Proceedings of
ASME 2006 International Student Conference
May 26-28, 2006, Istanbul, Turkey

SOFTWARE AND CONTROL ARCHITECTURE DEVELOPMENT OF AN AUTONOMOUS VEHICLE

Keith Yu Kit Leung, MASc Candidate
Lab for Autonomous and Intelligent Robotics
Department of Mechanical Engineering
University of Waterloo
Waterloo, Ontario, N2L 3G1
Canada
Email: kykleung@lair.uwaterloo.ca

Christopher M. Clark, Assistant Professor
Jan P. Huissoon, Professor
Lab for Autonomous and Intelligent Robotics
Department of Mechanical Engineering
University of Waterloo
Waterloo, Ontario, N2L 3G1
Canada
Email: cclark@mecheng1.uwaterloo.ca, jph@uwaterloo.ca

ABSTRACT

This paper presents the design and development of the software and control architecture of the Centaur, an autonomous amphibious and all-terrain utility vehicle project. The adaptation of a hybrid between planning and behaviour based control architectures for the vehicle is explained using decomposition methods. This paper will show how subsystems relate and operate with each other to accomplish the four essential tasks of perception, localization, cognition, and motion control. Implementation of the control architecture into a multithreaded and modular based software structure will be described. The discussion on software development will also include interfacing with hardware components on the autonomous vehicle and the design of a user interface.

INTRODUCTION

The Centaur, shown in figure 1, is an amphibious all- terrain utility vehicle suitable for many outdoor applications. In January 2006, a project was started at the University of Waterloo to convert a Centaur, donated by its manufacturer Ontario Drive and Gear, into an autonomous vehicle. The first step in the project was to define the desired level of autonomy, and a goal was set to develop the vehicle to be capable of autonomously traveling to pre-specified waypoints. When this is accomplished, the level of autonomy will be increased with the addition of a motion planner that can generate a waypoint list from some given start and goal positions. The objective of this paper is to provide documentation of the software and control architecture developed and used for the vehicle. The software developed serves as an architecture for linking specific system components for the autonomous vehicle, to be developed by other designers. Furthermore, this paper shows the potential of using a general purpose computer together with various sensors and actuators to create an autonomous vehicle.

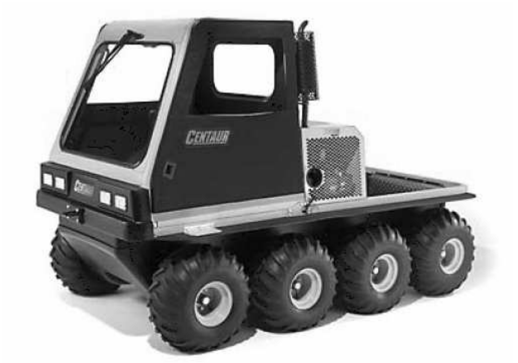


Figure 1. The Centaur

Control and processing of information for the vehicle is performed on a laptop running the autonomous vehicle system software. In many autonomous vehicle projects, multiple high performance computers are used. Examples of this can be found on vehicles competing in the DARPA Grand Challenge autonomous vehicle competition such as the entry presented in [1], which required the use of six Pentium-4 based computer systems. At the current stage of development of the Centaur, it is desireable to minimize the number of computers that will be required to control the vehicle autonomously. Being able to achieve this objective with a typical laptop computer eliminates the cost penalty of expensive high performance computers, and presents the potential of manufacturing cost effective autonomous vehicles in the future. The concept of using a laptop, as opposed to a complex fixed vehicle computer system, presents several additional benefits. These include allowing multiple designers to develop software modules for the vehicle offline and connect to the vehicle for testing, and the fact that a laptop is physically smaller, making it more convenient to install on the vehicle.

The autonomous vehicle system software is programmed in C++ and runs under the Windows XP operating system. While different operating systems such as Linux have their advantages and disadvantages compared with Windows, the choice of the specific operating system is mainly attributed to accessibility. Windows XP is a popular operating system and this allows any user who has obtained the vehicle control software to connect and operate the vehicle. Microsoft Foundation Class (MFC) libraries are used to simplify programming for the Windows environment.

