Restoring Software and Improving Inertial Sensors on the Overbot Autonomous Vehicle

Jaakko Karras
Harvey Mudd College

Advised By:
Professor Gabriel Elkaim, Autonomous Systems Lab
John Burr, Autonomous Systems Lab

SURF-IT REU, Summer 2009

Project Background

The Overbot is a former DARPA Grand Challenge autonomous vehicle, which participated in both the 2004 and 2005 races. The vehicle was donated to the UCSC Autonomous Systems Lab in the fall of 2005, where it is currently used for academic research. From a hardware standpoint, the Overbot consists of a six-wheeled Polaris Ranger utility vehicle, equipped with an assortment of sensors, actuators, and computers. These components are tied together through software, written in C++ and run on the QNX real-time operating system, to achieve autonomous driving. The work done this summer has focused on both the software and hardware elements. First, the Overbot codebase was restored to a working state for future work with novel path finding algorithms; then, with the software up and running, work was done in reducing temporal drift in the Overbot's inertial sensors.

As a research testbed, the Overbot has been used to implement a number of autonomous control algorithms and its codebase put through numerous design iterations. The myriad of software tweaks, however, had rendered much of the codebase non-functional and its history difficult to follow. The first phase of this project centered on understanding both the past and present states of the Overbot software, as well as restoring it to its previously functional form. The structure of the Overbot codebase is discussed in the following section.

The Overbot's autonomous driving hinges on its ability to obtain accurate data on its surroundings from an array of sensors. The Overbot derives a visual understanding of surrounding terrain using a laser range finder (LIDAR) unit. The vehicle is also equipped with a camera and software that carries out road-recognition, but this feature is no longer used. In addition to vision, the Overbot tracks its location relative to a set path using a GPS unit, coupled with an inertial sensor package. GPS data is used whenever possible; that is, when the data is within a threshold accuracy of 15 centimeters. When the GPS data becomes less accurate, or otherwise unreliable, position data is corrected using the inertial unit. Overall, this position-estimation approach works well with reliable GPS data. Unfortunately, temporal drift in the inertial unit renders the Overbot largely ineffective when the GPS becomes spotty, since the inertial unit will accumulate as much as two to four meters of error for every 100 meters of dead reckoning [1]. Clearly, errors of this magnitude are unacceptable when driving hundreds of kilometers on narrow desert roads, such as those encountered during the Grand Challenge. Much of this summer, therefore, was spent replacing the existing inertial unit with two high-precision modules, including a MIDG GPS/INS unit and a fiber optic gyroscope. This work is discussed in later sections.

Overbot Software

The Overbot software is structured as a handful of "servers", or processes that run under a central process called "Watchdog". The Watchdog process monitors each of the subordinate processes, restarting the entire system in the event that any of these fail. The various processes communicate amongst one another, as necessary, in accordance with the QNX message-passing standards. All software is launched by starting Watchdog with a "startfile" specifying the desired processes and waypoints file.

On the QNX PCs, startfiles are created under `/gc/src/qnx/nav/map/`. Watchdog is launched from `/gc/src/qnx/support/watchdog/watchdog <startfile>`. For instructions on launching Watchdog on the Overbot, refer to the yellow instruction card located in the vehicle itself.

At a high-level glance, autonomous driving is handled by three processes: the Steer, Move, and Map servers. The Map server aggregates data from the sensors to build a global terrain map, indicating drive-able regions around the vehicle. The Steer server consults the map maintained by the Map server and uses this information to perform path planning between waypoints. Once a path has been found, the Steer server propagates its information to the low-level Direction server, which then directly controls the front wheels. Acceleration and braking are managed by the Move server, which propagates information to its low-level counterpart, the Speed server.

Software Directory Structure

The following presents an outline of the Overbot codebase structure, highlighting those components that are of greatest interest.

`/gc/src/qnx:`

`/common:`	
`/lib:`	Contains libraries that need to be built at the beginning of all installations. Libraries are installed using `buildalllibs.sh.`
`/include:`	Contains header files for the Overbot codebase. Header files are moved here when installing applications.
`/control:`	
`/galil:`	Text files used for configuring the Galil motor controllers.
`/speed:`	Source code for the Speed server, which needs to be installed along with the rest of the software.
`/drivers:`	Contains driver software for hardware installed on the Overbot
`/lms:`	Driver for the LIDAR unit
`/dummylms:`	False driver which simulates LIDAR functionality, used for testing purposes

