SUMMARY

This is a progress report on an "open" CAN bus wheelchair controller system that I've been working on. Luckily I am now retired so I am able to devote a lot of uninterrupted time to a project that is much more involved than anything I have done before, with the exception of my daughter Rachele's head switch and gaze operated multilingual communications/computer control system which only reached its current state after years of work.

Having been involved with wheelchair design for over 20 years, I am well aware of the safety issues involved; after all, a wheelchair user can not "bail out" if something goes awry. Indeed, creating a "bullet proof" system is a major part of the work involved. For a geneticist working on a project that really needs the skills of an ME, an EE and a programmer this is decidedly non-trivial.

One of the satisfying, and at the same time worrisome, aspects of this is that I learn something new just about every day. For example, common practice is to protect a semiconductor switching circuit from relay coil kickback by putting a reverse-biased diode across the coil. This indeed will protect the driver, and it is a good way to do things when dealing with signal-level relays (or AC relays in general), it is, however, a relay-killer if used on high current DC relays. I only discovered this when a footnote on a spec sheet led me to these Tyco application notes. If you are dealing with controlling more than mA currents, they are a must read.

http://relays.te.com/appnotes/app_pdfs/13c3311.pdf

http://relays.te.com/appnotes/app_pdfs/13c3264.pdf

So what's worrisome about learning all these new things? What's worrisome is the question: What other equally critical factors exist that I should know about that, because of my ignorance, I don't even realize that I should study?

In the end, I expect to need a lot of testing at high power in a high noise environment before the design is really "final". And that's a problem for me: pushing everything to its limits requires equipment, money and time I don't (and won't ever) have, and I've yet to decide what level of risk I'm willing to tolerate in a system that's not been "tested-to-destruction". Hence, in both the hardware and software I'm trying to use a "belt-and-suspenders", fail-safe and safe-fail, approach and that's why the more people who are willing to invest some time in double-checking and questioning what I've done the more comfortable I will feel.

At this point, I have a three-node net running at 1M bits per second without dropouts (compared to the usual WC controller rate of 100k bps or so). Transmission rate will be set lower on the chair after testing at 1M to give a nice safety margin. During testing, a PC will also be connected to the motor controller to start its script each time it's powered up. Once I'm sure that the script doesn't have any bad bugs, it can be set to autorun on startup. The PC will have NO role in actual operation - no PC operating system is to be trusted EVER with driving a wheelchair!

This will be a very lengthy message. If you'd prefer to read it at leisure, I've attached a pdf version. There is also a zip file of the hardware (WC CAN circuits.zip) with schematic and pcb designs as pdf and DesignSpark files, and a zip file of the program files (WC CAN programs.zip).

PROJECT DESCRIPTION: HARDWARE

CAN nodes:

