Extending Low-Cost Linux Computers for Education and Applications in Embedded Control and Robotics

Clark Briggs[1]
ATA Engineering, Inc., San Diego, CA, 92128

James Strawson[2] and Thomas Bewley[3]
Coordinated Robotics Lab, UC San Diego, La Jolla, CA, 92093-0411

This paper continues our Design Education paper from last year on the development of educational robotics kits based on credit-card-sized Linux computers. It describes in more detail the software architecture and some of the lessons learned working with embedded computers and the Linux open-source software environment. The theme is a consistent architecture that can be stretched from middle school through college to advanced field deployment applications. The desired outcome is insertion of advanced multithreaded control and software concepts into engineering education through applications that capture the interest and imagination of the next generation of mechatronics engineers.

I. Introduction and Background

The democratization of the tools necessary for the creation of novel, effective, and profitable robotic systems forms a tangible motivator for STEM education. The highly visible Maker movement is about tech-enhanced creativity, often of the Rube Goldberg variety, and has already captured the imagination of countless children around the world. However, the current new revolution in robotics is distinguished from this Maker movement by qualities that include analysis-driven design, minimalism, low-cost manufacturing, robustness, coordination, efficient multithreading, and vision-enabled situational awareness leveraging modern cellphone technology. New teaching tools aimed at the high school level and beyond to bridge the difficult gap from Maker to Roboticist are needed to retain the interest of students and motivate them toward rewarding STEM-based careers.

Modern applications in robotics combine several traditionally isolated disciplines, and this fundamentally interdisciplinary perspective required for the effective design and control of robotic vehicles is one that students can find engaging as early as middle school. In an effort to harness and shape the adoption of embedded computers in education, the authors have developed a unique modular system of vehicles, electronics, software, and educational materials, built upon open standards and Linux, to support education in embedded control and robotics.[1]

The authors began in 2012 with a laboratory supporting UCSD's fall-quarter senior-level embedded control and robotics course MAE143c, based on a two-wheel Segway-like vehicle that is inherently unstable. Today, the kit for this class is controlled with the popular BeagleBone Black (BBB) microprocessor coupled with a general-purpose robotics breakout board, or "cape," incorporating much of the supplemental electronics needed in typical robotics applications. The software library is written in C and runs under Debian Linux as a real-time multithreaded application. A single-board (lower-cost) integration of the essential components of the BBB and the robotics cape for embedded applications is also under development.

After updating our progress toward commercial availability of the educational vehicle kits introduced last year, we present the goals, approach, and progress for our robot control software library, and we conclude with a brief description of extensions beyond the classroom to industrial applications.

---

[1] Technical Director Robotics and Controls, ATA Engineering Inc., 13290 Evening Creek Drive, San Diego, CA 92128, AIAA Associate Member
[2] Doctoral student, UC San Diego, 9500 Gilman Drive, La Jolla, CA, 92093-0411
[3] Professor, UC San Diego, 9500 Gilman Drive, La Jolla, CA, 92093-0411

II. BeagleMiP, BeagleRover, and BeagleMAV

At the Consumer Electronics Show (CES) in Las Vegas in January 2016, our team will formally announce the forthcoming commercial availability of two of the core educational kits that we have been developing: BeagleMiP and BeagleRover (BeagleMAV is still under development and will be released at a later date), as well as the single-board integration of the essential components of the BBB and the robotics cape for embedded applications upon which these kits will be based. All of these educational vehicle kits are built and controlled with the same Linux-based software library and electronics, which incorporates an InvenSense IMU, H-bridges, voltage regulation and battery management, a real-time clock, and a host of standard connectors for commercial off-the-shelf (COTS) I2C, SPI, R/C, and GPS peripherals. These kits are designed to reach retail SKUs in the neighborhood of $200 each, readily facilitating *individual ownership* by students. (A "warm-up" kit introducing how to use the BBB to drive seven-segment displays, the I2C and SPI buses, PWM and H-bridges to drive individual DC motors and servos, and quadrature encoders to monitor shaft rotation will also be made available.) In the experience of the authors, individual student ownership and hackability of prototype systems is essential to foster experimentation and creativity, and result in an unmitigated intellectual investment by students that is simply not achievable via group projects or institution-owned educational systems.

BeagleMiP (Fig. 1) is the centerpiece of this effort. It is an unstable, nonminimum-phase "mobile inverted pendulum," with dynamics similar to the standard test problem (a.k.a. "plant") of a pendulum swinging freely from a cart as it moves along a track, but it is much more compact, economical, and fun. We have found it to be quite useful and versatile for teaching feedback control theory at the professional level, and we have implemented several different types of controllers to stabilize it, including classical (SISO) control strategies in the successive loop closure framework, state-space control strategies, and adaptive control strategies. BeagleMiP is about 20 cm tall and 8 cm wide, with 3-D printed chassis and bumper components. It is rather simple to assemble, with only three small wire bundles to connect the two motor and Hall-effect-encoder pairs and the LiPo battery to the cape.