`  /vorad:`	Driver for Vorad obstacle detection unit
`/nav:`	Contains software responsible for steering and driving the Overbot
`  /fusednav:`	Newer version of `gpsins` (see below)
`  /gpsins:`	Server that communicates with the GPS and Inertial units
`  /gpsins_logger:`	Server that logs GPS and Inertial data
`  /map:`	Server that builds and maintains the global terrain map
`  /move:`	Server responsible for acceleration and braking decisions
`/support:`	
`  /remotemsgserver:`	Enables message passing between Watchdog processes and external applications (for testing purposes)
`  /usercontrol:`	Code for a control interface to the Overbot
`  /watchdog:`	Code for the global Watchdog application
`/vision:`	
`  /roadfollower:`	Software for camera-based road recognition
`  /dummyroadfullower:`	False roadfollower used for testing purposes

Software Installation Workflow

The software installation procedure is documented on the lab wiki, but is also reproduced below. The given workflow reproduces a basic, functional set of binaries using *revision 4040* of the Overbot subversion repository.

Checking out *Revision 4040*:

```
svn co -r 4040 svn://overtux.cse.ucsc.edu:3691/overbot/trunk/gc
```

Installing Binaries:

The following outlines the steps needed to recreate the basic functionality of the Overbot. Steps one through six can be automated using the software installation script posted on the ASL lab wiki.

1. Build Overbot Libraries
   a. `cd /src/qnx/common/lib`
   b. run `bash buildalllibs.sh` (within this directory)
2. Build Overbot Drivers
   a. `cd /src/qnx/drivers`

b. Drivers must be built individually by running:

```
make
make install
```

within each driver directory.

c. Perform for each of the following:

```
dummylms
lms
vorad
```

d. Under `lms` run:

```
make install_headers
```

3. Build Control Software
   a. `cd /src/qnx/control/`
   b. Make and install the contents of `/speed`
4. Build Nav Software
   a. `cd /src/qnx/nav/`
   b. Make and install each of the following:
   ```
   fusednav
   gpsins
   gpsins_logger
   map
   move
   ```
   c. Note that `mapfile` and `waypt` are no longer in use.
5. Build Support Software
   a. `cd /src/qnx/support/`
   b. Make and install each of the following:
   ```
   remotemsgserver
   usercontrol
   watchdog
   ```
6. Build Vision Software
   a. `cd /src/qnx/vision/`
   b. Make and install `dummyroadfollower`
   c. Note that `dummyroadfollower` is solely a simulator component and that the Overbot will later require the installation of `roadfollower`
7. Build GUI Interfaces
   a. The QNX GUI interfaces must be compiled from within QNX's PhAB development software
   b. Launch PhAB from `launch menu >> development >> builder`

m*anual drive:*

In PhAB:
i. Navigate to: `/src/qnx/support/manualdrive`
ii. Select `application >> build and run` or hit `F5`

iii. Select target:

Click `generate` button

From `ntox86_cpp` choose `gcc`

Click `generate`

iv. Click `make`

In terminal:

i. `cd /src/qnx/support/manualdrive/src/`

ii. As root: `make install`

gpsins_gui:

In terminal:

i. `cd /src/qnx/nav/gpsins_gui/src/`

ii. `make proto`

In PhAB:

i. Navigate to: `/src/qnx/nav/gpsins_gui`

ii. Select `application >> build and run` or hit `F5`

iii. The target should already be selected. If not, repeat steps from *manual drive*.

iv. Click `make`

In terminal, again:

i. `cd /src/qnx/nav/gpsins_gui/src/`

ii. As root: `make install`

navviewer:

In PhAB:

i. Navigate to: `/src/qnx/nav/tests/navviewer`

ii. Select `application >> build and run` or hit `F5`

iii. The target should already be selected. If not, repeat steps in *manual drive*.

iv. Click `make`

In terminal:

i. `cd /src/qnx/nav/tests/navviewer/src`

ii. As root: `make install`

Interfacing the Fiber Optic Gyroscope

As discussed in the section on project background, the Overbot's existing inertial sensors suffer from excessive temporal drift. One proposed solution to this problem is to replace the current set of inertial sensors, along with the GPS unit, with two precision units: the MIDG INS/GPS and a fiber optic gyroscope (FOG). Under the proposed solution, data obtained from both units would be coupled through a processing unit, such as a microcontroller, and used to derive accurate heading and position data. The following sections detail work that was done in interfacing the latter of the two units, the fiber optic gyroscope, using a dsPIC33F microcontroller.

