to 54 V. In addition, proper selection of voltage-divider resistors safely limits the voltage of the MAX6495's comparator input. These design considerations, among others, are presented in three reference documents: the MAX6495 data sheet, the MAX6495 Evaluation Kit documentation, and Maxim Application Note 4081.

Table A.1 on page 77 and Figure A.1 on page 78 organize the design computations. Note that resistor names in the worksheet are from Figure 2 of Application Note 4081.

V_{OV} =	48	V	**Overvoltage Threshold.** This value is somewhat arbitrary. It is comfortably above the highest voltage of a 24-V electrical system during normal operation while remaining below the MAX6495's regulated 54-V supply (V_Z in the worksheet).
$V_{OV,MAX}$ =	400	V	**Maximum Overvoltage.** An earlier reading of ISO 7637-2 led to this value. Now it appears that 227 V would have been adequate.
M_{OVSET} =	100		From page 9, column 2 of the data sheet.
R1 =	976E+03	Ω	**Sets Threshold.** I enter a commercially available resistor value that is approximately equal to the ideal value computed in the worksheet.
R2 =	26.1E+03	Ω	**Sets Threshold.** I enter a commercially available resistor value that is approximately equal to the ideal value computed in the worksheet.
TOL =	1.0%		**Tolerance of R1 and R2.** Used in worst-case computations.
$I_{D,MAX}$ =	3.0	A	**Worst-case Current.** I sum all of the supply's output powers, divide by 24 V, divide by efficiency, and round up.
$V_{SUPPLY,MIN}$ =	16	V	**Minimum Supply Voltage.** Below this value, the MAX6495 may not power the N-MOSFET.
R_3 =	43E+03	Ω	**Part of the 54-V supply.** I choose a commercially available resistor value $R_3 < R_{3,MAX}$.

Table A.1: Explanation of selected MAX6495 design values. Use with Excel worksheet on the following page.

This design worksheet was created from these documents:

1 Maxim MAX6495-MAX6499 Datasheet
2 Maxim MAX6495 Evaluation Kit documentation
3 Maxim Application Note 4081, Alternate Circuits for Overvoltage Protection: Tips and Tricks

V_{OV} =	48	V	Isolate load when input exceeds this voltage
$V_{OV,MAX}$ =	400	V	Worst-case input voltage
V_{TH} =	1.24	V	From datasheet: OVSET rising threshold
I_{SET} =	50E-09	A	From datasheet, OVSET max input current
M_{OVSET} =	100		Minimum multiple of I_{SET} for resistive divider
$I_{TOT,MIN}$ =	5.0E-06	A	Current in resistive divider, minimum
$R_{TOT,MAX}$ =	9.6E+06	Ω	R1 + R2, maximum
R_{TOT} =	1.00E+06	Ω	Enter actual R1 + R2. Must be less than $R_{TOT,MAX}$.
R1 =	974E+03	Ω	Top resistor of resistive divider.
R2 =	25.8E+03	Ω	Bottom resistor of resistive divider. Must be > 2 kΩ.
R1 =	**976E+03**	**Ω**	**Top resistor of resistive divider. Selected value.**
R2 =	**26.1E+03**	**Ω**	**Bottom resistor of resistive divider. Selected value.**
TOL =	**1.0%**		**Tolerance of R1 and R2**
$V_{OV,ACTUAL,MIN}$ =	46.7	V	Actual over-voltage value. Minimum.
$V_{OV,ACTUAL,MAX}$ =	48.5	V	Actual over-voltage value. Maximum.
$I_{TOT,MAX}$ =	400E-06	A	Worst-case current in resistive divider
$P_{TOT,MAX}$ =	**0.16**	**W**	**Worst-case power dissipation of resistive divider**
$I_{D,MAX}$ =	3.0	A	Maximum current draw through overvoltage-protection circuit
$R_{DS(ON),N}$ =	0.20	Ω	Spec of N-MOSFET.
$P_{NMOSFET,MAX}$ =	**1.80**	**W**	**Worst-case power dissipation of N-MOSFET**
V_{FD} =	1.40	V	Forward voltage drop of reverse-protection diode.
P_D =	**4.20**	**W**	**Worst-case power dissipation of reverse-protection diode**
V_Z =	54.0	V	Zener-diode working voltage
$P_{Z,MAX}$ =	3.0	W	Maximum allowed power in Zener diode
$I_{Z,MAX}$ =	56E-03	A	Maximum current in Zener diode
$R_{3,MIN}$ =	6.2E+03	Ω	Minimum value of resistor above Zener diode
$P_{R3,MAX}$ =	19.2E+00	W	Peak power in resistor above Zener diode
$I_{IN,MAX}$ =	150E-06	A	From datasheet: maximum supply current of MAX6495.
$V_{IN,MIN}$ =	5.5	V	From datasheet: minumum supply voltage of MAX6495.
$V_{SUPPLY,MIN}$ =	16	V	Minimum supply provided
$R_{3,MAX}$ =	60.7E+03	Ω	Maximum value of resistor above Zener diode
R_3 =	**43E+03**	**Ω**	**Resistor above Zener diode**
I_{R3} =	8.0E-03	A	Current in resistor above Zener diode
P_{R3} =	**2.8E+00**	**W**	**Power in resistor above Zener diode**