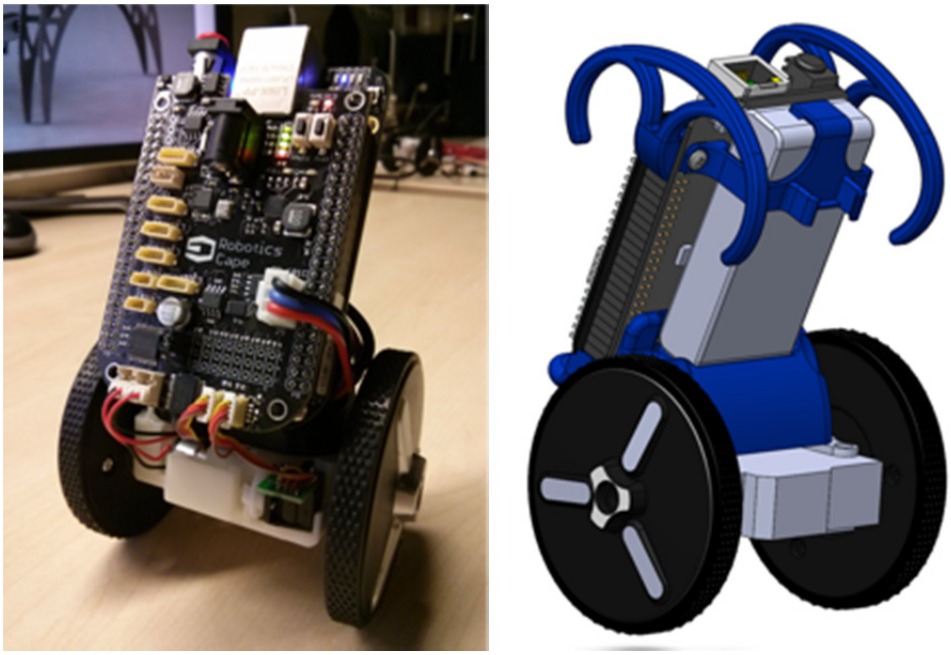

Figure 1. The BeagleMiP kits built, modeled, programmed, and stabilized by students in MAE143c at UCSD.

BeagleRover (Fig. 2) is a four-wheel-drive rover characterized by "extreme four-wheel steering," with each wheel capable of being turned independently (by servos) 120 degrees, which facilitates unique maneuvers such as turning in place and sideways translation for parallel parking in very small parking places. Four-wheel steering is a kinematically fascinating problem that is straightforward for students to visualize but challenging to coordinate well

in software. Since BeagleRover is dynamically stable, it is the logical starting point for makers as they study and work with this robotics programming environment. In addition to having a stable driving mode, BeagleRover is also capable of balancing like BeagleMiP on any of its four sides, allowing students using this kit to explore the stabilization problem.

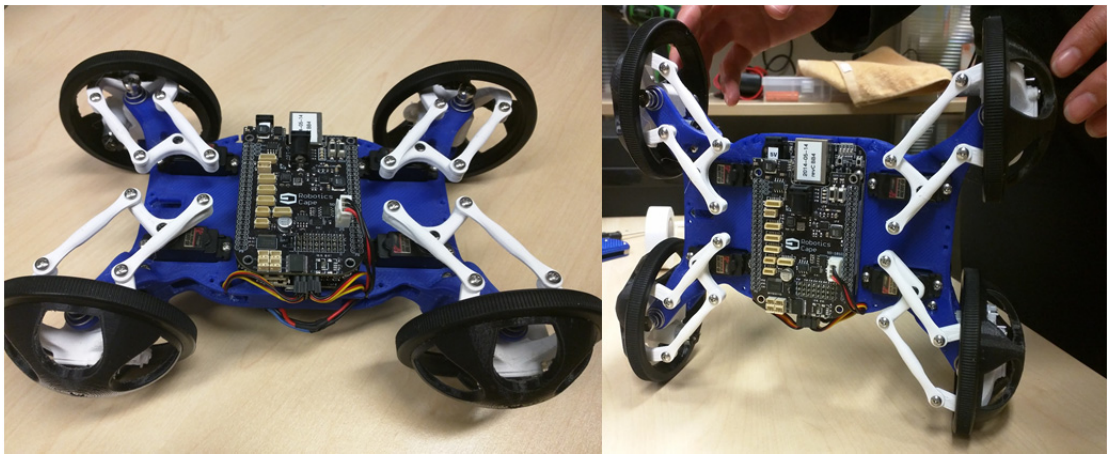

Figure 2. BeagleRover is characterized by four-wheel drive and extreme four-wheel steering.