The fiber optic gyroscope is a high-precision alternative to MEMS gyroscopes. The underlying technology is based on optical interference; two beams of light, traveling in opposite directions around a coil of fiber optic cable, recombine with one another, producing an interference pattern, as shown in *Figure 1* below. The resulting pattern depends on the phase difference between the two beams, which is dependent on the rate of rotation of the gyroscope. Hence, the interference pattern can be used to compute angular velocity. Whereas a conventional MEMS gyroscope might drift as much as 200 degrees per hour when held stationary [2], a quality FOG will drift on the order of 18 degrees per hour. This reduction in temporal drift is the primary motivator in switching to a FOG unit for heading estimates.

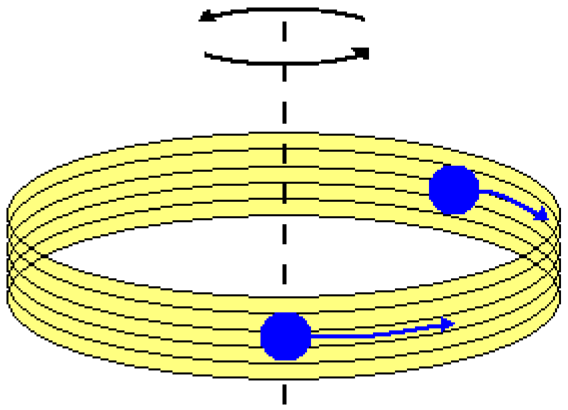

Figure 1. Diagram of fiber optic gyroscope operation.

Interfacing through QNX

Most of the software on the Overbot runs on a single computer under the QNX real-time operating system. One option, therefore, would be to interface the fiber optic gyroscope directly to that computer via a serial port. Data could be read in through a software driver, much as it is under the current hardware setup. This possibility was pursued, albeit briefly, using an existing serial library from the Overbot codebase.

The FOG serial port reader makes use of an Overbot library called *myserial.h*, which essentially encapsulates the UNIX *termios* library. Using *myserial.h*, the serial port reader sets up a serial object with parameters 38400baud, 8 data bits, 1 stop bit, no parity. This object is then used to read data, one bit at a time, into a simple state machine that scans for delimiters. In this case, data words are delimited with a carriage return, linefeed sequence (ASCII 13 and 10). Once received into a buffer, each data word is parsed by removing space characters and converting the remaining digits into a float.

It's worth noting that sending data to the screen in QNX is incredibly taxing on the system, and will cause serial data to be lost. The serial reader, therefore, minimizes use of standard output. Although the QNX serial port reader works sufficient well in channeling data from the FOG module to the Overbot computer, it was decided that interfacing both the FOG and the MIDG to the computer in this manner, as well as coupling the two through added software, would unnecessarily increase the load on that computer. Thus an alternate solution, involving a dsPIC33F microcontroller, was devised.

Interfacing through dsPIC33F

In the interest of decreasing the computational load on the Overbot's primary computers, both the FOG and MIDG units will be interfaced through a dsPIC33F microcontroller, which will perform estimation of the vehicle's position and heading. A block diagram of the proposed solution is given in *Figure 2*. An added advantage of bundling the two sensors in this manner is that it increases the modularity and maintainability of the Overbot's various sensor groups.

The software for interfacing the FOG to the dsPIC33F microcontroller is written in C, specifically for the Microchip C30 compiler. The software makes use of both UART modules, one for reading the FOG unit and the other for transmitting. Data is accumulated in a series of circular buffers, originally written by Mariano Lizarraga for the ASL autopilot and modified slightly for use with the FOG. Both the receive and transmit functions are interrupt driven, allowing the microcontroller to perform any future processing that might be desired, such as Kalman filtering.

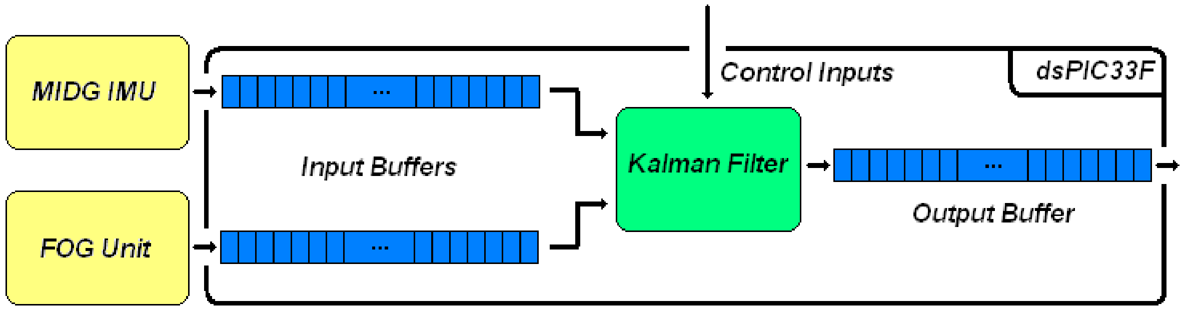

Figure 2. Block diagram for MIDG, FOG bundle.