Figure A.1: MAX 6495 Design Worksheet

A.2 LED-Sign Power Routing

Here I compute the amount of routing required in the PCB layout to ensure uniform LED illumination. The result of this computation, which reveals that routing power using 1-oz copper is insufficient, justifies the power-routing strategy described in Section 2.1.3.

First, through direct observation I determined that a 10% reduction in LED current causes a just noticeable variation in LED brightness. And so I chose to limit the reduction in LED current—and hence the reduction in the current of each LED's series-connected current-limiting resistor—to 10%. Since current-limiting resistors are linear devices, Ohm's law implies that I need to limit the reduction in the voltage across each resistor to 10%. So what is the voltage across a current-limiting resistor normally?

The PCB uses a 3.3-V LED supply. With a worst-case 2.4-V LED forward voltage[4] and 0.44 V across the LED driver[33], the voltage across the resistor is $3.3 - 2.4 - 0.44 = 0.46$ V. Ten percent of this value is 0.046 V; I need to check whether the total voltage drop across both the power and ground traces is less than this value.

The power and ground routes run along the PCB horizontally, connecting directly to the power pins of horizontally-arrayed components. I compute the voltage drop along these traces as a function of trace size measured in "squares" (a measurement unit that is useful when computing trace resistance[43]). These computations are for so-called "1-oz" copper foil, which has a "square resistance" $R_{\square} = 0.5$ mΩ. This means that a square of the foil, independent of its absolute dimensions, has a resistance of 0.5 mΩ across opposite edges.

I start with the assumption that the power and ground traces are fed from the

middle of the board, which minimizes the worst-case voltage drop. Each column of LEDs draws current I_{COL}. Since the voltage drop across the power trace reduces the LED current, I use the 10% current reduction in my computations, for an LED current of 18 mA.

$$\begin{aligned} I_{\mathrm{COL}} &= N_{\mathrm{LED}} \times I_{\mathrm{LED}} \\ &= 13 \times 0.018 \text{ A} \\ &= 0.234 \text{ A} \end{aligned}$$

Connecting pairs of adjacent columns is a power-trace segment with a fixed resistance that depends on the dimensions of the segment, measured in squares, $N_{\square}$.

$$\begin{aligned} R_{\mathrm{COL}} &= R_{\square}\, N_{\square} \\ &= 0.0005\, N_{\square} \end{aligned}$$

The voltage along each power-trace segment drops slightly due to its resistance and the current flowing through it. The current depends on the number of columns that the segment feeds: the farthest segment feeds one column; the segment next to it feeds two, etc. The segment that feeds n columns has a voltage drop estimated by the equation below. I am aware that the power-trace segment that is closest to the power source is shorter than the inter-column segments, but for simplicity in later computations I use the same length for all segments.