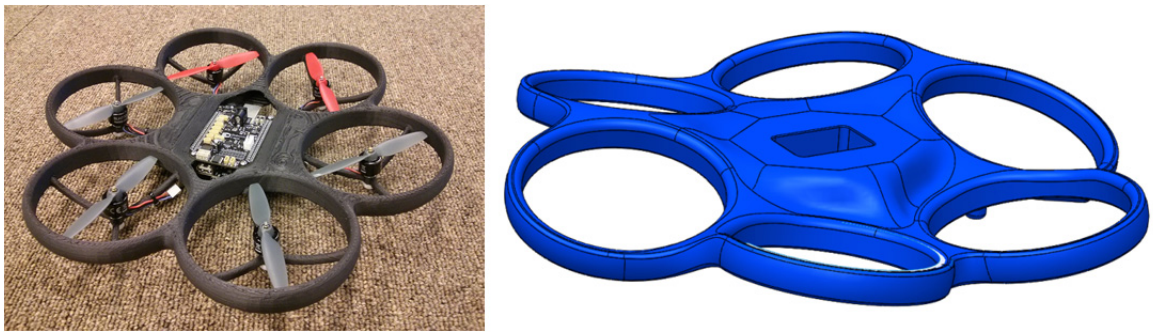

Figure 3. BeagleMAV, built and controlled with the same electronics and software library as BeagleMiP.

BeagleMAV (Fig. 3) is a multirotor micro air vehicle in "hexacopter" configuration which is popular amongst hobbyists, as well as a versatile platform for airborne photography and scientific measurement. Students working with BeagleMAV focus on stability augmentation systems, handling qualities, and automatic pilots. Our recent work on BeagleMAV has focused on exploiting the extra degrees of freedom provided by using six rotors instead of four: our recent improvements to this design, to be reported separately, tilt each of the six rotors to optimized angles, as illustrated at right in Fig. 3, to provide the capability of generating *direct force* in the lateral directions *without banking*, to improve the station-keeping capability of the vehicle while operating as a stable camera platform.

The monocoque frame used by BeagleMAV is unique in that it can be 3-D printed in one piece using low-cost fused deposition modeling (FDM) printers commonly available in the 3-D printing hobbyist market. Designed through careful finite element analysis (FEA) modeling and iteration, the frame is extremely light at 137 grams and yet resilient enough to survive bumps against walls while flying indoors by novice pilots. This strength-to-weight ratio can be attributed primarily to three factors. First, the use of the BBB and cape as structural components as well as circuit boards takes advantage of the stiffness of their fiberglass construction. Next, the monocoque structure itself means that there are very few individual components for the user to assemble, and thus very few metal fasteners are used. Finally, production with 3-D printers enables a unique hollow monocoque structure that derives strength from its geometry instead of relying on expensive materials such as carbon fiber.

Well-documented educational reference solutions will be included with all three platforms. For each platform, the documentation will come in two volumes, the first designed to be comprehensible and compelling prior to college, and the second focusing on the more detailed and precise analysis and design that can be performed once the college-level subjects of 3-D kinematics and dynamics, signals and systems, linear circuits, classical control, and state-space control are mastered. Thus, though the study of these dynamically interesting systems is satisfying at the maker level, it prompts students to ask additional questions that compel them to study physics, math, electronics, and other STEM-related subjects much more deeply. That is, BeagleRover, BeagleMAV, and BeagleMiP form a family of inexpensive test vehicles that physically embody a rich variety of advanced topics in robotics, kinematics, dynamics, control, and multivehicle coordination that a student may grow with through high school, college, and beyond.

III. Driven by Commercial Embedded Computers

A key challenge in modern-day robotics is keeping up with the rapid growth of processors, languages, and operating systems that could be useful in this field and deciding which emerging technologies to adopt and which to skip over. Many of the initial prototypes developed in our lab are programmed in C on low-cost Linux boards like the BBB, which prove to be much more flexible and extensible than similarly priced Arduino boards (as the Arduino "sketch" environment proves to be much more limited in its multithreading capabilities). That said, some repetitive simple tasks like brushless motor control are often best solved with dedicated daughter boards built around low-level processors like PICs, connected to much more powerful cellphone-grade processor boards which solve the higher-level tasks like vision-based situational awareness. Our desire to stay current with the powerful vision-based software libraries currently being developed for augmented-reality cellphone games motivates us to adopt nontraditional languages for high-level control system implementation, such as Java on Android. Our successes in reaching the mass market with millions of units of various toys, like MiP, in partnership with WowWee motivate us to deploy feedback algorithms to extremely low-cost microprocessors, like the ARM Cortex M0, with multithreading tools built on simple real-time operating systems (RTOSs) such as Keil RTX. Also essential to this effort are the user-side applications driving these robotic vehicles, with smartphones, tablets, and laptop computers communicating over Wi-Fi and Bluetooth. Further, to achieve the required power efficiency, we must offload many tasks to the multiple heterogeneous special-purpose hardware units, like DSPs, PRUs, and FPGAs, that come on the new system-on-chip (SOC) electronics.

The range of skills required to determine and program the appropriate mix of processors, languages, and operating systems outlined above is daunting. What is desperately needed is a well-documented set of software libraries that simplify and clarify the task of porting codes from $50 Linux-based prototyping boards down to $1 ARM Cortex M0 chips, and up to powerful vision-based cellphone electronics. Though we (and others) are working on such software libraries, as outlined below, much work remains to be done.