In terms of hardware, it is necessary for the computer running the software to be equipped with at least one universal serial bus (USB) port to use all available modes of the program. The Centaur is equipped with a USB hub through which all the sensors and actuators on the vehicle are connected. A laptop with an Intel or compatible processor is also required, as the hardware counter on the CPU will be used by the vehicle control software.

The underlying architecture of the autonomous vehicle system software was designed with modularity as an important consideration. Modularity allows different functions to be localized, which will make debugging easier, and allows for the study of the effects when a specific module of the vehicle is tuned [2] .

A multi-threaded program structure is also required for the autonomous vehicle application since the software is required to simultaneously read from various sensors operating at different frequencies, while performing the required computation for vehicle motion control [3]. At the moment, the software is capable of communicating with a global positioning system (GPS) receiver, an inertial measurement unit (IMU), a laser range finder, and stepper motors linked to the vehicle's throttle body and steering system.

The vehicle control software provides two running modes: simulation and real time vehicle control. Most system modules operate identically in both modes, with the exception of hardware communication threads which are disabled and replaced by a simulator module that generates artificial sensor readings.

Interaction with users occur during program startup configuration through dialog windows. Settings for the serial ports can be configured through these dialog windows. The file logging option can also be activated at start up. Additionally, vehicle waypoints are manually entered using the waypoint manager before the software takes control of the vehicle. A graphical user interface (GUI) is available during run-time to display vehicle information in both simulation and real-time autonomous operation mode.

CONTROL ARCHITECTURE

The autonomous vehicle system employs a hybrid of planning and behaviour based control architectures. Using an episodic planning system, the cognition routine of creating or modifying the waypoint list is performed during program startup and when the vehicle is blocked by an obstacle.

The point tracker module acts as a behaviour, mapping estimated position to desired forward and angular (yaw rate) velocities. These desired velocities are passed to the low level control to determine the appropriate throttle and steering inputs. Figure 2 summarizes the control architecture.

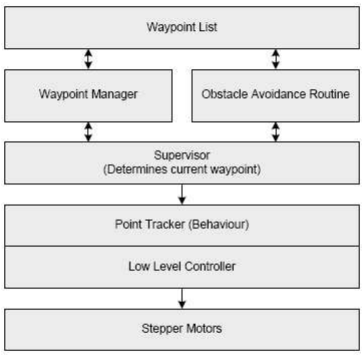

Waypoint List
Waypoint Manager
Obstacle Avoidance Routine
Supervisor
(Determines current waypoint)
Point Tracker (Behaviour)
Low Level Controller
Stepper Motors

Figure 2. HYBRID CONTROL ARCHITECTURE

A temporal decomposition of the control architecture is shown in figure 3, showing the bandwidth of various components and functions of the vehicle control software. Due to the bandwidth of different components of the system, a multithreaded program structure is required. The reported bandwidth of the motion control loop of 20000 Hz is an estimation obtained in simulation mode using the precision timer of a Dell Inspiron 6000 laptop with a 1.5 GHz Intel processor. This is expected to decrease in real-time vehicle control mode as additional threads will be running to communicate with hardware components. Nevertheless, the decrease in bandwidth is not expected to adversely affect system performance in terms of vehicle responsiveness.

SOFTWARE STRUCTURE AND SYSTEM MODULES

System modules for the autonomous vehicle can be classified into five groups: Perception, localization, cognition, motion control, and overhead components. To elaborate on these groups and explain these in the context of autonomous vehicles, perception involves taking sensor readings, which generally involves information from which position, velocity and acceleration can be derived. Determination of these states is known as the localization process. The vehicle control software currently contains two localization methods: Kalman filter localization, and particle filter localization. A description of how these filters work is beyond the scope of this paper, but there are many references available on these topics in literature such as [4] and [5].

Once the state of a vehicle is known, the cognition process addresses how and where it should move to reach its goal.

