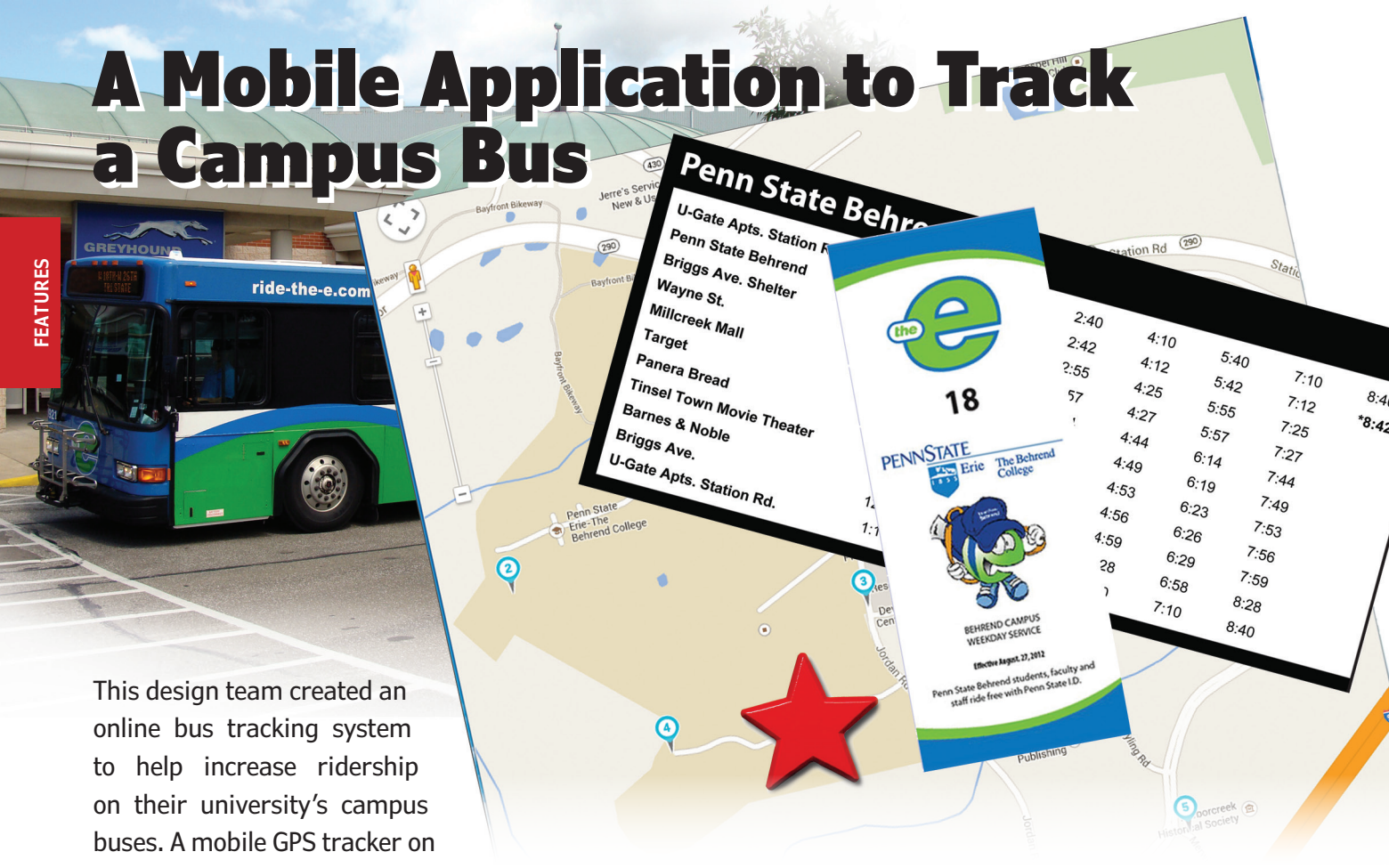
FEATURES

A Mobile Application to Track a Campus Bus



This design team created an online bus tracking system to help increase ridership on their university's campus buses. A mobile GPS tracker on the bus communicates over a radio link to a base station. The system reliably and accurately predicts the bus location even when the vehicle is out of radio contact range.

By Chris Coulston, Daniel Hankewycz, and Austin Kelleher (US)

The continued expansion of the Penn State Erie campus in Erie, PA, has been accompanied by a shuttle bus service provided by the Erie Metro Transit Authority (EMTA). The shuttle services eight stops around our campus on a loop that covers approximately 3 miles in 20 min. Riders can request stops anywhere on the loop and the bus picks up riders who flag it down between stops. Unfortunately, ridership on the bus is low, partly because of the bus' unpredictable schedule. Riders want to know where the bus is and when it will arrive at a stop.