IV. Educational Components of Embedded Computer Control

The vehicle kits support several topics in dynamics and control such as discrete-time control and stability augmentation of continuous-time stable and unstable systems. Roboticists also need implementation techniques such as PCB design and fabrication, 3-D printing, and 3-D CAD. At the higher levels of study and effective application to real-world problems, there are topics in multivehicle coordination, navigation, and vision.

The UCSD embedded control and robotics course begins with simple exercises in "touching the world" with the BBB. The exercises include controlling LEDs and motors and reading sensors such as the gyros and accelerometers. These are core mechatronics topics. Other computer science topics are represented in the MIP balance software, including real-time priorities, multithreading, network message handling, and handling faults. These software areas are the subject of the subsequent sections.

V. The Core Robotics Cape Library and Applications

In this section, we overview the functions provided in the robotics cape library. The library provides a set of functions to allow easy programming of robotics-oriented tasks for our target audience of mechanical engineers and students not yet familiar with embedded systems. There are over 25 example programs and a user manual included with the robotics cape installation package. Table 1 provides a list and brief description of these functions.

The library functions cover setting up and using hardware such as GPIO, PWM, H-bridge motor controllers, servos, and the user I/O devices such as LEDs and buttons on the cape. A few of the architectural features are presented in the remainder of this section.

Table 1. User functions in the robotics cape library.

Initialization	
`int initialize_cape();`	Set up hardware (GPIO, PWM, ADC). Set `runState` to `PAUSED`. Create lock file in `/tmp`. Set `Ctrl-C` handler.
`int cleanup_cape();`	Close hardware. Set `runState` to `EXITING`.
Motors, Servos and Encoders	
`int enable_motors();`	H-bridge IC to enabled.
`set_motor(motor, duty);`	Set `motor` to normalized valued `-1.0` to `1.0`.
`int disable_motors();`	Stop all motors by putting H-bridge IC in low-power mode.
`int kill_pwm();`	Set all motors to `0.0 duty`.
`send_servo_pulse_normalized(ch, input);`	Specialized control of servos by pulse width.
`send_servo_pulse_us(ch, us);`	Specialized control of servos by normalized `duty 0.0` to `1.0`.
`set_encoder_pos(ch, value);`	Set an encoder to zero or an offset value.
`long int get_encoder_pos(ch);`	Read and encoder.
User Interface	
`int setGRN(uint8_t i);`	Set the green user LED to on or off.
`int setRED(uint8_t i);`	Set the red user LED to on or off.
`int getGRN();`	Read the green user LED value as on or off.
`int getRED();`	Read the red user LED value as on or off.
`int set_pause_pressed_func(int (*func)(void));`	Set Pause button down response function.
`int set_pause_unpressed_func(int (*func)(void));`	Set Pause button up response function.
`int get_pause_button_state();`	Read the Pause button up or down state.
`int set_mode_pressed_func(int (*func)(void));`	Set Mode button down response function.
`int set_mode_unpressed_func(int (*func)(void));`	Set Mode button up response function.
`int get_mode_button_state();`	Read the Mode button up or down state.
Managing the Battery	
`float getBattVoltage();`	Read the battery voltage.
`float getJackVoltage();`	Read the barrel connector voltage.
`int get_adc_raw(int p);`	Read one of the seven ADC channels.
`float get_adc_volt(int p);`	Read one of the seven ADC channels normalized.
Inertial Measurement Unit	
`initialize_imu(sample_rate, orientation[9]);`	Set up the Invensense MPU-9150 IMU chip.
`set?GyroOffset(offset);`	Set the bias offset where ? = X, Y or Z gyro.
`int loadGyroCalibration();`	Load the prebuilt offset file `/root/robot_config/gyro.cal`.
`set_imu_func((*func)());`	Set a function to be called whenever the IMU interrupt occurs.
Remote Control	
`int initialize_dsm2();`	Enable RC input from DSM Spektrum or Orange transmitters/receivers.
`int is_new_dsm2_data();`	Poll for new input.
`get_dsm2_ch_raw(int channel);`	Read a channel without normalization.
`get_dsm2_ch_normalized(channel);`	Read a channel normalized by `/root/robot_config/dsm2.cal`
Start Up	
`Auto_Run_Script.sh`	Automation to run a predetermined program after boot for connectionless usage.

The library uses a global state machine for controlling the robot. The states are `UNINITIALIZED`, `RUNNING`, `PAUSED`, and `EXITING`, and the state variable is accessed through getter and setter functions. During initialization via `initialize_cape()` the run state is set to `PAUSED`, and after `cleanup_cape()` it is set to `EXITING`. The user can set it to `EXITING` to cause the robot to shut down. The library also catches the standard Linux `Ctrl-C` signal and sets the run state to `EXITING`.

Control of hobby servos and brushless speed controllers is done with pulse duration modulation whereby position or speed set points are updated by sending a pulse between 800 µs and 2200 µs wide over a single signal wire. To demonstrate usage of the complex hardware units on the new SOCs, the library uses the BBB's Sitara Programmable Real-Time Unit (PRU) to do the timing for up to eight simultaneous servo signals. This takes the burden of tight low-level pulse timing off the general-purpose ARM core and allows simple implementation of servo-specific functionality such as cancelling signaling in the event of a software crash.