$$\begin{aligned} V_{\mathrm{DROP,n}} &= n\, I_{\mathrm{COL}}\, R_{\mathrm{COL}} \\ &= 0.234 \times 0.0005\, n\, N_{\square} \end{aligned}$$

$$= \quad 0.000117\, n\, N_{\square}$$

Summing the voltage drops along all of the segments

$$\begin{aligned} V_{\text{TOTAL}} &= \sum_{n=1}^{16} V_{\text{DROP,n}} \\ &= \sum_{n=1}^{16} 0.000117\, n\, N_{\square} \\ &= 0.000117\, N_{\square} \sum_{n=1}^{16} n \\ &= 0.000117\, \frac{16(16+1)}{2}\, N_{\square} \\ &= 0.0159\, N_{\square} \end{aligned}$$

Now I can use this equation to compute the voltage drop along the power trace and the ground trace. Using the actual PCB layout, the power trace is divided into segments with dimension $1.66\,\square$. The same layout shows that the ground trace segments have dimension $2.63\,\square$. So the total drop is across a series connection of these two traces, or $4.29\,\square$:

$$\begin{aligned} V_{\text{POWER}} + V_{\text{GND}} &= 0.01768\, N_{\square} \\ &= 0.0159 \times 4.29 \\ &= 0.068 \text{ V} \end{aligned}$$

This computed voltage drop of 0.068 V is more than the upper limit of 0.046 V that I had determined must be met to preserve uniform LED brightness, and so I cannot route power and ground traces solely through 1-oz copper foil as described.

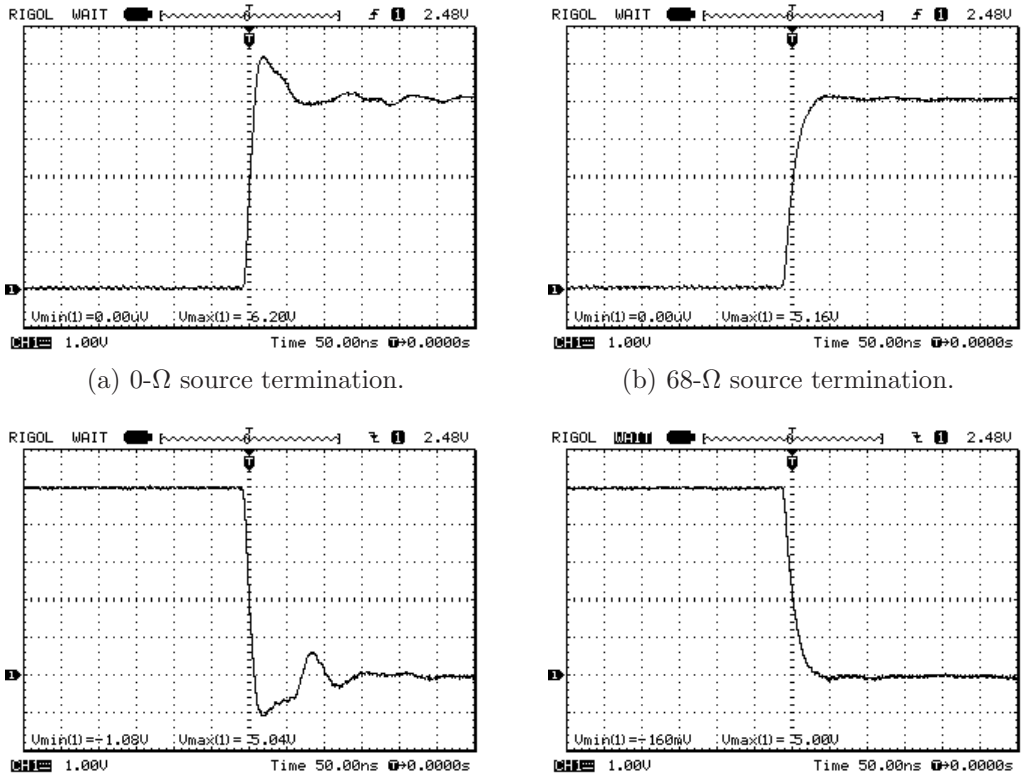
A.3 Ribbon-cable Termination Resistors

Each ribbon cable wire is driven through a source-termination resistor. I can estimate the required value of each termination resistor by comparing the impedance of a ribbon cable wire to the impedance of its driver.