In addition to the environmental benefits of riding the bus, increased ridership would decrease the growing traffic problems encountered between classes. To address low ridership, a team of engineering students and faculty constructed an automated vehicle locator (AVL) to track our campus shuttle and provide accurate estimates of when the shuttle will arrive at each stop.

THE BIG PICTURE

Before we dive into the bus tracker's details, we'll provide a broad overview of the hardware and software used in this project. If you don't entirely understand how the system works after reading this section, don't worry, that's what the rest of the article is for! **Figure 1** shows the bus tracker's hardware, which consists of three components: the user's smartphone, the base station placed at a fixed location on campus, and the mobile tracker that rides around on the bus.

Early on, we decided against a cellular-based solution. Instead we placed a code division multiple access (CDMA) modem (think cell phone) on the bus as the mobile tracker. While this concept benefits from wide-ranging cellular coverage, it incurs monthly cellar network access fees. The concept presented in **Figure 1** requires the design to cope with a 900-MHz radios' limited range.

Figure 2 shows the software architecture running on the hardware from **Figure 1**. When the user's smartphone loads the bus tracker webpage, the JavaScript on the page instructs the user's web browser to use the Google Maps JavaScript API to load the campus map. The smartphone also makes an XMLHttpRequests request for a file on the server (stamp.txt) containing the bus' current location and breadcrumb index (more on this later).

This information along with information about the bus stops is used to position the bus icon on the map, determine the bus' next stop, and predict the bus' arrival time at each of the seven bus stops. The bus' location contained in stamp.txt is generated by a GPS (EM-408) in the form of an NMEA string. This string is sent to a microcontroller and then parsed. When the microcontroller receives a request for the bus' location, it formats a message and sends it over the 900-MHz radio link. The base station compares the bus position against a canonical tour of campus (breadcrumb) and writes the best match to

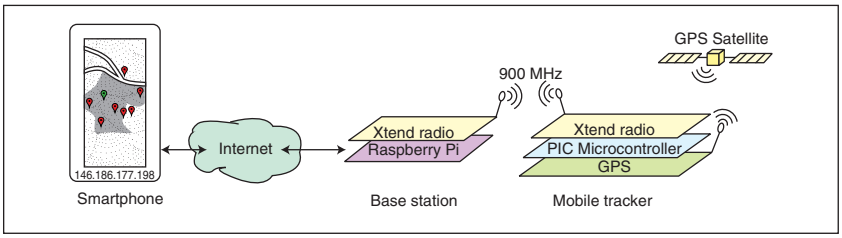


FIGURE 1
The bus tracking system includes a user's smartphone, the base station, and the mobile tracker.