Robots commonly use rotary encoders, and our ground vehicles such as MIP have encoders on each wheel. The BBB includes specialized enhanced quadrature encoder pulse (EQEP) hardware to read these wheel encoders, but the EQEP kernel module in the Linux distribution is broken and has not been fixed. After successfully figuring out how to hack the source into our own kernel module, we turned to raw reading of the hardware registers using `mmap()` as a far simpler solution. While there are dangers and vulnerabilities to this approach, it is significantly faster. For example, reading the quadrature encoders is roughly two orders of magnitude faster. Toggling a GPIO pin using file I/O to `/sys/class/gpio` took 58 µs, but writing directly to hardware registers with `mmap` takes only 0.8 µs.

The cape provides two user buttons for quick robot control such as pausing the system or changing drive modes. While the buttons can be programmed by the user to do anything desired, they are labeled PAUSE and MODE to suggest these most common use scenarios. The library registers these button presses with the Linux input events subsystem and sets up the hardware debounce in the cape device tree fragment. There is a special thread that is spawned to catch the input events by reading `/dev/input/event1` and dispatch the user-defined key down or key up functions. In the typical robot example, the PAUSE button down event sets the run state to `EXITING`.

The cape comes with an Invensense MPU-9150 nine-axis inertial measurement unit, a popular sensor among hobbyists and roboticists. This sensor contains sets of three orthogonal accelerometers, gyroscopes, and magnetometers used to estimate the robot's orientation in space. Furthermore, the MPU-9150 contains a digital motion processor (DMP) which runs digital low- and high-pass filters on the accelerometer and gyroscope signals, respectively, in addition to estimating orientation. The fused sensor results can be read directly over the I2C bus and then interpreted as either a Quaternion vector or as Euler angles. The robotics cape library configures the DMP to sample sensors at a constant rate and sets an interrupt routine to read the sensor data, offloading the timing of the discrete controller and its complex attitude computations to the DMP. When we eliminated our initial complementary filter based on raw gyro and accelerometer readings and used the DMP, we reduced the time to obtain our attitude estimate from 2 µs to 1 µs and virtually eliminated the timing jitter on the fast balance control loop.

The library also uses threads for R/C control and MAVLINK telemetry. Spektrum DSM2 and DSMX brand radios of up to nine channels as well as Orange brand DSM2 transmitters and receivers are supported. The R/C listener thread is attached to the serial port that serves the DSM connector on the cape and has special functions for binding, calibration, normalization, and decoding of the control streams. The robot example application can be controlled with either brand of R/C equipment. The telemetry thread uses a MAVLINK sender that supports heartbeat and common message formats such as attitude reporting, and is compatible with common hobbyist receiver applications.

Our robots don't always start up and run with a console. The library uses standard Linux services to run a prescribed program after boot. `Auto_Run_Script.sh` is a bash script which lives in `/root`. During the installation process, the user is asked if they would like to automatically start any of the included examples like `balance`, `drive`, and `fly` for the BeagleMiP, BeagleRover, and BeagleMAV kits, respectively, or activate their own robot program.

VI. Expanding to Larger Applications

To support graduate research and initial industrial applications, the robotic cape library and example programs have been expanded in several areas, laying the foundation for navigation, coordinated group behavior, and, ultimately, vision. This growth has shown the general application architecture, the choice of embedded Linux, and the low-cost microprocessors to be up to the task. The primary target application has been the deployment of a squad of autonomous vehicles into a human-denied environment for the purpose of information gathering. The authors'

candidate projects have been aerial surveying of large land areas, exploration of buildings in the early stages of a structure fire, and hurricane in-situ weather science measurements using autonomous balloons.

In this section, we look briefly at the impact on the software architecture of several application-level examples, communications based on Linux client-server patterns, and fault protection.

A. Multithreaded Multi-Rate Application

To illustrate a more complex application structure, the fast and slow loops of the robotic cape balance program were extended in several functional directions. Several minor capabilities distributed through the slow loop and several new capabilities were all added as new threads. See Table 2. A common thread structure is used where each thread has a state machine, a single timed loop, and a cleanup phase. The threads are all established by the main program, each with their own appropriate priority.

Table 2. Thread architecture.

Name	**Function**	**Priority**	**Rate**
main	Initialization	Normal	4 Hz (sleep)
balance	Fast loop	20 (Real Time)	200 Hz (IMU int)
steering	Heading control	17 (Real Time)	50 Hz (tight)
navigator	Path following	13 (Real Time)	50 Hz (tight)
faux gps	Position reading	1 (Real Time)	~200 Hz (NatNet)
logger	Data capture	1 (Real Time)	@client rate
fault protection	Tip & low battery	Normal	50 Hz (sleep)
ui	Button monitor	Normal	50 Hz (sleep)
commander	Console commands	Normal	Socket wait
power monitor	Gas gauge, battery	Normal	1 Hz (sleep)
faux lidar	Distance ranging	Normal	4 Hz (sleep)