Ribbon-cable wires have an impedance of approximately 100 Ω. Level-shifting drivers have an impedance of approximately 22 Ω to 33 Ω (using data sheet values to compute V_{OH}/I_{OH} and V_{OL}/I_{OL}). These figures lead us to estimate that a terminator should have a resistance of approximately 67 Ω to 78 Ω.

To settle on a single value, I performed a bench test using an actual driver, a 68-Ω resistor, a ribbon cable, and one of the system's LED boards. Subfigures (a) and (c) of Figure A.2 show the waveform at the far end of the ribbon cable (at the LED driver's input) when the level shifter drives the ribbon cable directly. I see that without source termination, the signal at the LED driver's input shows unacceptable overshoot and undershoot of 1.20-V and 1.08-V respectively, magnitudes that far exceed the 0.7-V limit. Subfigures (b) and (d) show that adding 68-Ω source termination eliminates the overshoot and undershoot while maintaining fast edges.

With these observations, I chose to use 68-Ω source terminators on every ribbon-cable signal wire. (Ribbon-cable ground wires are connected directly to ground.) In addition, since each level shifter usually drives *two* ribbon-cable wires (one for each LED PCB), each ribbon-cable signal wire is driven through its own termination resistor (Figure A.3). The alternative arrangement of driving two parallel-connected ribbon-cable wires through a single 68 Ω resistor must be avoided: with little resistance between the far ends of the two connected cables, the level shifter would induce oscillations due to reflections at the cable ends.



(a) 0-Ω source termination.

(b) 68-Ω source termination.

(c) 0-Ω source termination.

(d) 68-Ω source termination.

Figure A.2: Effect of source termination on ribbon-cable signal integrity. Waveforms show the voltage on a CMOS device pin that is driven by a ribbon cable both with and without source termination.

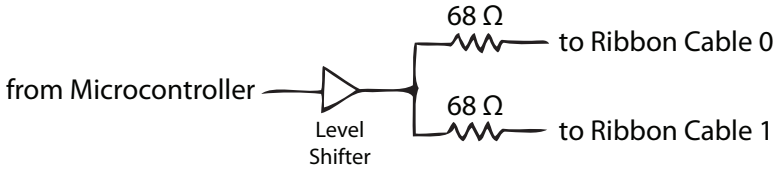


Figure A.3: Termination for ribbon cables.

Supplemental Files

Number	Description	File Format
1	Computations: MAX6495	Excel file
2	Source Code: Route Sign	Source-Code Directory
3	Source Code: Route Selector	Source-Code Directory
4	PCB: Control	Directory of PDFs
5	PCB: Filler	Directory of PDFs
6	PCB Raspberry Pi Serial Port	Directory of PDFs
7	PCB: Route Selector	Directory of PDFs
8	Source Code: Base Station	Source-Code Directory
9	Bill of Materials	Excel file
10	PCB: LED Display Board	Directory of PDFs

Bibliography

[1] AeroComm, Inc. CL4490-1000 Industrial 900-MHz ConnexLink User's Manual, Version 1.4, 2005.

[2] Thomas Anderson and Michael K. Reiter. Geni: Global environment for network innovations distributed services working group, 2006.

[3] Atmel Corporation. Datasheet. 32-bit Atmel AVR Microcontroller. `http://www.atmel.com/Images/doc32059.pdf`, 2012. [Online; accessed July 4, 2012].

[4] Avago Technologies. Datasheet. HLMP-EL55/EG55/EL57/EH57/ED57 T-1 3/4 (5 mm) Precision Optical Performance AlInGaP LED Lamps. `http://www.avagotech.com/docs/AV02-1541EN`, 2009. [Online; accessed July 19, 2012].