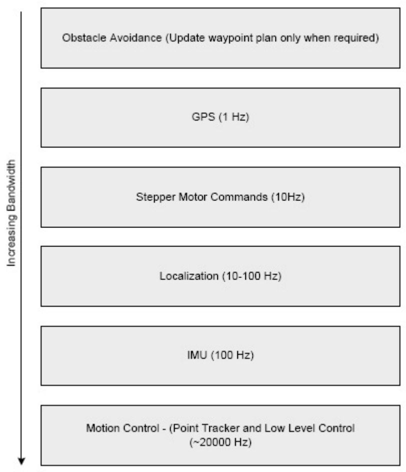

Increasing Bandwidth
Obstacle Avoidance (Update waypoint plan only when required)
GPS (1 Hz)
Stepper Motor Commands (10Hz)
Localization (10-100 Hz)
IMU (100 Hz)
Motion Control - (Point Tracker and Low Level Control (~20000 Hz)

Figure 3. TEMPORAL DECOMPOSITION

Finally, the motion control component addresses what a vehicle needs to do to move according to the cognition process. Figure 4 summarizes this grouping and figure 5 shows the interaction between modules through a control decomposition.

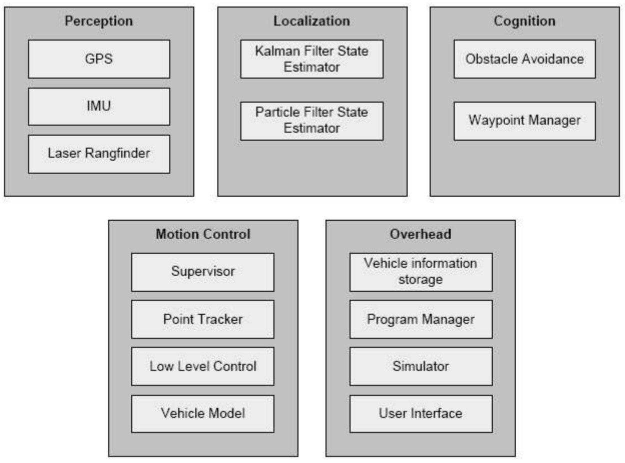


Figure 4. MODULE CLASSIFICATION

Multithreading

As explained previously, since sensors run at different frequencies, perception modules require individually dedicated program threads. Only a single localization routine is used for state estimation, and this is selected in the program startup configuration. The selected localization routine also requires a dedicated thread, as processing time may affect the bandwidth of the entire system if it runs in series with motion control modules. Motion control components run in the main program thread with the exception of the module that handles communication with the stepper motors for controlling throttle and steering.

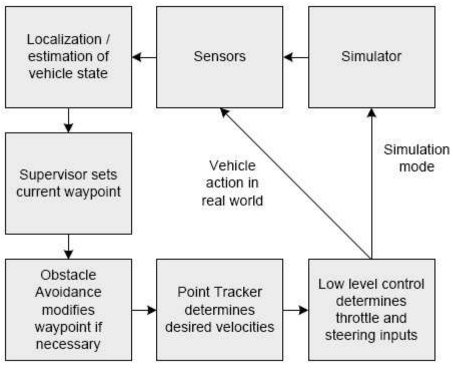

Localization / estimation of vehicle state
Sensors
Simulator
Supervisor sets current waypoint
Vehicle action in real world
Simulation mode
Obstacle Avoidance modifies waypoint if necessary
Point Tracker determines desired velocities
Low level control determines throttle and steering inputs

Figure 5. CONTROL DECOMPOSITION

Figure 6 summarizes the usage and purpose of threads in the program.

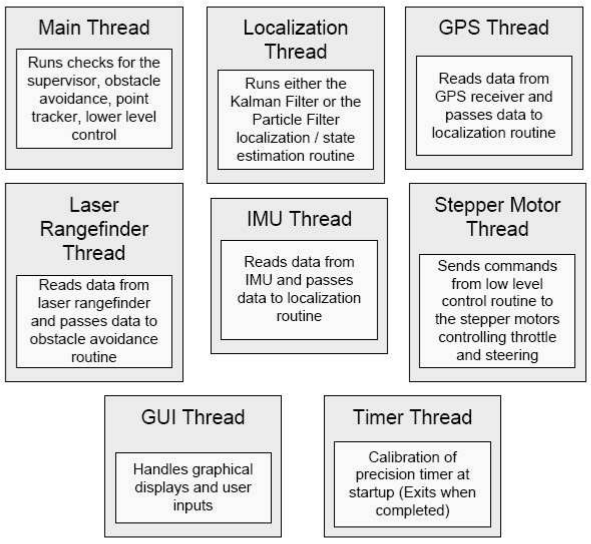

Main Thread
Runs checks for the supervisor, obstacle avoidance, point tracker, lower level control
Localization Thread
Runs either the Kalman Filter or the Particle Filter localization / state estimation routine
GPS Thread
Reads data from GPS receiver and passes data to localization routine
Laser Rangefinder Thread
Reads data from laser rangefinder and passes data to obstacle avoidance routine
IMU Thread
Reads data from IMU and passes data to localization routine
Stepper Motor Thread
Sends commands from low level control routine to the stepper motors controlling throttle and steering
GUI Thread
Handles graphical displays and user inputs
Timer Thread
Calibration of precision timer at startup (Exits when completed)