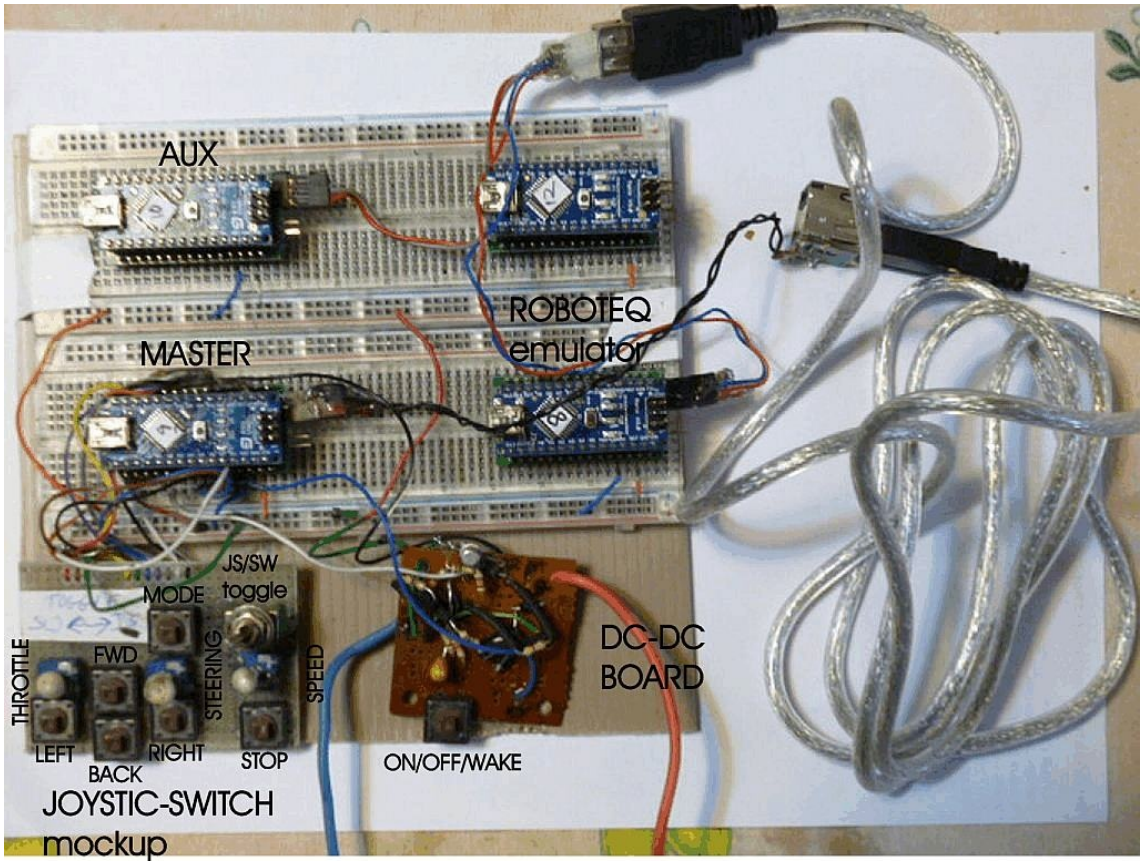
The system will have four CAN bus nodes, though more could be added if other functionality is needed:

(1) the MOTOR CONTROLLER node that handles the high current needs of the motors and some medium-current digital outputs. I am designing around the characteristics of the Roboteq HDC2450, but relatively little work would be needed to adapt this to other controllers. Some external hardware is also needed: high current safety disconnect relay, SPDT reed relay for power to the Roboteq's microprocessor, pre-charge resistor, regeneration bypass diode (in case the safety relay opens), voltage clamping on the motors (if not built in to the motors themselves), relays or semiconductor H-bridges for actuators etc.

(2) the MASTER node that collects information from an analog joystick and/or switches, and does the arithmetic and logic to turn that into CAN messages to the Roboteq for controlling the motors, brakes etc. Because the Roboteq does not treat CAN the same way as it treats analog, pulse or serial input, all of the calculations for scaling the joystick, setting dead band, implementing a speed pot (if desired), applying different degrees of exponential curving and so on must be done here. This is actually similar to Dynamic DX/DX2 systems in which the power module only has motor programming, and the master computer is in the remote. Of all the driving calculations needed, only mixing and motor compensation are handled in the Roboteq node. MASTER also traps errors, collects information from the Roboteq and Aux nodes, and sends this information on to the DISPLAY node. A toggle switch is used to go between joystick and switch input, and momentary contact switches are read to give forward, reverse, left, right, mode (driving, seat, lights), power On (or Wake)/Off, and STOP.

(3) the AUX node. Although the Roboteq has a number of 1A digital outputs, there are not actually enough to do everything one might want to do: separate left and right brakes, seat actuators (1, 2 or 3), lights. Moreover, I don't want to have separate On/Off switches for different modules so we need some way to control power to the Roboteq via the CAN bus. These functions will be handled by AUX node. AUX can also handle a shock sensor &/or other stability sensor (such as I now have on Rachi's chair), and speed slow-down &/or inhibit sensors on the seat actuators. (This is where one could add gyros, compass, even a full inertial navigation system if one wanted too - I don't, but a Nano has enough guts for all of this.)

