OBD Communication with the BeagleBone and ELM327

By Need4C

Introduction

The OBD (On-Board Diagnostic) port on a car provides lots of different data about the car, including data about the engine, error codes, and more. In this guide, we'll outline how to take advantage of this port to query information from the car in real time using the Beaglebone and the ELM327 chip (found in the vast majority of OBD-II cables on the market).

Setup

To communicate with the car, you'll need an OBD-II cable. These cables are abundant online, commonly using the ELM327 chip to interface with the car's ECU (engine control unit), and providing a serial connection over USB to send and receive data. Look for a cable that uses the ELM327, and does serial over USB. You can use an OBD ELM327 Bluetooth dongle instead of a cable if you prefer, but this guide will not cover any Bluetooth communication.

An ELM327 OBD-II USB cable (basically every cable you'll find looks like this)

On the other end, you'll need your BeagleBone (or any other Linux computer/board). We'll be writing a C module to communicate with the ELM327 over Serial, to query data and receive the response.

Serial Communication in C

To talk to the ELM327 and fetch data from the car, we're going to need to communicate over serial in our app. There is lots of code online for doing serial communication in C, so google around if you're having trouble getting it working. The general structure for our code is to set the option flags on the serial terminal struct, set blocking options on the serial terminal struct, then open the serial port file descriptor.

First up, setting option flags on the termios struct:

```
static int set_interface_attribs (int fd, int speed)
{
      struct termios tty;
      memset (&tty, 0, sizeof tty);
      if (tcgetattr (fd, &tty) != 0) {
            printf("obd.c: error %d from tcgetattr", errno);
            return -1;
      }
      fcntl(fd, F_SETFL, 0);

      tty.c_cflag |= (CLOCAL | CREAD);
```

```
        tty.c_lflag &= !(ICANON | ECHO | ECHOE | ISIG);
        tty.c_oflag &= !(OPOST);
        tty.c_cc[VMIN] = 0;
        tty.c_cc[VTIME] = 100;

        if (tcsetattr (fd, TCSANOW, &tty) != 0) {
                printf("obd.c: error %d from tcsetattr", errno);
                return -1;
        }

        cfsetospeed (&tty, speed);
        cfsetispeed (&tty, speed);

        return 0;
}
```

Next, let's write a function to configure if you want reading/writing from the serial port to be blocking (ie to wait at the read/write call until all data has been read/written):

```
static int set_blocking (int fd, int should_block)
{
        struct termios tty;
        memset (&tty, 0, sizeof tty);
        if (tcgetattr (fd, &tty) != 0) {
                printf("obd.c: error %d from tggetattr", errno);
                return -1;
        }

        tty.c_cc[VMIN]  = should_block ? 1 : 0;
        tty.c_cc[VTIME] = 5;            // 0.5 seconds read timeout

        if (tcsetattr (fd, TCSANOW, &tty) != 0) {
                printf("obd.c: error %d setting term attributes", errno);
                return -1;
        }

        return 0;
}
```

Finally, let's open the serial port file descriptor, using the two functions we just defined. Note that we use a baud rate of 38400 for the serial connection (the ELM327 chip specifies either 9600 or 38400 baud), and we're turning blocking on. Set the `OBD_SERIAL_PORT` constant to the path of the serial port, which will probably be `/dev/ttyUSB0`.

```
static int open_serial_port_fd()
{
	char *port_name = OBD_SERIAL_PORT;

	int serial_port_fd = open(port_name, O_RDWR | O_NOCTTY | O_NDELAY);
	if (serial_port_fd < 0){
		printf("obd.c: error %d opening %s: %s", errno, port_name,
			strerror(errno));
		return -1;
	}

	int status;
	status = set_interface_attribs (serial_port_fd, B38400);
	status = set_blocking (serial_port_fd, 1);
	if (status != 0) {
		return -1;
	}

	return serial_port_fd;
}
```

Now you can read from and write to the file descriptor like you would with any other file descriptor:

```
write(serial_port_fd, “SHOW ME WHAT YOU GOT”, 20);

char buffer[20];
read(serial_port_fd, &buffer, 20);
```

OBD Commands

To send commands, we’ll have to wade through some documentation to figure out what the command string is, and how to parse the result. We’ll cover the basics of commands, but for all the nitty gritty technical details on sending and receiving commands, take a look at the ELM327’s fantastic documentation (see `ELM327DS.pdf`, page 30 has all the details for this next part).

