I was working on a project recently and it was the first one which was involved enough to make the sensor networking complicated. In the end, I think the communication was the bottleneck in terms of overall performance and I'm wondering how more experienced people would have solved this problem. This is a long read, but I think it's pretty interesting so please stick with it. The problem was to design an autonomous blimp capable of navigating an obstacle course and dropping ping pong balls into brown box targets. Here goes:

Sensors

- 4D Systems uCAM-TTL camera module - UART interface
- HMC6352 Digital Compass - I2C interface
- Maxbotix Sonar ez4 - 1 pin analog interface

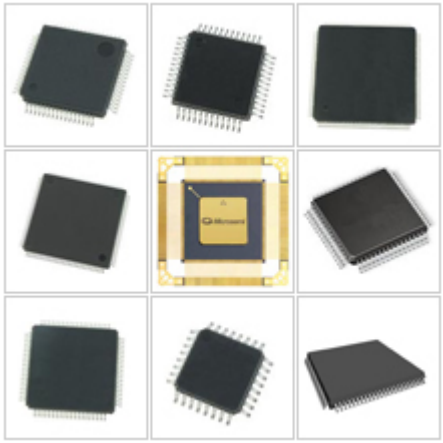
Actuators

- 2x L293D motor drivers (connected to simple hobby motors) - These were used to drive 6 motors bidirectionally. They required PWM inputs in order to vary the speed. Now 3 of our motors were always doing the same thing (the ones that controlled up/down movement) so they only required 2 PWM outputs from our controllers to control all 3 motors. The other 3 motors which controlled lateral movement all needed individual control (for omni-directional movement) so that was another 6 PWM outputs required from our controllers.
- Servo motor - PWM interface

Controllers

For reasons that will become clear later, we ended up using 2x ATMEGA328Ps. We used an Arduino Uno to program them (we didn't have access to an ISP) but we fab'd a custom PCB so we didn't have to use arduino boards since that would just add unnecessary weight to our blimp. As for why we chose the ATMEGA328P, I was very familiar with the arduino environment and I think that made the code development much quicker and easier.

Communication & Processing

- 2x Xbee Basic
- 2x ATMEGA328P: http://www.kynix.com/Detail/551814/ATMEGA328P.html
- 
- Desktop computer running C++ w/ openCV

So as you can tell from the camera module, most of our project relied on computer vision. The blimps could only carry so much weight and we didn't feel comfortable implementing computer vision on a microcontroller. So what we ended up doing was using XBee's to relay the image data back to a desktop computer. So on the server side we received image data and used openCV to process the image and figure stuff out from it. Now the server side also needed to know height information (from the sonar) and compass information.

The first wrinkle was we were not able to have the camera controlled by a microcontroller for a couple reasons. The main issue was internal memory on the uP couldn't handle storing an entire frame. There might have been ways around this through clever coding but for the purposes of this question let's pretend it was impossible. So to solve this problem, we had the server side send camera commands through the XBee transceiver and the XBee receiver (on board the blimp) had its output wired to the camera's input.

The next wrinkle was that there are not enough PWM's on a single ATmega328P to control all the motors BECAUSE the I2C interface uses one of the PWM pins (damn them...). That is why we decided to use a 2nd one. The code actually lent itself perfectly to parallel processing anyway because the height control was completely independent of the lateral movement control (so 2 micros was probably better than one attached to a PWM controller). Therefore, U1 was responsible for 2 PWM outputs (up/down) and reading the Sonar. U2 was responsible for reading the compass, controlling 6 PWM outputs (the lateral motors), and also reading the Sonar. U2 also was responsible for receiving commands from the server through the XBee.

That led to our first communication problem. The XBee DOUT line was connected to both the microcontroller and the camera. Now of course we designed a protocol so that our micro commands would ignore camera commands and camera commands would ignore micro commands so that was fine. However, the camera, when ignoring our micro commands, would send back NAK data on its output line. Since the command was meant for the micro we needed someway to turn off the camera output to the XBee. To solve this, we made the micro control 2 FETs which were between the camera and XBee (thats the first FET) and also between U2 and the XBee (thats the second FET). Therefore, when the camera was trying to send info back to the server the first FET was 'on' and the second FET was 'off'. Unfortunately there appeared to be some cross talk with this method and sometimes when the server was trying to receive height data for example, it would read a NAK from the XBee.

So to give you an idea of how this worked here are a few examples:

1. Server requests a picture - PIC_REQUEST goes through XBee and arrives at U2 and camera. U2 ignores it and camera sends back image data.
2. Server just finished processing a picture and is sending motor data to tell blimp to turn right - MOTOR_ANGLE(70) gies through XBee and arrives at U2 and camera. U2 recognizes as a micro command and thus turns off Camera's FET (but perhaps the camera already responded with a NAK?? who knows...). U2 then responds to the command by changing motor PWM outputs. It then turns Camera's FET back on (this was the default setting since image data was most important).
3. Server realizes we've come to a point in the obstacle course where our default hover height now needs to be 90 inches instead of 50 inches. SET_HEIGHT goes through XBee and same thing happens as in example 2. U2 recognizes the SET_HEIGHT command and triggers an interrupt on U1. U1 now comes out of it's height control loop and waits to receive serial data from U2. That's right, more serial data. At this point the U2's FET is on (and camera's FET is off) so the server receives the height that U2 is also sending to U1. That was for verification purposes. Now U1 resets its internal variable for height2HoverAt. U2 now turns off it's FET and turns the camera FET back on.

I definitely left out a good amount of information but I think that's enough to understand some of the complications. In the end, our problems were just synchronizing everything. Sometimes there would be data left over in buffers, but only3 bytes (all our commands were 6 byte sequences). Sometimes we would lose connection with our camera and have to resync it.

So my question is: What techniques would you guys suggest to have made the communication between all those component more reliable/robust/simpler/better?

For example, I know one would've been to add a delay circuit between the on board XBee out and the camera so that the micro had a chance to turn off the camera's talk line before it responded to micro commands with NAKs. Any other ideas like that?

Thanks and I'm sure this will require many edits so stay tuned.

Edit1: Splicing the camera's UART data through one of the micros did not seem possible to us. There were two options for camera data, raw bit map, or JPEG. For a raw bitmap, the camera just sends data at you as fast as it can. The ATmega328P only has 128 bytes for a serial buffer (technically this is configurable but I'm not sure how) and we didn't think we'd be able to get it out of the buffer and through to the XBee fast enough. That left the JPEG method where it sends each package and waits for the controller to ACK it (little handshaking protocl). The fastest this could go at was 115200 baud. Now for some reason, the fastest we could reliably transmit large amounts of data over the XBee was 57600 baud (this is even after we did the node/network pairing to allow the auto-resend capability). Adding the extra stop in our network (camera to micro to XBee as opposed to just camera to XBee) for the micro simply slowed down the time it took to transfer an image too much. We needed a certain refresh rate on images in order for our motor control algorithm to work.