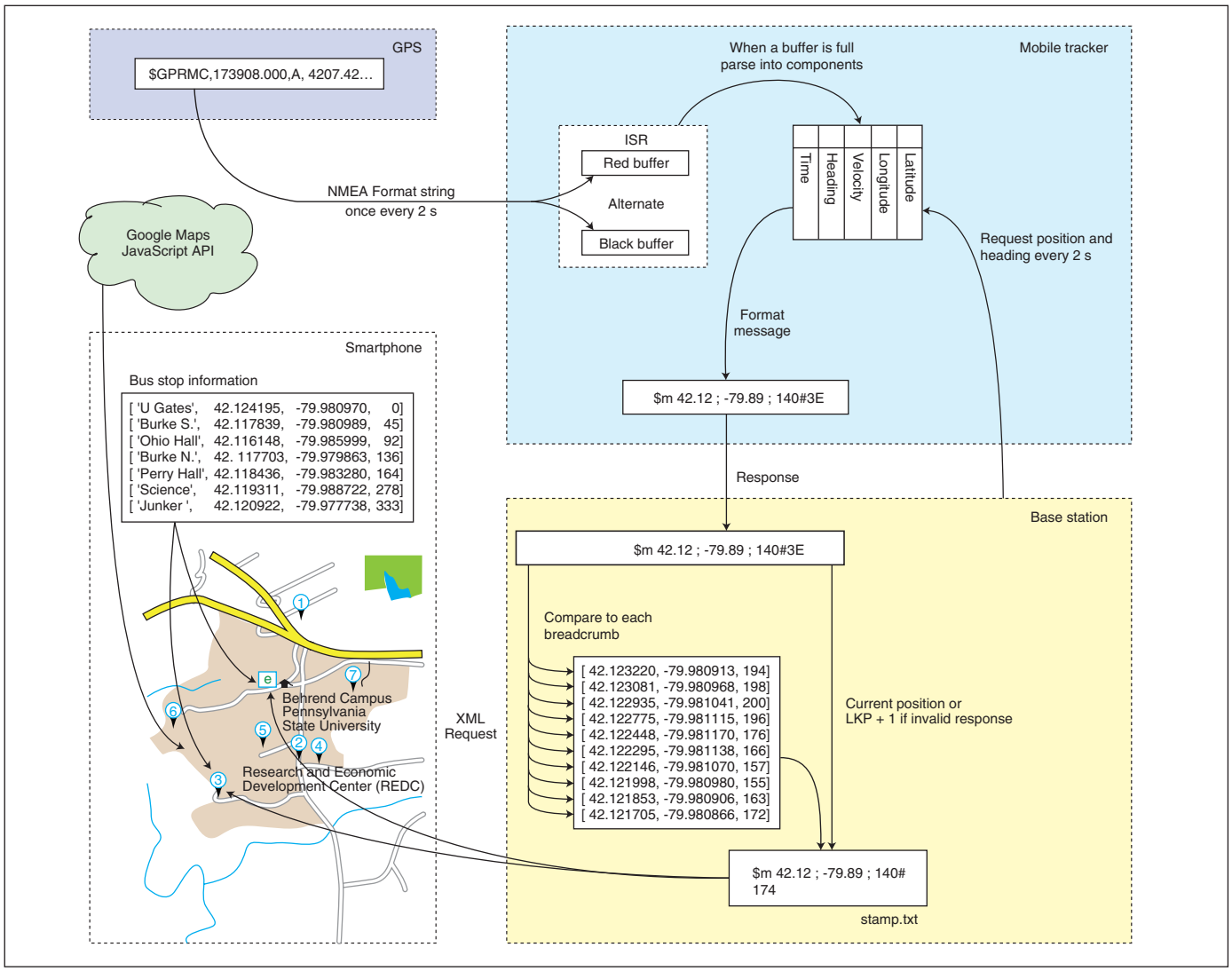


FIGURE 2
The bus tracker's software architecture includes a GPS, the mobile tracker, a smartphone, and the base station.

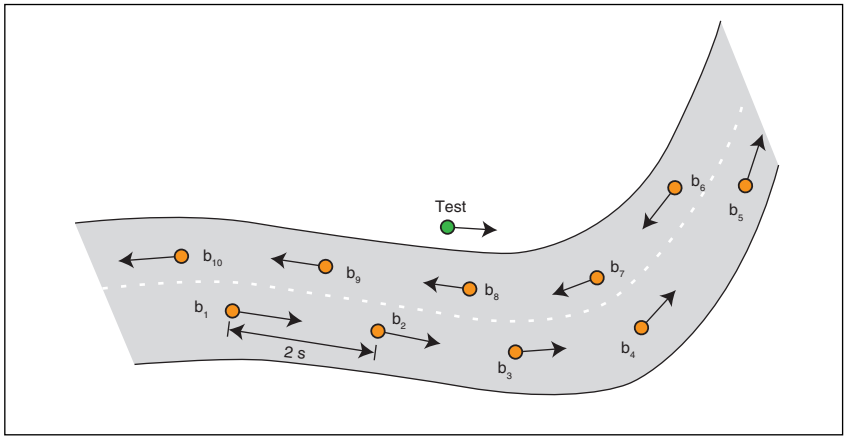


FIGURE 3
Breadcrumbs (b_1 – b_{10}) containing the bus' position and orientation information were taken every 2 s during a test-run campus tour.

stamp.txt.

Early in the project development, we decided to collect the bus' position and heading information at 2-s intervals during the bus' campus tour. This collection of strings is called "breadcrumbs" because, like the breadcrumbs dropped by Hansel and Gretel in the eponymously named story, we hope they will help us find our way around campus. **Figure 3** shows a set of breadcrumbs (b_1 – b_{10}), which were collected as the bus travelled out-and-back along the same road.

The decision to collect breadcrumbs proved fortuitous as they serve an important role in each of the three hardware components shown in **Figure 1**. Now that you have the big picture, we'll dive into the design decisions that converted this concept into a robust bus-tracking system.