Most threads implement a common structure such as Fig. 4. After initialization, a thread enters the “running” state and executes a loop with a sleep call at the end of its activity. If the global run state machine is closing, it falls out to perform its cleanup and exit.

```
printf("Steering controller is running.\n");
steeringState = runningSteering;
do {
   encoderCountsL = -(get_encoder_pos(2));
   encoderCountsR = get_encoder_pos(1);
   gamma =  2.0*3.14159/352.0*(encoderCountsR-encoderCountsL)/2.0/1.35;
   eGamma[0] = gammaRef - gamma;
   uGamma = .2*eGamma[0]+40.0*(eGamma[0]-eGamma[1])/50.0;
   usleep((int)(1000000.0*dt));
} while (runState == running);
printf("Steering controller closed\n");
steeringState = closedSteering;
return;
```

Figure 4. Typical thread: steering controller.

B. Logging

Capturing data from the onboard code can be problematic. The implementation needs to have minimal impact on the CPU load, memory footprint, and network traffic. While it is common to stream debugging print statements to

the remote console, this rapidly degrades system performance. To this end, a simple in-memory logging utility has been developed. As part of the normal application shutdown, the memory-resident data is written to the file system on the SD card in a neutral CSV text format. The data can then be analyzed back on the development system in any common spreadsheet tool.

Logging is initialized and cleaned up in the logging thread, but data collection happens in the using thread. The data structures are unique to the study need and thus vary widely. For example, when studying the timing and latency of the fast balance loop, data structures such as this were used:

```
struct datum {
    float  t1;
    float  t2;
    float  t3;
    float  sleepRequest;
    }datum;
struct datum *datumRec;
int numLogDatums;
int logCounter;
float tStartLogging;
```

Then at the bottom of the fast balance loop, prior to sleeping, the times collected during the loop are saved to the in-memory array.

```
if(logState == logging)
    {if (logCounter < numLogDatums)
          {
                datumRec[logCounter].t1 = fT1;
                datumRec[logCounter].t2 = fT2;
                datumRec[logCounter].t3 = fT3;
                datumRec[logCounter].sleepRequest = fSleepRequest;
                logCounter++;
          }
    else
          {
                logState = notlogging;
          }
    }
```

The logging thread allocates the memory array based on a user-defined number of datums (`numLogDatums`) and writes the collected data to a file in a custom CSV format after logging has finished.

Logging can be started after a given elapsed time (`tStartLogging`) or the using thread can set `logState` to `logging` or `notlogging` as conditions warrant.

C. Path Following

It is straightforward to extend the basic MIP balance routine to remote joystick-controlled driving and steering, but these must serve some purpose that contributes to autonomous exploration, with layers—such as navigation, path generation, and path following—to be realized on the way to the ultimate goal-oriented behavior. Quite a bit of infrastructure is needed to accomplish all these activities, and it has been difficult to find modest-sized tasks that can be completed as interim accomplishments. For example, a fully autonomous application will require vision, simultaneous localization and mapping (SLAM), and probably structure from motion (SFM). The authors chose to postpone those development activities, instead providing position knowledge and routing from the support servers. The robotics development laboratory is instrumented with an off-the-shelf motion capture video system which tracks reflective markers on the vehicles and publishes their positions and attitudes in network streams, and the onboard client software is implemented as if this were a GPS position reading. Similarly, to postpone development of navigation in complex and cluttered environments, routes are compiled into static maps. With knowledge of position and routes thus supplied, we have successfully developed path-following routines.

To follow a given straight-line path, the steering controller is given a commanded turn rate that will bring the vehicle closer to the desired line path. Several algorithms have been tried, but the current implementation projects a desired goal point a fixed distance forward on the path and computes a constant velocity turning rate to arrive there. Since the vehicle motion is imperfect, this steering command is recomputed at a 40 Hz rate, typically.

The simple line-following behavior has been augmented with some special maneuvers. For example, for cases where the vehicle finds a dead-end path and must turn around in a fixed space, a special turn-around-in-place maneuver was implemented. This was later extended to simple obtuse angle corner turns.

The navigation controller will need to make decisions about path choices at intersections. The routine that executes the path following has several choice functions, such as always choose left, always choose first, choose least turning angle, or choose largest turning angle. Building interiors often have multiple paths, complex intersections, and loops. The navigator can recognize and handle loops as they are discovered using several strategies. The algorithms have been developed and tested in simulation for injecting multiple vehicles into an unknown map, ensuring complete exploration and minimum routing return and exit. These have been partially tested in the lab with real vehicles. Two trials with MIP vehicles in the lab are shown in Fig. 5.