FOG Software Outline:

The following is an outline of the FOG interface software, most notably the configuration of the two UART modules.

dsPIC_FOG.c:

configure():
Turns off nested interrupts (ability for interrupts to interrupt one another)

switchClock():
Performs clock switch from slow, stable internal RC oscillator to faster external clock. PLL configured for 40MHz (40MIPS).

initCircBuffers():

Initializes transmit and receive CircBuffers for both UARTs.

configUART1():

Configures receive interrupt to fire whenever the receive buffer exceeds three bytes (out of four), and the transmit interrupt to fire after each transmission. Also sets baud rate to 38400, which corresponds to the baud rate on the FOG unit.

configUART2():

Repeats the configuration of UART1, but for UART2. Hence, the two UARTs are identical.

configFOG():

Configures the FOG to perform integration, and zeros the unit by sending it a sequence of two characters.

RX Interrupt Service Routines (x2):

The two receive-oriented interrupt service routines are relatively straightforward. These will read the contents of the hardware receive buffers and append the read bytes to the larger software receive buffers. In the event that the software buffer is full, the interrupt is turned off.

TX Interrupt Service Routines (x2):

The transmit-oriented interrupt service routines simply read data off the front of the transmit CircBuffer, writing it to the UART for transmission. In the event that the transmit CircBuffer is empty, the interrupt is turned off.

main()

The main function runs an infinite loop which constantly moves data from the receive buffer of the receiving UART to the transmit buffer of the transmitting UART. Processing stages could be inserted in between this transfer. In addition to shuffling data between the software buffers, the main function also checks any disabled interrupts, reactivating these if necessary.

MAX3111 Additional UART Module

The dsPIC33F microcontroller supports up to two UART modules. As can be seen from the block diagram in *Figure 2*, the MIDG/FOG bundle requires three: two for receiving data and one for transmitting it. Hence, an additional UART module is needed to complete the package. Maxim's MAX3111 SPI to UART chip lends itself nicely to this task. Among its many useful features, the Maxim chip contains internal RS-232 transceivers and receivers, meaning that configuring an additional UART module for the dsPIC33F is as simple as writing an SPI driver.

MAX311x Software:

The software for the Maxim UART chip is contained in a simple library called *MAX311x* (includes both header and implementation files). This software effectively provides an abstracted UART in the sense that it encapsulates the entirety of the required SPI interaction, allowing the user to simply read and write to a pair of transmit and receive buffers (as they would when interacting with a standard UART module). The content of the *MAX311x* library is outlined below.

MAX311x.h:

The header declares the constructor for the MAX311x. This is the only function that needs to be called to set up and begin interacting with the UART module. The constructor takes pointers to two CircBuffers, which the software then uses to transmit and receive data through the abstracted UART. The user interacts with the UART by writing to and reading from these buffers.

MAX311x.c:

The implementation file contains a number of configuration and auxiliary functions that interact with the MAX311x chip:

configTimer():

Configures timer #2 for use with the output compare module, which generates a CMOS clock for the MAX311x chip. The timer period is set to 9, which produces a square wave of frequency 1.842 MHz.

c*onfigOC():*

Sets output compare module #1 to toggle whenever timer #2 hits a value of 8. The value of 8 is largely arbitrary; any value within the period of time #2 would be fine.

configSPI():

This function is crucial in that it sets up the appropriate SPI waveforms for communicating with the MAX311x chip. The waveforms are configured as CPOL = 0, CPHA = 0 using the SPIxCON1bits.CKP and SPIxCON1bits.CKE bits. The serial clock is configured to run at 2.5MHz using the primary and secondary prescalars. Framed mode is disabled and slave select is run manually using _RB2 (digital I/O pin).

configIRQ():

Sets up external interrupt #4 to listen to the IRQ pin on the MAX311x chip. The interrupt is active low.

configMaxim():

Configures the Maxim chip by sending a WriteConfig command over the SPI interface. The Maxim UART is configured to interrupt on both receive and transmit. The serial format is set to 8 data bits, 1 stop bit, no parity and 38400 baud.