[5] Bay Area Circuits, Inc. website. `http://bayareacircuits.com`.

[6] Peter Brissette. personal communication, 2012.

[7] S. Bromage, C. Engstrom, J. Koshimoto, M. Bromage, S. Dabideen, M. Hu, R. Menchaca-Mendez, D. Nguyen, B. Nunes, V. Petkov, D. Sampath, H. Taylor, M. Veyseh, J. J. Garcia-Luna-Aceves, K. Obraczka, H. Sadjadpour, and

B. Smith. SCORPION: a heterogeneous wireless networking testbed. *SIGMOBILE Mob. Comput. Commun. Rev.*, 13(1):65–68, June 2009.

[8] J. Burgess, B. Gallagher, D. Jensen, and B.N. Levine. MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks. In *Proceedings of the 25th IEEE International Conference on Computer Communications, INFOCOM 2006*, 2006.

[9] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag, New York, NY, USA, second edition, 2004.

[10] International Organization for Standardization. *ISO 7637-2 Road Vehicles, Electrical Disturbances from Conduction and Coupling: Part 2, Electrical transient conduction along supply lines only*. ISO, 2004.

[11] Freescale Semiconductor, Inc. Datasheet. MMA7361L: ±1.5g, ±6g Three Axis Low-g Micromachined Accelerometer. `http://www.freescale.com/files/sensors/doc/data_sheet/MMA7361L.pdf`, 2008. [Online; accessed July 5, 2012].

[12] Freescale Semiconductor, Inc. Datasheet. MAG3110: 3-Axis, Digital Magnetometer. `http://www.freescale.com/files/sensors/doc/data_sheet/MAG3110.pdf`, 2012. [Online; accessed July 5, 2012].

[13] Global positioning system standard positioning service performance standard, 4th ed. `http://www.gps.gov/technical/ps/2008-SPS-performance-standard.pdf`, September 2008. [Online; accessed July 16, 2012].

[14] Hammond Manufacturing. Product drawing. 1455J1202BK Enclosure. http://www.hammondmfg.com/pdf/1455J1202.pdf, 2012. [Online; accessed July 25, 2012].

[15] H.W. Johnson and M. Graham. *High Speed Digital Design: A Handbook Of Black Magic*. Prentice Hall, 1993.

[16] James Koshimoto, Matt Bromage, Vladislav Petkov, and Katia Obraczka. Slugtransit: a location-based public transportation management system. In *Proceedings of the 6th International Conference on Mobile Technology, Application & Systems*, Mobility '09, pages 34:1–34:4, New York, NY, USA, 2009. ACM.

[17] Phillip A. Laplante. *Real-Time System Design and Analysis*. John Wiley & Sons, 2004.

[18] Lineage Power Corporation. Datasheet. EHHD006A0B Series (Eighth-Brick) DC-DC Converter Power Modules, 18–75Vdc Input; 12V/6Adc Output. http://www.lineagepower.com/oem/pdf/EHHD006A0B.pdf, 2011. [Online; accessed July 8, 2012].

[19] Lineage Power Corporation. Datasheet. EHHD020A0F Series (Eighth-Brick) DC-DC Converter Power Modules, 18–75Vdc Input; 3.3V/20Adc Output. http://www.lineagepower.com/oem/pdf/EHHD020A0F.pdf, 2011. [Online; accessed July 8, 2012].

[20] Maxim Integrated Products. Application Note 4081. Alternate Circuits for Overvoltage Protection: Tips and Tricks. http://pdfserv.maxim-ic.com/en/an/AN4081.pdf, 2007. [Online; accessed July 4, 2012].

[21] Maxim Integrated Products. Datasheet. 72V, Overvoltage-Protection Switches/Limiter Controllers with an External MOSFET. http://datasheets.maxim-ic.com/en/ds/MAX6495-MAX6499.pdf, 2012. [Online; accessed July 4, 2012].