Figure 6. THREAD USAGE SUMMARY

With a multi-threaded program structure, it is important to ensure that one thread does not write to memory while another thread is also writing or reading to the same memory block. The outcome of such event is often unpredictable [6]. This concern is solved by using critical sections, a type of object available through the MFC library that handles multithread access control. Multiple threads also imply conflicting competition for processing time. MFC event objects are used to pause threads that are waiting to act on the arrival of future data. This is a much more efficient method of handling waiting routines compared to polling. Polling is a technique used to continually check on the status of something in a loop until the desired status has been reached. The problem with this technique is the inefficient use of computation cycles. While this method may be justified for use if the CPU has no better actions to perform [3], this is not the case with the autonomous vehicle system. The event trigger method uses the built in message system of the Windows operating system, and can be viewed as a software version of hardware interrupt routines.

HARDWARE INTERFACING

The autonomous vehicle system software will be receiving data from a GPS receiver, an IMU, a laser rangefinder, and stepper motors as shown in figure 7. With serial communication, the same argument for multithreading can be applied in terms of reading data in the receive buffer. Instead of constantly polling newly received characters in the serial port buffer, a communication routine should pause until a new incoming character is detected in the respective input buffer. When a piece of data has been successfully and completely received, a trigger should then set off for any threads waiting to process the new data.

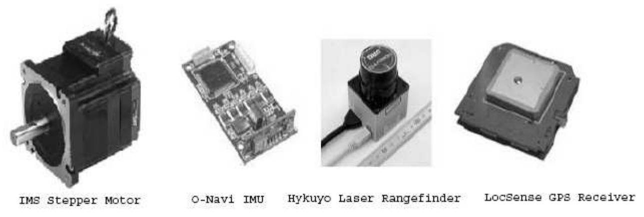


Figure 7. HARDWARE COMPONENTS USED ON THE CENTAUR

Serial transmission of commands is only performed to the stepper motors. Since the motors are mechanical systems with relatively low bandwidth compared to the processing speed of all other program routines, commands are only sent out at a frequency of 10 Hz, even though they are calculated at a much higher rate in the motion control program loop. The output frequency was selected by testing various other frequencies on the actual vehicle.

THE SIMULATOR

When the autonomous vehicle system is in simulator mode, desired throttle and steering commands are not transmitted to the stepper motor hardware, but become inputs to the simulator module instead, as shown in figure 5.

Within the simulator module, the vehicle model is used to determine the accelerations and angular velocities of the vehicle based on the throttle and steering inputs. The details of the vehicle model will not be discussed in this paper, but accelerations and angular velocities are integrated over the period of a simulation time step using a set of differential equations to generate a real vehicle state in the virtual world.

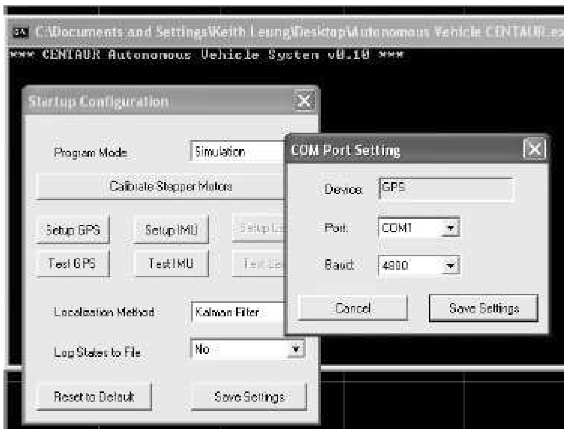
Acceleration and angular velocities calculated by the vehicle model are also used to generate IMU readings by adding noise based on a Gaussian distributed model. This model is based on real IMU data gathered on the Centaur, and the generated IMU data is used by the chosen localization routine to come up with new state estimates.