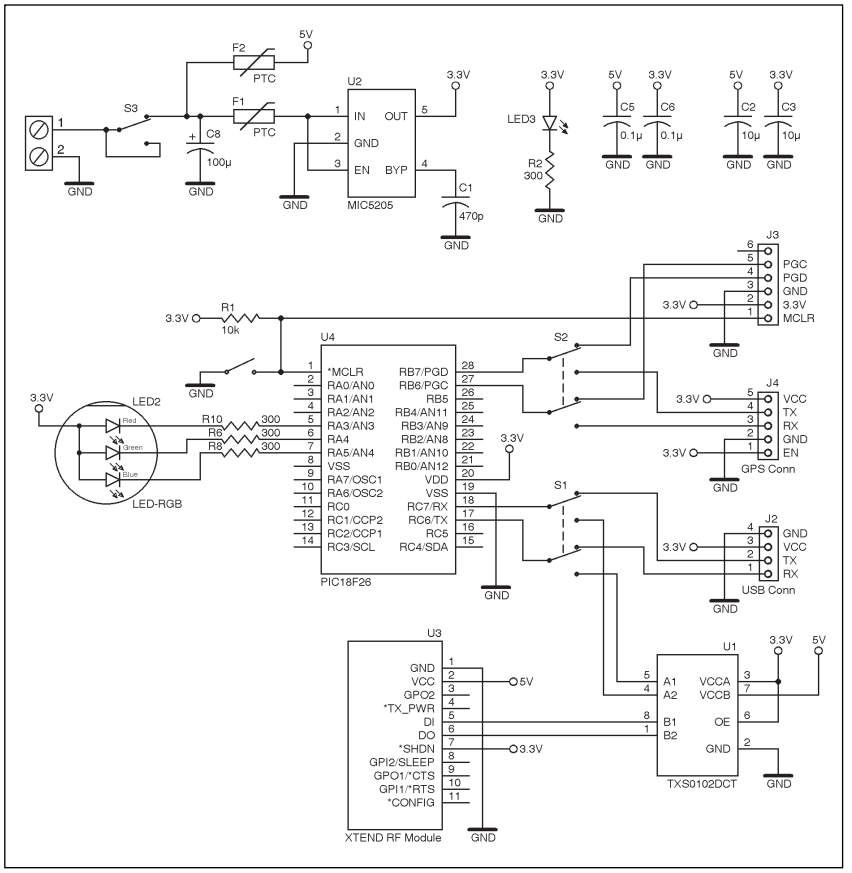


FIGURE 4
The mobile tracker includes a Microchip Technology PIC18F26K22 microcontroller, a Micrel MIC5205 regulator, a Digi International XTend RF module, and a Texas Instruments TXS0102 bidirectional translator.

MOBILE TRACKER

The bus houses the mobile tracker. **Figure 4** shows the schematic, which is deceptively simple. What you see is the third iteration of the mobile tracker hardware.

In terms of hardware, the best decision we made was to abandon the idea of trying to integrate a 12-to-5-V converter onto the mobile tracker PCB. Instead we purchased a $40 CUI VYB15W-T DC-DC converter and fed the mobile tracker 5-V inputs.

Using an off-the-shelf isolated power supply reassured the EMTA that our circuit would not damage their bus and helped convince us that power spikes on the bus' mains power would not damage our circuit. A pair of PTC thermal fuses between the power supply and circuitry provided additional safeguarding.

We used Micrel's MIC5205 regulator to step down the 5 V for the 3.3-V GPS, which easily supplied its peak 80 mA. Since we ran the radio at 5 V for the best range, we ended up with mixed voltage signals. We used a Texas Instruments TXS0102 bidirectional voltage-level translator, which handles voltage-interfacing duties between the 5-V microcontroller and the 3.3-V GPS module.

We selected Microchip Technology's PIC18F26K22 because it has two hardware serial ports, enabling it to simultaneously communicate with the GPS module and the radio modem when the bus is traveling around campus. We placed two switches in front of the serial ports. One switch toggles between the GPS module and the Microchip Technology PICkit 3 programming pins, which are necessary to program the microcontroller. The second switch toggles between the radio and a header connected to a PC serial port (via a Future Technology Devices FT232 USB-to-serial bridge). This is useful when debugging from your desk. An RGB LED in a compact PLCC4 package provides state information about the mobile tracker.

Digi International's XTend RF modules are the big brothers to its popular XBee series. These radios come with an impressive 1 W of transmitting power over a 900-MHz frequency enabling ranges up to a mile in our heavily wooded campus environment. The radios use a standard serial interface requiring three connections: TX, RX, and ground. They are simple to set up. You just drop them into the Command mode, set the module's source and destination addresses, store this configuration in flash memory, and exit. You never have to

```
----------------- System Status --------
 Tour  0
 GPS src  USART2
 Sim type  clean
 Output  USART1
----------------- WDT configuration --------
 WDT:  Enabled
 WDT pre:  1:4096
----------------- GPS Status -----------
 Time  133020.000
 Date  101013
 Lat:Lon   42.118458:-079.982588
 Velocity  13
 Heading:  62
 GPS signal:Locked
----------------- Query GPS ------------
?: help  s: gps String t: Time
p: Position  l: Latitudeg: lonGitude
v: Velocity  m: map  h: Heading
i: GPS sIgnal status d: Date
--------------- Change system mode ------
o: tOggle source of NMEA strings between simulation and USART2
c: Change USART used by printf
k: Kick simuation ahead 30 seconds
f: conFigure GPS to 9600 baud RMC
F: configure GPS to Factory defaults
R: Reboot the system
```