(4) the DISPLAY node. This node receives messages from MASTER to display them on a small TFT screen and to log them on a micro SD card. Graphics will show which way to move the joystick for different seat movements or light settings, an indicator of whether in joystick or switch-driving mode, present voltage and Amp Hours consumed since last charge.



MASTER, AUX and DISPLAY each contain: an Arduino Nano (or open-hardware copy) microprocessor, a CAN controller/transceiver (Microchip 2151 + Microchip 2551) board with the same footprint as the Nano plus two extra rows of headers to make external connections easy, and a 500 mA DC-DC converter to provide 5V for the electronics. The DC-DC board has the same footprint and headers as the CAN board, so all three can be stacked or mounted separately to make best use of space. The Nano board is 18 mm X 43 mm (0.7" X 1.7") , and the other two are 23 mm X 43 mm (0.9" X 1.7"). There's a decoupling capacitor on top of each chip, a tantalum filter capacitor on each 5V power input, and an aluminum electrolytic on the 24V input to the DC-DC board.

Although the Nano is quite small, it is a full development board and includes a USB-serial chip and mini-B USB connector for programming it (the USB-serial chip goes to sleep if the USB is not connected). For the moment, for the MOTOR CONTROLLER node I have programmed a RoboteqEmulator using the same Nano+CAN board hardware, but eventually that program will have to be translated from Arduino C++ to Roboteq MicroBasic.

Programming of MASTER and RoboteqEmulator nodes is essentially complete and a first-draft of AUX is running, but I've yet to begin on DISPLAY (partly because the screen I want to use is on seemingly-perpetual back order; I guess it's pretty popular).

"Green" CAN boards have been made and I have four nodes working with these. One trace is different in the "final" design, and I screwed up the silk-screening on the green boards, but if any of you are interested in trying things out, I have 12 bare boards to give away: you need at least 2 to have a working bus, 3 to run all three current nodes. So six of you can each have a pair, or four of you can have 3. Send me an e-mail if you want some. You will have to mount SMD

components on both sides, open one trace and add one wire jumper. I do my SMD soldering with an ordinary toaster oven and a watch with sweep-second hand, and position the components with very fine tweezers and a stereoscope borrowed from the University of Siena - though a decent magnifying glass would also work.

The design of the DC-DC converter board has been finalized, but so far all I have is a hand-made prototype. The converter used with MASTER is on/off controlled from the Nano, while the other two will be always on. The board is the same, but many components are simply left out on the 2 simpler boards. One momentary-contact push button serves for ON, OFF and WAKE FROM SLEEP. When MASTER is turned off, it first sets all Roboteq outputs to zero, turns off power to the Roboteq (via a message to AUX), then puts all of the CAN boards into sleep mode, puts the AUX and DISPLAY processors into a PWR_DOWN sleep mode, then is finally itself powered down. Turning ON is the reverse. Inactivity for a user-set time causes everything to go to sleep (but not to power down) to save power, but gives a quick (no more than a few milliseconds) wake-up response with the same pushbutton.

Rachi's system will have one additional box, but it's not a CAN node. It's a multiplexer to allow the same head and foot operated switches to control the chair and control her computer. On her present chair, this is, for historical reasons only, a 12 V through-hole component board, but the new one will be 5V SMD. Connected by DB9 to MASTER, it will receive 5V from the MASTER node OR'ed with 5V from a PC USB port so that it will power down when neither the chair nor computer are being used.