Along with IMU readings, GPS data is also generated by the simulator by adding bias and noise to the real vehicle position. Again, the noise added is based on the characteristic of the GPS receiver observed in field tests. While the noise is normally distributed, the bias is programmed to slowly drift over time and jump occasionally to simulate GPS satellites entering and exiting the field of view of the receiver. The likelihood of a jump increases with time elapsed since the last jump.

To keep track of simulation time steps, and for generating IMU and GPS readings at the correct frequency, the precision timer of the CPU is used. The timer actually operates as a 64-bit counter which is reset when the computer starts up. The counter is incremented every CPU cycle and by knowing the CPU frequency, time intervals can be calculated.

GRAPHICAL USER INTERFACE

The autonomous vehicle system software uses dialog boxes at startup, as shown in figure 8, and a graphical window during other times to interact with users. A command prompt style window is also available for displaying data which is useful for debugging. The graphical window is programmed using the OpenGL application programming interface. This graphical window is capable of showing the real and estimated vehicle states with 3D vehicle models in simulation mode. The current waypoint is also displayed. In real-time operating mode, only the estimated state and the waypoints will be shown. Camera views can be cycled through at any time with a single keystroke to give various top down views and vehicle chase views. Detailed figures such as exact vehicle coordinates, IMU and GPS outputs can also be toggled on and off. Other functions that can be triggered through the GUI include program exit and simulation reset. Figure 9 is a screenshot of the GUI window.



C:\Documents and Settings\Keith Leung\Desktop\Autonomous Vehicle CENTAUR.e

```
*** CENTAUR Autonomous Vehicle System v0.10 ***
```

Startup Configuration

Program Mode: Simulation

Localization Method: Kalman Filter

Log States to File: No

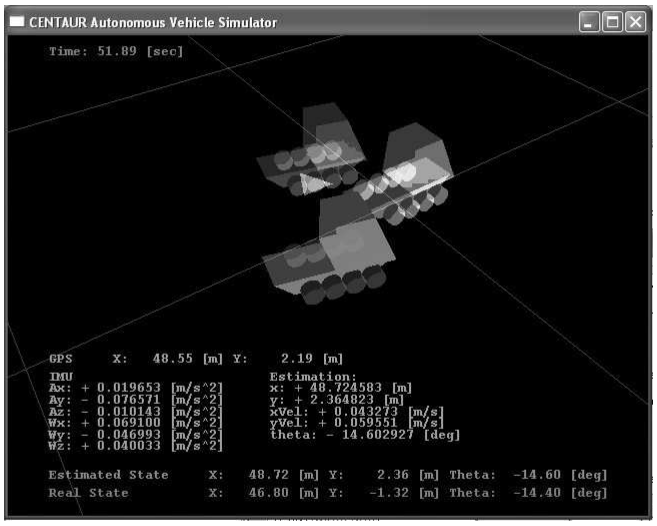
COM Port Setting

Device: GPS

Port: COM1

Baud: 4800

Figure 8. CONFIGURATION DIALOG BOXES AT PROGRAM STARTUP



```
Time: 51.89 [sec]

GPS      X:   48.55 [m] Y:    2.19 [m]
IMU                         Estimation:
Ax: + 0.019653 [m/s^2]      x: + 48.724583 [m]
Ay: - 0.076571 [m/s^2]      y: + 2.364823 [m]
Az: - 0.010143 [m/s^2]      xVel: + 0.043273 [m/s]
Wx: + 0.069100 [m/s^2]      yVel: + 0.059551 [m/s]
Wy: - 0.046993 [m/s^2]      theta: - 14.602927 [deg]
Wz: + 0.040033 [m/s^2]

Estimated State      X:   48.72 [m] Y:    2.36 [m] Theta:  -14.60 [deg]
Real State           X:   46.80 [m] Y:   -1.32 [m] Theta:  -14.40 [deg]
```

Figure 9. GRAPHICAL USER INTERFACE

PROGRAM MANAGER AND VEHICLE INFORMATION STORAGE

Several overhead modules are required for the operation of the software. The program manager takes care of storing configuration settings, and ensures that the proper threads are initiated in different program modes. It also ensures that all threads exit properly and all communication ports are closed when the program exits or resets. Furthermore, the precision timer resides within the program manager to provide accessibility to all program modules.