LISTING 1
The code shows the mobile tracker's executive function menu.

```
xbee = serial.Serial( port=/dev/ttyAMA0, baudrate=9600,    xonxoff=False,
     dsrdtr=False,   writeTimeout=0,
                     timeout=0.5,    stopbits=serial.STOPBITS_ONE,
     rtscts=False,   parity=serial.PARITY_NONE)

while True:
    xbee.open()    # open the serial port
    fcntl.lockf(xbee,fcntl.LOCK_EX) # blocking lock for exclusive access
    xbee.flush()    # remove any remnant junk
    xbee.write('m')            # query mobile position
    inLine = xbee.readline()   # read response and then
    fcntl.lockf(xbee,fcntl.LOCK_UN) # unlock the port for another user
    xbee.close()               # and then close the port

    minIndex = closestBreadCrumb(inLine) # index of closest breadcrumb
    f = open("/var/www/remote/stamp.txt", "w")
    if (minIndex != -1):    # error code = -1
f.write(inLine[0:string.find(inLine,'#')+1]+'\n' )
f.write(str(minIndex) + '\n')
lkpIndex = minIndex          # refresh Last Know Position
    else:
lkpIndex += 1;
f.write("$m " + str(breadCrumb[lkpIndex][1]))
f.write(" ; " + str(breadCrumb[lkpIndex][2]))
f.write(" ; " + str(breadCrumb[lkpIndex][3]) + "#\n")
f.write(str(lkpIndex)+'\n')
    time.sleep(2.0)
```

LISTING 2
Python code runs on the base station to query the mobile tracker.

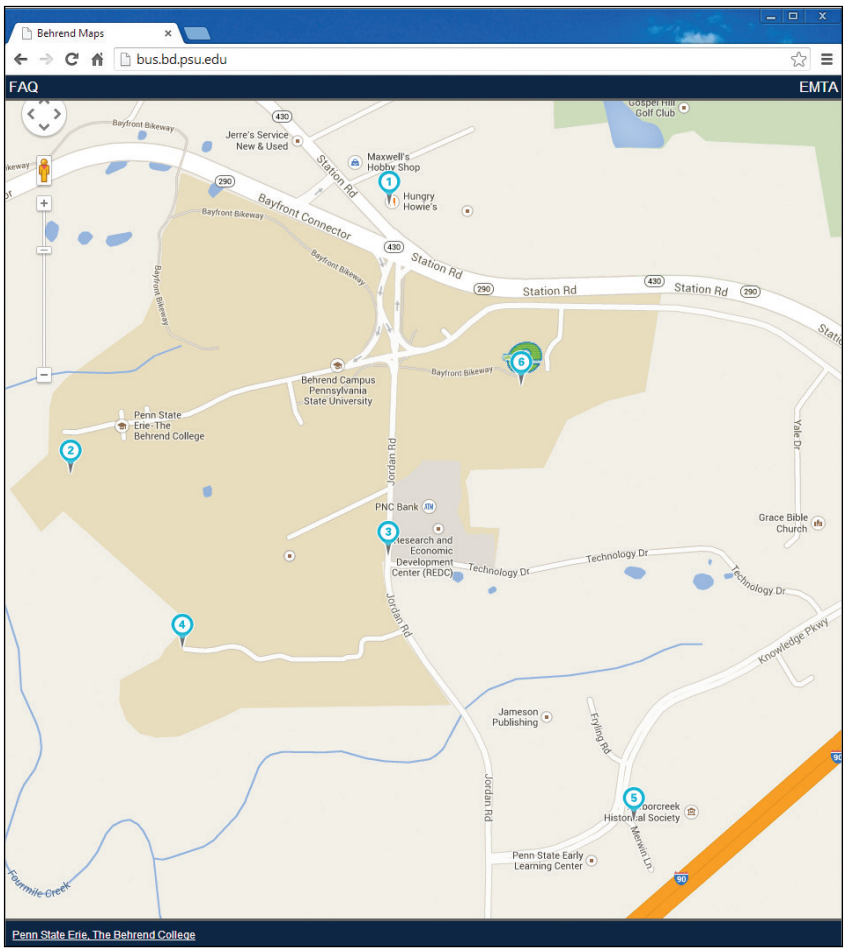


PHOTO 1
This is the bus tracker web application. Numbers are bus stops and the "e" is the bus' location.

deal with them again. Any character sent to the radio appears on the destination modem's RX line.

The GPS receiver utilizes the CSR SiRFstarIII chip set, which is configured to output a recommended minimum specific (RMC) string every 2 s. An RMC string consists of several comma-delimited fields:

```
$GPRMC,173908.000,A,4207.4253,N,07
958.8668,W,5.23,140.26,290513,,,*73
```

From left to right, they are the header identifying the type of string, the time (24-h format), the lock status (A = valid), the latitude, the N/S indicator, the longitude, the east/west indicator, the speed (knots), the heading, the date, and an XOR-based checksum.