I have my three nodes communicating very nicely at 1M bits-per-second (chair systems are generally much, much slower), but some changes might well be needed when there's all the "noise" of a real installation. For the most part, I'd prefer to suppress noise at the source rather than try to filter it out, but some filtering of CAN-H and CAN-L might be needed and there's no room for that on the boards as designed. I am trying to avoid that, however, as filtering the CAN lines can cause bit timing problems worse than the problems caused by noise - Dynamic never got their DX-5SW module to work reliably for this reason and pulled it from the market. One thing I might have to do is add a couple resistors and a connection to the 2.5V Vref output on the MCP2551 to avoid common mode voltage drift, but, again, I'd rather try first to avoid ground loops (which would be the cause of the drift) rather than hide the effect.

Cabling and connections:

This has been a hair puller, and being almost bald that's unfortunate. Dynamic's proprietary cables have 2 heavy 24V wires that can handle as much as 12 Amps over short distances, and a small, shielded twisted pair for CAN. I have not been able to find a stock cable of that type, so would buy cables from Dynamic were it not for the fact that their jacks are OEM so can't be bought. CAN bus specifies 120 ohm impedance twisted pair with 120 ohm termination resistors in the end nodes, but one can get away with large deviations from this standard, as Dynamic and others do. For example, Dynamic allows, indeed pushes, a "star" rather than "daisy-chain" configuration of the bus (to limit voltage drop on the 24V wires) so there can't be termination resistors at the two ends because there aren't two ends, nor can they use distributed termination among all nodes because any two wheelchairs may have different numbers of modules. They get away with this because impedance mismatch doesn't much matter on a short, slow speed network (though I suspect it contributed to the timing problems on the DX-5SW). For now, my compromise looks like this:

(1) Daisy-chain bus with 120 ohm terminators at the ends, separate high-amp wiring to the Roboteq and any other modules that need high power, and only enough 24V capacity on the CAN cables to power the electronics.

(2) The best USB-2 cables available (e.g. Belkin pro or gold) have 20ga power wires, a 25ga twisted signal pair, inside foil shielding with drain wire for the signal pair and outer braid shielding of the cable. USB impedance is 90 ohms instead of CAN's 120, but the bus is very short (well under a small fraction of a wavelength; CAN is good to 1 km with proper impedance match and we're talking about a few meters at most - BTW, USB2, because it is much faster, is good only to 5M) so I don't think this will be a problem.

(3) 5 pin XLR latching connectors, insulated from ground but with continuity via the shells for the outer braid and a separate pin for the inner shield. Inner foil grounded at only one end (to avoid loop currents) and the same for the braid, though I might want to connect that to ground via a capacitor to avoid any DC offset, losing some shielding effect. Using 5 pin connectors also makes sure that a WC charger plug doesn't get plugged in. With a 7.5 A/pin rating, 5 pin XLR is many-fold more than adequate for powering the electronics.

I have one misgiving about using XLR plugs rather than the more-fragile and non-locking USB connectors. USB connectors have longer contacts for the power lines than the signal lines - it's part of what makes them "hot swap". This ensures that voltage can not be put on a FET gate before it is powered up; doing that can destroy the gate. As that's a nice protection, I may end up with USB-A plugs anyway, though I'd still have to make up my own cables as the 20/25 ga. cables come in a limited selection of lengths, USB-A male to USB-A male is not USB spec so not available in these cables (but would help keep someone from plugging the wrong stuff in), and cables with various right angle plugs (which would keep them from hitting things) that I've found, though good quality, are 25/28 gauge. I've made up right angle USB cables for Rachi's computer (after cracking a mother board when a sticking-out connector got clobbered), so know I can do it, but it's a tedious task and the results aren't "pretty".