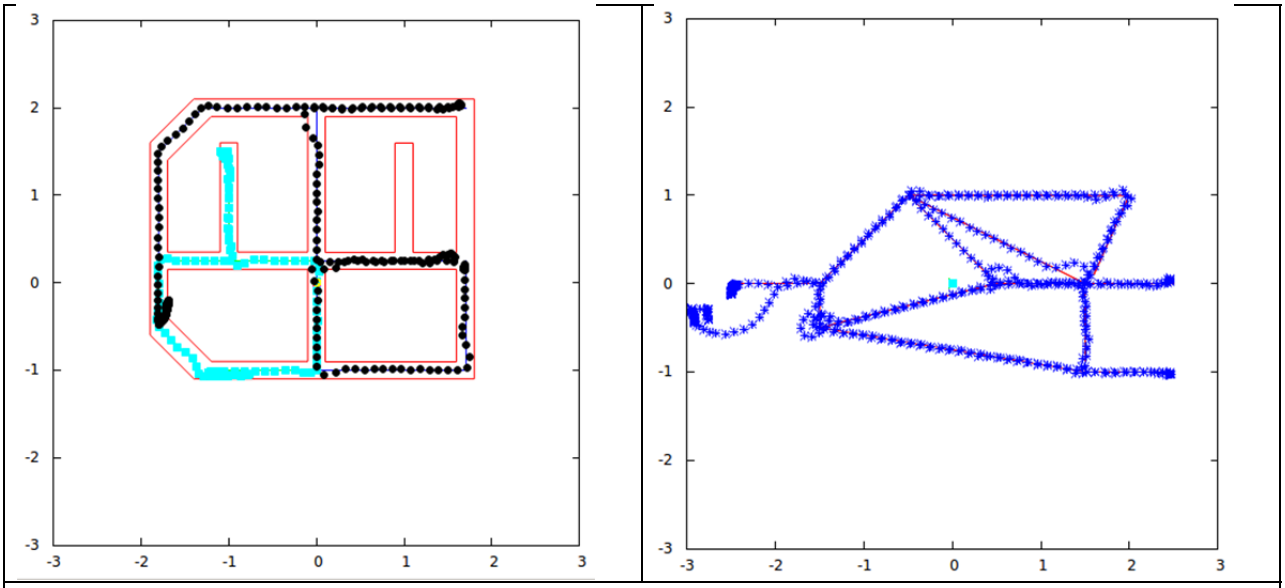

Figure 5. Left: two MIPs in a floor plan. Right: one MIP explores a complex maze with loops.

These views show the floor plan, walls, and other vehicles generated on the support equipment server. The vehicles themselves do not currently know about or sense obstacles such as walls and the other vehicles, but vehicle-to-vehicle communications, essential for map sharing for example, have been built. Each vehicle runs a `gabby` thread that broadcasts its position and an `eavesdropper` thread that listens for and collects information broadcast by the other vehicles. However, this work on group behavior was suspended and work on path following has moved outdoors, employing a newly built 1/10th scale R/C truck that uses GPS for position knowledge.

D. Client/Server Communications

One of the major benefits of using Linux is basing communications on the IP client and server architecture. While it is likely that the final field deployment will require some alternative communications media, all of the educational and most of the research objectives can be readily fulfilled using Linux networking.

For example, the command responder recognizes a variety of commands sent by the user from the controlling workstation or laptop. There are simple commands such as "`go`" and "`exit`," optional commands such as "`RC`" to enable remote joystick control, and complex commands such as "`sendPath 5`" and "`usePath 4`." The commands

can be implemented as binary or JSON text messages. The server-side command sender includes an expression parser to support preparation of complex commands.

The command receiver thread blocks on a socket read. If binary messaging is being used, the packet contains a type discriminator and a union of allowed message structures. The message type is used in a switch statement to then obtain the payload data structure. If JSON text messaging is being used, the message is first parsed by the JSON library, then the "`cmd`" field is used in the switch structure.

The position measurements from the motion capture system in the laboratory are implemented as a simulated GPS server. The onboard GPS client thread reads binary or JSON text messages that contain either "`position`" messages with "`x`," "`y`," and "`heading`" values or an "`exit`" message to shut down the connection. When driving outdoors, the GPS client reads standard NEMA GPS sentences over a UART from an external GPS module. A parser recognizes the sentence type and decodes each field, locating and saving the latitude and longitude values.

There are applications for several of the IP mechanisms. For example, the motion capture system uses multicasting to publish the locations of all the bodies in the test volume. The command server uses UDP broadcast when there are multiple client vehicles or TCP unicast when there is a single client vehicle. All of the server-to-vehicle network communications are done via Wi-Fi since the vehicles must run untethered.

E. Fault Protection

A common requirement in complex systems is to maintain some required level of fault tolerance. To illustrate the idea, a fault detection, reporting, and dispositioning mechanism has been developed. Fault protection implements issue tracking, where issues have a priority `(enum fpLevel{info, serious, fatal})` and are maintained on a list. Issue raisers call `fpRaiseIssue(enum fpLevel level, enum fpSubSystem subsystem, char * text)` to get the issue put on the list. Since issues are raised in the caller's context, the routine must be fast and non-blocking; it simply links them onto the end of the work queue for the fault protection thread, and the fault protection thread later picks up the issues from the list and dispositions them.

This fault protection architecture separates fault recognition from fault handling. Fault recognition occurs close to the code that controls the activity, but the fault response happens elsewhere. This also has the advantage of moving all the console chatter to one thread where it can be silenced by level and recorded.