[22] Elwyn McLachlan. Collecting quality gps data in a canopy environment. In *22nd Annual Esri International User Conference*, Redlands, CA, 2002. ESRI.

[23] The Network Simulator ns-2. http://www.isi.edu/nsnam/ns.

[24] ns-3. http://www.nsnam.org.

[25] PlanetLab Consortium. https://www.planet-lab.org.

[26] SCALABLE Network Technologies. http://www.scalable-networks.com.

[27] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols. In *WCNC'05*, 2005.

[28] Kenneth J. Schmier and Paul Freda. Public transit vehicle arrival information system. US Patent 6,374,176, April 2002. Assigned to Nextbus Information Systems, Inc.

[29] SiRF Technology, Inc. NMEA Reference Manual. http://www.sparkfun.com/datasheets/GPS/NMEA%20Reference%20Manual-Rev2.1-Dec07.pdf, 2007. [Online; accessed July 4, 2012].

[30] H. Soroush, N. Banerjee, A. Balasubramanian, M.D. Corner, B.N. Levine, and B. Lynn. DOME: a diverse outdoor mobile testbed. In *Proceedings of*

the 1st ACM International Workshop on Hot Topics of Planet-Scale Mobility Measurements (HotPlanet '09), New York, NY, USA, 2009. ACM.

[31] H. Soroush, N. Banerjee, M.D. Corner, B.N. Levine, and B. Lynn. DOME: a diverse outdoor mobile testbed. Technical Report UM-CS-2009-23, Dept. of Computer Science, Univ. of Massachusetts Amherst, 2009.

[32] Texas Instruments Incorporated. Datasheet. SN74LVC4245A Octal Bus Transceiver and 3.3-V to 5-V Shifter with 3-state Outputs. `http://www.ti.com/lit/ds/symlink/sn74lvc4245a.pdf`, 2005. [Online; accessed July 4, 2012].

[33] Texas Instruments Incorporated. Datasheet. CD54/74AC563, CD54/74AC-573, CD54/74ACT563, CD54/74ACT573 Octal Transparent Latch, 3-State. `http://www.ti.com/lit/ds/symlink/cd74ac573.pdf`, 2012. [Online; accessed July 4, 2012].

[34] UBM Electronics Embedded.com & EE Times. 2012 embedded market survey, 2012.

[35] TransLoc, Inc. TransLoc Product Overview. `http://transloc.com/products`, 2012. [Online; accessed July 4, 2012].

[36] Rachel Warren. iPhone app will add Campus Transit. *The Daily Reveille*, September 28, 2010. `http://www.lsureveille.com/news/iphone-app-will-add-campus-transit-1.2348971` [Online; accessed July 9, 2012].

[37] Washington Metropolitan Area Transit Authority. Metro – Bus – NextBus Frequently Asked Questions. `http://www.wmata.com/rider_tools/nextbus/faq.cfm`, 2012. [Online; accessed July 4, 2012].

[38] WebTech Wireless, Inc. How NextBus Works. http://news.nextbus.com/how-nextbus-works-2, 2012. [Online; accessed July 4, 2012].

[39] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: a wireless sensor network testbed. In *Proceedings of the 4th international symposium on Information processing in sensor networks (IPSN '05)*, Piscataway, NJ, USA, 2005. IEEE Press.

[40] Wikipedia. Flip-disc display — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Flip-disc_display, 2012. [Online; accessed July 8, 2012].

[41] Wikipedia. LSU Tiger Trails) — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/LSU_Tiger_Trails, 2012. [Online; accessed July 9, 2012].

[42] Wikipedia. Metrobus (Washington, D.C.) — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Metrobus_(Washington,_D.C.), 2012. [Online; accessed July 4, 2012].

[43] Wikipedia. Sheet resistance — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Sheet_resistance, 2012. [Online; accessed July 21, 2012].

[44] Peter Wilson. *The Circuit Designer's Companion*. Elsevier Science, 3rd edition, 2012.