SOFTWARE

CAN protocol:

A standard CAN frame consists of an identifier (11 bits) and up to 8 bytes of data. The CAN standard includes a number of features, both in how things are coded and in the physical layer (CRC check, bit-stuffing timing check, repeated transmission until receipt is acknowledged) to make sure that messages arrive intact. Because a wheelchair is a life-critical device for a user who may not be able to "bail out" if things go awry, my programming adds a couple further layers of protection. My messages contain at most 4 bytes of data and the other space is used for the bit-wise complements of these. A receiving node first does the built-in CAN checks and filters for proper ID before putting it in a message buffer (the MC 2151 has two buffers with separate masking/filtering). My (AUX, DISPLAY or Roboteq script) program then compares the data and their complements to make sure that the information is correct and then sends a confirmation message back to MASTER. If the original message was an information request (e.g. battery amps), the confirmation is the requested information itself. If the original message was a control or configuration command (such as a motor command), the confirmation message is the same data (and complements thereof) with the ID changed to show that it is the confirmation. MASTER

again does the data/complement check, traps any errors and does not proceed on unless a proper confirmation is received.

For my nodes, the 11 bit ID is divided into three fields: 2 bits to say whether this is a command, information or configuration message; 6 bits for the name of the message using the same names as Roboteq uses in their Constants.h file with a few additions for functions the Roboteq doesn't have; and 3 bits that say from whom the message is coming or to whom it is going. Each of my nodes filters incoming messages to accept only the ones intended for that node.

The Roboteq, however, is a special case. It can be set to either (1) accept all messages, or (2) accept messages from only one node. Case (2) would be OK but for the fact that Roboteq has tied up 7 (of the 11) ID bit to specify the one node to listen to which means that my division of the ID into fields wouldn't work. Hence, the Roboteq will be set to accept all messages and its script will then have to discard the irrelevant ones. Workable, but slower than proper masking and filtering. I've tested this by setting AUX to accept all messages then checking whether it responds properly to its intended messages (YES) or whether the ack from AUX interferes with RoboteqEmulator seeing, and confirming, its intended messages (NO, they get through just fine). OK, I'm all set, I can just let the Roboteq accept everything and ignore their (inflexible) ID filtering. (This is the sort of thing that would drive one nuts with USB or other server-client serial scheme. CAN is great, thank you Bosch!)

The MASTER node program:

User-changeable settings --
A series of user settings are at the top of the MASTER program. These basically correspond to the non-motor programming settings found in any wheelchair controller. It's just a list, without any fancy graphical user interface, but I actually like that better. I'd appreciate it if you'd read through these to see if their meaning is reasonably clear, and to see if I've left out something. Do notice that there are NO acceleration settings - no rubber-band or ocean liner driving is available here. The Roboteq has it's own Roborun software for motor configuration, and those settings can provide mushy behavior if really, really wanted, as well as providing such basic functions as Amp and temperature limits, etc.

Startup --
The startup routine includes setting various internal parameters, pin identifications etc., configuring the CAN controllers, holding the DC-DC converter ON after the momentary push-button starts it up and so on. That's all just ordinary housekeeping. More importantly, it includes a number of safety checks. It makes sure the high-current contactor's coil is neither shorted nor open and that its contacts are not fused. It checks that the brake coils are neither open nor shorted. It makes sure that all Roboteq outputs are in "neutral" and makes sure that the Joystick is not out-of-neutral. The OON check is also run later whenever the controller is toggled between joystick and switch-controlled driving.

The startup section of the MASTER node program also manages use of EEPROM for storing data that we don't want to disappear between sessions. For now that's just Amp Hours used and last mode (driving, seating or lights) in use. Because EEPROM has a finite write-cycle life, these data are stored in ring memories to spread the writes over 40 addresses for each variable. Lifetime of the EEPROM with these rings is 25 years if current gets measured and stored (or mode gets changed) 220 times a day.