Take care when interpreting the latitude and longitude as they are in a DDMM.MMMM (degree, minute, second) format where D digits are degrees and the M digits are minutes. The Google Maps API expected latitude and longitude in DD.DDDDDD (decimal degree) format. The solution is to multiply the minute's value by 100/60. Dividing it by 60 scales it to [1-0] and then multiplying it by 100 puts it in base 10 notation. The checksum is just the bitwise XOR of the characters following the `$` and preceding the *.

The mobile tracker's firmware listens for commands over the serial port and generates appropriate replies. Commands are issued by the developer (e.g., `?`, which generates the help menu shown in **Listing 1**) or by the base station. The most common command issued by the base station is `m`, which generates a semicolon-delimited string consisting of the bus' latitude, longitude, and heading, followed by a checksum (e.g., `$m 42.121161 ; -079.982918 ; 256#4e`).

Burning breadcrumbs into the mobile tracker's flash memory proved to be a good design decision. With this capability, the mobile tracker can generate a simulated tour of campus while sitting in the lab bench.

BASE STATION

The base station consists of an XTend RF module connected to a Raspberry Pi's serial port. The software running on the Raspberry Pi is responsible for everything from running a Nginx open-source web server to making requests for data from the mobile tracker. From **Figure 1**, the only additional hardware associated with the base station is the 900-MHz XTend radio connected to the Raspberry Pi over a dedicated serial port on pins 8 (TX) and 10 (RX) of the Raspberry Pi's GPIO header.

The only code that runs on the base station is the Python program, which periodically uses the mobile tracker to request the bus' position and heading (see **Listing 2**). The program starts by configuring the serial port in the common 9600,8,N,1 mode. Next, the program is put into an infinite loop to query the mobile tracker's position every 2 s.

The serial port is opened with exclusive access to enable different processes to issue commands to the mobile tracker (through the serial port) without the responses becoming garbled with one another. This situation occurs frequently during development any time diagnostics need to be performed on the mobile tracker while the code in **Listing 2** is executing.

The `closestBreadCrumb(inLine)` function in **Listing 2** determines how far along the bus is in its route using the breadcrumb array from **Figure 3**. The input to this function is the current latitude, the longitude, and the bus' heading ("test" in **Figure 3**). It returns the breadcrumb's with the minimum distance to the bus using:

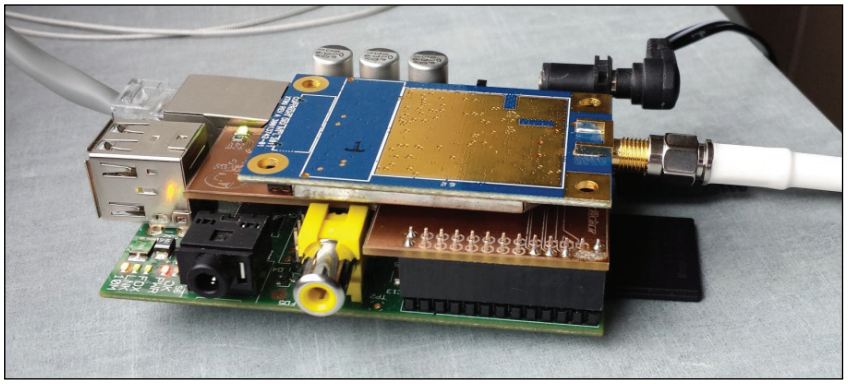
$$\text{dist} = |\Delta\text{head}| + \sqrt{(111{,}320 \times \Delta\text{lat})^2 + (74{,}630 \times \Delta\text{lng})^2}$$

Regardless of where you are on earth,

```
/**
* This function reads the contents of the stamp.txt file
* @return a string containing latitude, longitude, heading
*/
function readGPS() {

    var xmlhttp;
    if (window.XMLHttpRequest) {
  xmlhttp=new XMLHttpRequest();
    } else {
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
    }

    xmlhttp.onreadystatechange=function() {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
     gpsString=xmlhttp.responseText;
    }

    xmlhttp.open("GET","./remote/stamp.txt",true);
    xmlhttp.send();
}

/**
* This function updates the position of the bus and information available to users
* @param refreshBusStops Boolean directing function to update arrival times at bus stops
*/
function updateMapMarkers(refreshBusStops){

    var cords = new Array;

    readGPS();                  // "returns" global gpsString
    setTimeout(function(){      // Since it takes a while to get a response
  cords = parsePosition(gpsString);
  bus.currLong = parseFloat(cords.pop());
  bus.currLat = parseFloat(cords.pop());
  bus.currHead = parseInt(cords.pop());
  bus.breadIndex = parseInt(cords.pop());
  if(refreshBusStops == "TRUE") refreshBusStopMarkers();
  refreshBusPosition();
    }, 300);
} // end
```

LISTING 3
This Java code reads the GPS and sets the bus stop markers' positions.