To exchange information between different program modules, a vehicle information storage class was created. This class is responsible for storing the latest vehicle state estimate and sensor readings. Member functions of this class used for updating and reading data are enhanced with critical section objects for safe multithreading access as explained previously. All modules within the program have access to this class.

A file logging option can be activated during program startup. This data logger will record vehicle states at a frequency of 10 Hz. To prevent the file writing process from impeding other more important routines essential to vehicle operation, memory is allocated on the heap for storing the logged data. As data entries increase, reallocation of memory can occur when the previously allocated block of memory is unable to hold more data. The newly allocated block will be twice the size of the previous. If reallocation is unsuccessful because there is a lack of free memory, the logged data will be written to file immediately. The logged data is also written to file just before the program exits.

CURRENT PROGRESS

All individual components of the vehicle control software have been tested in simulation. Simulation results show that the vehicle is able to follow any set of waypoints with all the modules of the vehicle control software working collectively. Low level control of the vehicle has been tested in field tests and shown to work successfully with desired velocity inputs from the point tracker system. The next step of the project is to have all components tested individually in real life and have all components work together to navigate the vehicle through a simple set of waypoints.

FUTURE WORK

Continual development of individual modules of the vehicle control system will be carried out to improve the performance of the vehicle. Particular areas of potential improvements include the vehicle model, the low level control routines, as well as localization methods. Field tests will allow the vehicle model to become more accurate and allow fine tuning of the low level control routines. Testing will also allow parameters of localization methods to be tuned to perform better in the real world.

From simulation runs, it was estimated that the motion control loop is capable of running at 20000 Hz. Since this bandwidth is more than adequate for responsive vehicle performance, and since communication with the the stepper motors only occurs at 10 Hz, computation time could be allocated to other routines such as localization. Currently, the Kalman localization routine is sometimes unable to keep up with incoming data from the IMU and will occasionally miss a reading. By increasing the priority of the localization routine, the situation should improve, although experimentation is required to observe how this change will affect the rest of the system.

Through the development of the software, limitations imposed by the Windows operating system were observed. For instance, moving a window while the program is running will actually pause the program and affect timing and calculations. Also, the autonomous vehicle system is not guaranteed to be allocated full CPU usage as Windows manages the available resources to all other running programs. An investigation into other operating systems will be performed.

The Centaur has a sister amphibious vehicle named the Argo. For future research involving autonomous control of multiple vehicles, a radio transceiver should be added to the arsenal of hardware onboard the Centaur. The development of sophisticated motion planning module is also a part of future plans to make the vehicle more autonomous and eliminate the need for users to specify waypoints between the start and goal positions.

CONCLUSION

The development of the autonomous vehicle system software for the Centaur has been generally successful. The software architecture provides the link between different modules and it allowed individual systems to demonstrate their performance through simulation. The software was also useful in the development of components that interact with hardware components. It is important to realize that the equipment required to create an autonomous vehicle such as the Centaur does not necessarily require a large monetary resource for backing the project. The work that has been done so far will serve as the foundation of future research and development of vehicle autonomy.

ACKNOWLEDGMENT

The Centaur along with its sister vehicle Argo were donated to the University of Waterloo by Ontario Drive and Gear.

REFERENCES

[1] U. Ozguner, K.A. Redmill, A. B., 2004. "Team terramax and the darpa grand challenge: A general overview". In IEEE Intelligent Vehicles Symposium, pp. 232–237.
[2] Siegwart, R., and Nourbakhsh, I., 2004. *Introduction to Autonomous Mobile Robots*. Massachusetts Institute of Technology.
[3] Valvano, J., 2000. *Embedded Microcomputer Systems: Real Time Interfacing, 2nd Edition*. Thomson-Engineering Pulishers.
[4] Gelb, A., ed., 1999. *Applied Optimal Estimation*. Massachusetts Institute of Technology.
[5] Doucet, A., Freitas, N., and Gordon, N., eds., 2001. *Sequential Monte Carlo Methods in Practice*. Springer.
[6] Plauger, P., 2002. Thread safety in the standard c++ library. MSDN Library.