_INT4Interrupt():

The INT4Interrupt service routine fires whenever the Maxim chip has transmitted or received data. The service routine begins by determining the cause of the interrupt (by sending a readData request) and then handles the interrupt accordingly. Note that the ISR handles interrupts using the *spiTransaction()* helper function, which is blocking.

spiTransaction():

spiTransaction() performs a single SPI transfer by writing the two bytes it's given and returning the bytes it receives. SPI transactions are initiated by writing to the SPI transmit buffer, which starts the sequential transfer of bits out of the unit. Pin _RB2 is driven low to signal the beginning of a transfer to the MAX311x chip.

initMAX311x():

This is the constructor for the abstracted UART. It takes, as its arguments, pointers to two CircBuffers: one that the user will use for queuing outgoing data, the other for reading received data.

MAX311x Hardware:

A small circuit board was designed for the MAX311x hardware after successful testing on a breadboard (*Figure 3)*. The circuit board is relatively straightforward, and takes in the full set of SPI input signals (DIN, DOUT, SCLK, IRQ, CS), as well as a clock signal and power. Initial testing indicates that the PCB works well with the MAX311x library software.

The SPI and clock signals can be drawn from the daughter board attached to the Explorer16 development board. Wires are connected as follows:

SDO1 (RF8): Red wire, connects to DIN pin on MAX311x
SDI1 (RF7): Green wire, connects to DOUT pin on MAX311x
SS1 (RB2): White wire, connects to CS pin on MAX311x
SCK1 (RF6): Red wire, connects to SCLK pin on MAX311x
INT4 (RA15): Green wire, connects to IRQ pin on MAX311x
MAX311x Clock (RD0): White wire, connects to X1 pin on MAX311x

Power (+3.3V) and ground can be drawn from a standard power supply, or from the Explorer16 board itself. To avoid problems, ensure that the two connections share a common ground.

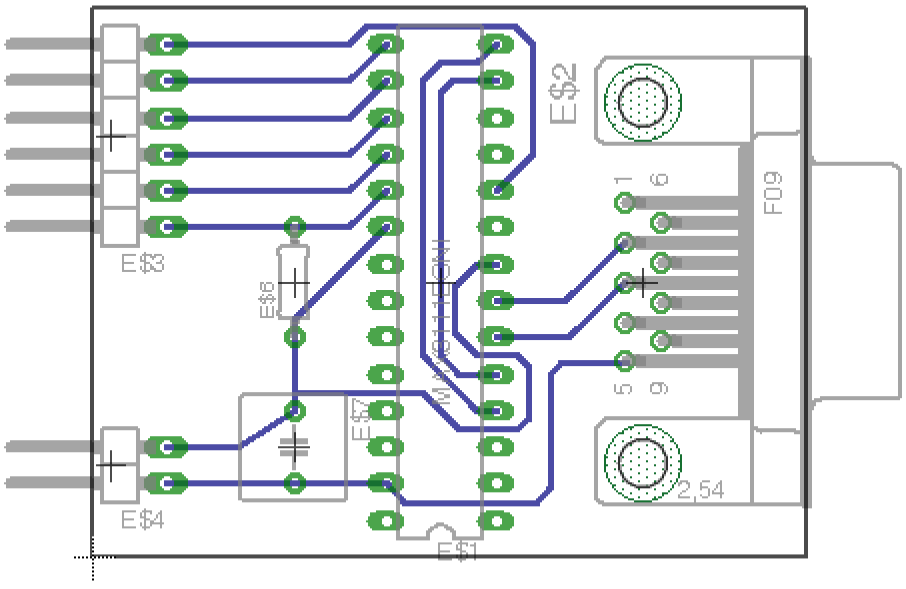

Figure 3. Circuit layout for MAX311x hardware.

Conclusions and Future Work

Initial testing indicates that the FOG unit, interfaced through the dsPIC33F microcontroller, drifts on the order of 18 degrees per hour. Considering that the typical drift of a MEMS package is on the order of 200 to 300 degrees per hour, this is a considerable improvement. Future work remains in polishing up the MAX311x hardware, and in interfacing the remaining MIDG INS/GPS sensor. With the two sensors in place, processing algorithms can be written to produce improved estimates of vehicle position and heading.

Acknowledgments

This project would not have been possible without the support of Professor Gabriel Elkaim, John Burr, and the UCSC Autonomous Systems Lab. I appreciate their technical guidance and insights. I would also like to thank the SURF-IT 2009 REU program, as well as the NSF for administering and providing the funding for this research opportunity.

References

[1] Team Overbot. (2005). DARPA Grand Challenge 2005 Technical Paper.

[2] Elkaim, G., Lizarraga, M., Pedersen, L., *Comparison of Low-Cost GPS/INS Sensors for Autonomous Vehicle Applications. 2008.*