PHOTO 2
The mobile tracker's base station includes a Raspberry Pi, an interface board, and a radio modem.



a decimal degree of latitude (north/south) is always 111,320 m. This is not so with longitude. At our campus' longitude, which is 42.12° north, a decimal degree is sin(42.12) × 111,320 = 74,630 m. The square root term in the equation converts the difference between a breadcrumb's position and bus' position into a distance in meters. The heading is included in this calculation because without it, we would not be able to tell if the bus was traveling up the road toward the residence hall, or down the same road toward the engineering building. For example, in **Figure 3** the point

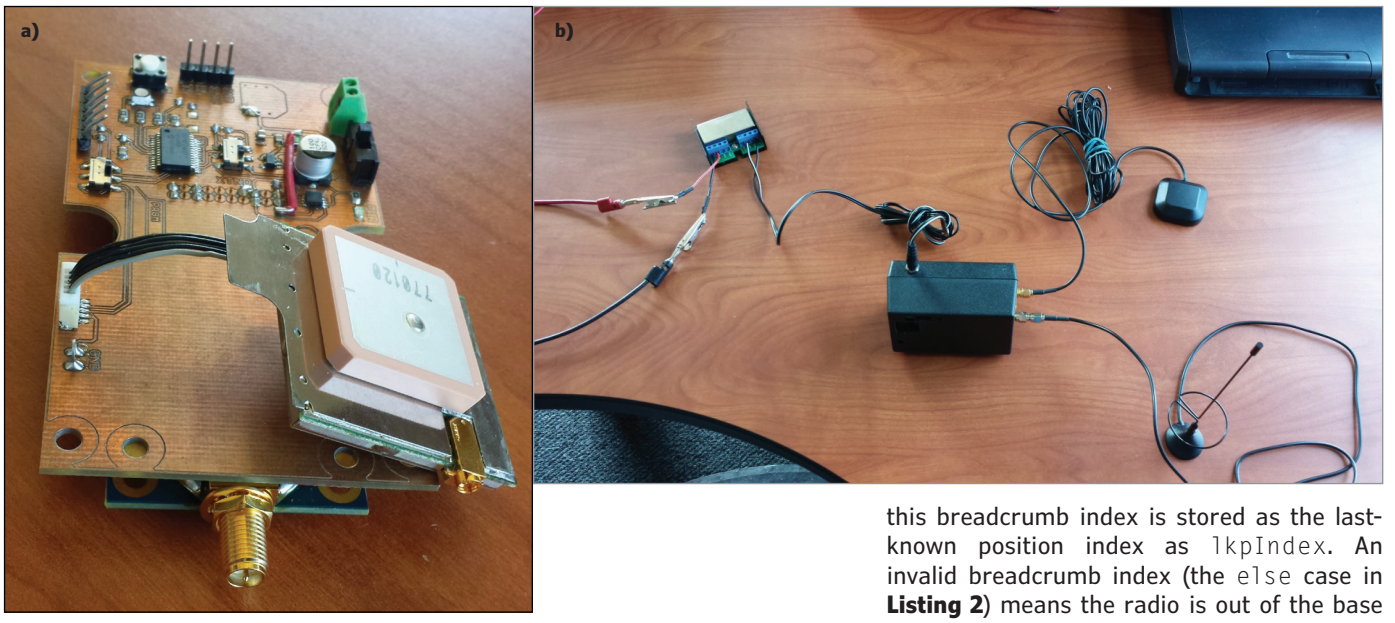


PHOTO 3
The mobile unit **(a)** resides in the black case, ready for installation in the bus **(b)**.

labeled "test" is geographically closest to breadcrumb b8. However, including the heading it shows that it really is closest to b2.

If the `closestBreadCrumb(inLine)` function returns a valid breadcrumb index (the "if" case in **Listing 2**), the bus' location and heading along with the breadcrumb index are written to the file stamp.txt. In addition, this breadcrumb index is stored as the last-known position index as `lkpIndex`. An invalid breadcrumb index (the `else` case in **Listing 2**) means the radio is out of the base station's range.

When this happens, the last-known position index is incremented and the breadcrumb at that index is stored in stamp.txt, which creates an inferred position. This inferred bus position is updated every 2 s until the mobile tracker comes back in range of the base station, at which point the code in **Listing 2** will resume storing the mobile tracker's actual position in stamp.txt.

This technique of fusing an inferred position when radio contact is lost is beneficial to users. The user is assured that the system is properly functioning when he sees the bus moving normally on its route.

The drawback of this technique is that the bus position may "glitch" when transitioning from the inferred position back to its actual position. In practice, this technique works exceptionally well, surpassing our highest expectations. This is a good thing as the mobile tracker is out of radio contact with the base station during about 25% of its route.