Heuristics can be applied as appropriate to the issue level and the raising subsystem. As an example of a fatal event, when the user types `Ctrl-C (SIGINT)` on `stdin`, the signal is caught in the standard Linux fashion. This handler uses the issue registration process with a `fatal` class message about `Ctrl-C`.

```
void cleanup(int signo){
    if (signo == SIGINT){
           fpRaiseIssue(fatal, fpssFP, "FP: received SIGINT Ctrl-C");
    }
}
```

Then, in the fault protection thread loop, any fatal issue is used to change the run state machine.

```
if (theIssue->level == fatal)
{
    printf("Fault Protect has a fatal issue with text: %s\n",
           theIssue->issue);
    runState = closingRun;
    balanceState = closingBalance;
    fpState = interruptedFP;
}
```

General informational messages can be ignored or passed on to the console as they are dispositioned.

When the application closes, the fault protection thread saves the accumulated issue list to a file for the postmortem analysis and optionally prints them to the console.

Other issues caught in various threads include low battery voltage in the power monitor, commands received in the command monitor, sensor errors in the balance loop, and various informational progress messages about path-

following performance from the navigator. For vehicles that can be remotely controlled with an R/C transmitter, the kill switch is caught in the R/C listener thread and submitted as a fatal issue.

VII. Progress on Next Steps

While the BBB is a very capable computer, its single general-purpose CPU is easily overwhelmed with low-level control, network communications, and high-level application algorithms. Many of the emerging SOCs, like the Black, have additional special-purpose hardware that must be leveraged to achieve maximum performance (high speed at low power). Developing and managing software in such a heterogeneous computing environment is quite challenging. Our team has had some success in the use of the BBB's PRU to synthesize additional PWM channels (for servo and ESC control) and to perform encoder counting, but significant additional work in this direction remains. In particular, supporting vision requires accelerated image processing using libraries which have been ported onto special-purpose hardware.

The next-generation SOCs will have multiple general-purpose CPUs that need to be exploited with the Linux symmetric multiprocessing (SMP) capabilities. This will change the current thread/task priority assignment scheme in support of the mixed demands of low-level real-time hardware control and the application-level algorithms of, for example, navigation and group coordination.

A. Vision

To support vision, hardware-accelerated image processing is required. The authors are working with Qualcomm Corp., a major UCSD sponsor, to gain access to their SnapDragon SOC and its associated accelerated implementation of OpenCV called FastCV. One of our vehicles under development is a stair-climbing robot. To explore vision-based navigation, we have prototype vison algorithms that match a 3-D stair-step template to the camera image as the vehicle approaches a stairwell. These algorithms were developed on a desktop PC using OpenCV then ported to a cell phone that uses the SnapDragon SOC. The accelerated FastCV Hough line algorithm provides eight to twelve times speed-up over the general-purpose processor OpenCV implementation. Fig. 6 shows two images of the stairwell from the development environment and an image from the cell phone. The left image is the output of the Canny edge detector and the Hough line detector. The center image shows the stairwell template in blue and the matching Hough lines in green, after setting a suitable initial position. The right image is a screenshot from the matching algorithm working in real time at five frames per second on a SnapDragon cell phone using FastCV.

Figure 6. A camera snapshot (left) Canny edges, Hough lines (center), and cell phone screenshot (right).

B. Path Following Outdoors Using GPS

As mentioned above, the path-following algorithms were initially developed indoors using a motion capture system to provide position and heading knowledge. Since many of our vehicle applications do require covering significant distances outside, a $1/10^{th}$ scale R/C truck was fitted with a common hobbyist GPS module and a BBB computer and robotics cape. The GPS client thread in our software was modified to collect and parse the standard NEMA GPS sentences, looking for latitude, longitude, speed over ground, and course over ground.

Fig. 7 shows the truck successfully navigating around the first corner in the parking lot but missing the second.

Figure 7. Driving outdoors in the parking lot using GPS (work in progress).

The straight line segments represent the desired path, and the meandering line is the path driven. While this back-and-forth behavior is characteristic of the path-following algorithm, it is excessive. Like the indoor MIP, between GPS position updates the truck drives on wheel odometry. The MIP odometry works well at MIP scales of a few feet, because both steering and distance are well measured by the wheel encoders. The truck, however, has an encoder only on the rear axle and no direct steering measurement. The estimate of steering angle used apparently causes significant drift in the internal position estimate that grows over time. The vehicle is being modified to directly measure the steering angle to better investigate this issue.

VIII. Conclusion

The development of agile vehicles and their supporting embedded control systems in a manner that is accessible to students yet full-featured enough to support autonomous group behavior has been challenging. The UCSD controls class using the MIP has been very popular, both at UCSD and at the Jet Propulsion Laboratory in Pasadena, and several other schools are expected to adopt it in the near future.

References

1. Bewley, T., J. Strawson, and C. Briggs, "Leveraging Open Standards and Credit-Card-Sized Linux Computers in Embedded Control & Robotics Education," AIAA-2015-0801, 56th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 2015.