First off, we need to figure out what we’re going to send. Find the command you want to send from the `OBD-II PIDs.pdf` document. You’ll need to know the mode and the PID (Parameter ID) to make the command.

As an example, let's look at the command to check all supported PIDs, which is mode 01, PID 00. Note that these numbers are in hex, and basically all of the data we send/receive will be in hex. To send this command, we'll make a string in the form of "`$MODE $PID\r`", which in our case looks like "`01 00\r`". Write that to the serial port file descriptor, and read back the response, which will look something like `41 00 BE 1F B8 10`. The ELM327 documentation on page 30 explains what this means:

> The 41 in the above signifies a response from a mode 01 request (01 + 40 = 41), while the second number (00) repeats the PID number requested. A mode 02, request is answered with a 42, a mode 03 with a 43, etc. The next four bytes (BE, 1F, B8, and 10) represent the requested data, in this case a bit pattern showing the PIDs that are supported by this mode (1=supported, 0=not).

So responses contain a first number to validate the mode (40 + mode), a second number to validate the PID (just the PID), and the rest of the numbers are the data we requested. We don't care about the data in this sample query, it's just a good command to make sure the connection is working. For other commands, we'll need to do some work to get a human readable value from the response.

In C, it's easy to prepare command strings by using something like this:

```
unsigned int mode = 0x00;
unsigned int pid = 0x01;
char command_string[5];
sprintf(command_string, "%02X %02X\r", mode, pid);
```

OBD Response Conversion

Since OBD commands return hex values as response data, we'll need to convert that hex into something more readable. Referring back to the list of OBD PIDs used earlier, we see there are formulas specified for each PID that we can use for conversion. For example, `01 0C` is the command for getting the current RPM. It's formula specifies `(256*A + B) / 4`, which means we'll need to take the first result value (A) and the second result value (B), apply this operation to them, and we'll have our result.

Let's take a crack at it. We'll send `01 0C` to the car, and let's say we get back `41 0C 1A F8`. We know `41 0C` are values confirming our query (40 + 1 to indicate our query mode was `01`, and `0C` to indicate our query PID was `0C`), and we have an additional 2 values: `1A F8`. These

are the values A (`1A`) and B (`F8`), which we'll use in our formula (`((256*A + B) / 4)`). In this case, we can plug the numbers in and we get 1726.00 RPM.

Most of the PIDs will require conversions like this, so it may be worthwhile to write functions to do the conversions. Here's an example conversion function for RPM:

```
static float convert_rpm(unsigned int* result_values)
{
	return ((float) result_values[0] * 256.0 + (float) result_values[1])
		/ 4.0f;
}
```

Where result_values is an array containing the values returned from our query (result_values[0] being A, and result_values[1] being B).

Tips, Tricks, and Issues

1. The ELM327 has a slew of commands that modify how it sends/receives data, which can be helpful for you to configure at the start of your application. These commands start with `AT`, and don't get sent to the car. Specifically, we used the `ATE0` command to turn off echo, which makes parsing the response easier since you won't have to pass over the command text every time.
2. If you're reading garbage data from the serial port, the problem is very likely that the baud rate is incorrect (welcome to the wonderful world of serial!). We experienced one particularly annoying problem, where our application would read garbage data until we manually used `screen` to connect to the serial port at the baud rate we wanted (38400), at which point our app read data perfectly fine. We speculate this might have something to do with our code incorrectly setting the baud rate, so it may be worth investigating this further if you have the same issue. We did not come up with a solution to this issue.
3. You can test your code without plugging the BeagleBone into a car every time by using a simulator, like `obdsim` (found packaged with the `obdgpslogger` application, which can be installed by running `apt-get install obdgpslogger`). Run `obdsim` and it'll give you a virtual serial port which you can connect your app to and run queries as if you're connected to an actual car.
4. We found that the car we tested on returned the mode differently than the ELM327 specifies. The mode should be 40 + $MODE, but we were just getting $MODE back. For example, if the mode is 1, the returned mode is 41, but we were receiving 1. A small change to our parsing function fixed this, handling a return mode of 1 as a success.
5. Remember to modularize your code! For example, setting up a module for serial communication, a module for command creation, a module for result parsing, and a module containing the definitions of command/result structures makes it easy to

add/change commands in the definition module without affecting large portions of your code.