SMARTPHONE

Photo 1 shows the user interface presented by the Bus Locator web page. The Raspberry Pi's Nginx server delivers the user index.php, which directs the browser to load a campus map via the Google Maps JavaScript API and JavaScript code, which overlays the bus and bus stops.

Listing 3 shows two functions, `readGPS` and `updateMapMarkers`, which overlay the bus and bus stop icons on the campus map. The `readGPS` function uses Asynchonous JavaScript (AJAX) and XML to read the contents of the stamp.txt file generated by **Listing 2**. After creating a request object (`xmlhttp`) and sending a request to read stamp.txt, things get a little weird. The main

PROJECT FILES

circuitcellar.com/ccmaterials

RESOURCES

Nginx, http://nginx.org

PySerial, http://pyserial.sourceforge.net.

Raspberry Pi, www.raspberrypi.org.

W3Schools, www.w3schools.com.

SOURCES

SiRFstarIII Chip set
CSR plc | www.csr.com

VYB15W-T DC-DC Converter
CUI, Inc. | www.cui.com

XTend and XBee RF modules
Digi International, Inc. | www.digi.com

LM7805 Linear regulator
Fairchild Semiconductor Corp. | www.fairchildsemi.com

FT232 USB-to-Serial bridge
Future Technology Devices International, Ltd. | www.ftdichip.com

MIC5205 Regulator
Micrel, Inc. | www.micrel.com

PIC18F26K22 Microcontroller and PICkit 3 debugger
Microchip Technology, Inc. | www.microchip.com

TXS0102 Bidirectional voltage-level translator
Texas Instruments, Inc. | www.ti.com

processing thread breaks into two parts; one runs `xmlhttp.onreadystatechange` and the other continues executing the main thread. The `xmlhttp` object goes through five states on its way from being created, `readyState==0`, to being completed with a response ready, `readyState==4`. These state changes are monitored by the callback function associated with the `onreadystatechange` variable. When the response is ready, the stamp.txt file's entire contents are available through `xmlhttp.responseText`.

Since the thread that contains the XML request split from the main thread, the `updateMapMarkers()` function needs to reunite them to use `gpsString`. `setTimeout` does this by creating a 300-ms delay for the read of stamp.txt before executing its callback function. This function parses out the information contained in stamp.txt, occasionally refreshes the bus stop markers, and redraws the bus icon. These last two functions require information derived from the breadcrumb array.

The breadcrumb array was analyzed offline to determine the index in the array for each bus stop. This is information is stored in the bus stop array (see the rightmost column in "Bus stop information" in **Figure 2**). The bus' estimated time of arrival (ETA) at a particular stop is two times the difference between the bus' current breadcrumb index, which is given by `bus.breadIndex` and the bus stop's breadcrumb index. The factor of 2 is introduced because there is 2 s between each breadcrumb. Each stop's ETA is provided to the user whenever the user lingers over a particular bus stop. Breadcrumbs also help determine which bus stop the bus is headed toward. This is accomplished by taking the bus' breadcrumb index and finding the next highest bus stop breadcrumb index.

FINAL IMPLEMENTATION

Photo 2 and **Photo 3** show the final hardware implementation of the base station and mobile tracker. The base station radio is connected to an L-com 8 dBi flat-patch antenna (white cable) covering a roughly 120° arc. This is perfect for its location near Bus Stop 4 in **Photo 1**. A Fairchild Semiconductor LM7805 linear regulator on the interface board provides auxiliary 5-V power for the radio. The mobile tracker is installed entirely inside the bus. Initial concerns about attenuation of the GPS and radio signals through the bus' fiberglass skin proved to be unmerited.

You can view the bus tracker's final implementation at http://bus.bd.psu.edu from 7:40 AM to 7:00 PM EST Monday through Friday. The system works remarkably well, providing reliable, accurate information about our campus bus. Best of all, it does this autonomously, with very little supervision on our part. It has worked so well, we have received additional funding to add another base station to cover an extended campus route next year.

ABOUT THE AUTHOR

Chris Coulston (coulston@psu.edu) holds a BA, Physics (Slippery Rock University); a BS and an MS, Computer Engineering (Penn State University); and a PhD, Computer Science (Penn State University). He has worked at Penn State Erie for 13 years an Electrical and Computer Engineering professor and currently serves as the Department Chair of the Computer Science and Software Engineering department. His technical interests include embedded systems, computer graphics algorithms, and sensor design.

Daniel Hankewycz (djh5533@psu.edu) is a second-year Computer Engineering student at Penn State Behrend. Daniel enjoys working with embedded systems, PCB design, and computer numerical control (CNC) technology.

Austin Kelleher (alk5492@psu.edu) is a second-year Computer Science student at Penn State Erie. His technical interests include cross-platform technologies, network security, and artificial intelligence.