Running --
The joystick is read on two analog inputs that check for out-of-range conditions (a failed joystick or broken wire), a speed pot is read on another analog input, and switches are read on a series of digital inputs. Each analog input is read multiple times to average out noise and jitter, and the number of reads is a parameter that can be adjusted for more or less-steady joystick hands. Raw joystick input is scaled in accord with user set parameters for dead band and exponential curving (4 integer arithmetic approximations from none to strong), the data and bitwise complements are put into a CAN frame, sent and checked for arrival of a confirmation message. If the joystick stays at 0, the message is not repeated, but if it is held at a steady non-0 value messages are sent once every 900 msec to satisfy the Roboteq's watchdog timer. If the joystick is not rock steady, messages are sent as fast as the loop can cycle (every 2 milliseconds, worst case even with CAN rate reduced to 125k bps). I don't know if Roboteq's processor is as fast, but it probably will be OK as the majority of the 2 msec is for analog reads, averaging and calculation, rather than for CAN messaging.

The Joystick-switch toggle and the mode switch --
To allow switch driving in different situations (e.g. tight spaces indoors, outdoors such as sidewalks, and wide open spaces), when switch-driving the mode switch cycles through 3 user-defined speed profiles. To ensure that there are no surprises, any time the joystick-switch toggle goes to "switch", the driving profile goes to its slowest setting, and when toggling in either direction, we check for Throttle and Steering neutral before executing the change. We wouldn't want the chair to go off on its own because a switch was kept pressed while changing to joystick, nor would we want the chair to fly off because the someone was still holding on to the joystick while changing to switch driving.

In switch-driving mode, the switches are used to set discrete throttle and turn values. If using lights or seat actuators via the joystick, the joystick values are converted to discrete light or actuator commands.

Setting brakes and power management --
If the joystick goes to 0, current flow is checked and when the motors stop moving the brakes are set (there's a parameter for setting the delay for this - so that the chair stops moving first, but doesn't drift). If the brakes happen to be set when movement is later called for, the brakes are first released, current flow is checked to make sure the brake coils are OK, then the motion command message gets processed. If the brakes fail to open and the chair tries to move, the chair will probably move (sluggishly) but the motors would overheat and could be damaged, hence the check each time that the coils are working.

If current flow stays at minimum (i.e. just brakes) for a while, the high-current contactor is opened. If the contactor is open when we later send a motion, seat or lights action, the contactor is first commanded to close. There is no need to check for proper operation as a contacter coil failure here just means that the chair won't do anything.

If MASTER stays completely inactive for an extended time (10 seconds for testing, but multiple minutes to infinite in actual use), the modules go to sleep. If MASTER is actually turned off (i.e. DC power turned off), the other Nano nodes are put to deep sleep and the Roboteq is powered down before power to MASTER is cut off.

Why have I fussed so much with power consumption? Because I don't want the batteries drained if the chair is sitting around doing nothing, even for a week or two. A high-current contactor relay draws quite a bit of current; the best I've found for this use draws 35 mA, but others may draw as much as half an amp or more. A Roboteq left on and doing nothing draws >100 mA. When awake, the Nano + CAN board may draw as much as 30 mA, but when asleep they drop to microamps (with power LED disconnected). Quiescent current of each DC-DC converter is 5 mA, but except for the Master which can be turned off, these have to be left on unless a separate switched power line were run to each node. Because, in normal operation, the contactor will open only when current is minimal, it should last essentially forever (mechanical life for the one I've chosen is 10^7 cycles), but even in emergency use, where it might be breaking a hefty current, it should have a long life (rating is 5X10^4 operations at 300 Amps and 13.5 VDC; obviously less at >=24 volts, but still more than we'll ever have to contend with.) This is an automotive high current relay and not a traditional contactor-in-a-can. Those handle much more current, but have a lifetime measured in the hundreds of operations and a coil current many-fold higher.

Aux and RoboteqEmulator node programs:

Both of these programs have the same basic structure: they poll continuously for incoming CAN messages, parse the ID to find out whether intended for the Roboteq (which doesn't have a decent mask/filter) and to extract the message type and name that allow selecting among procedures that set motor commands, digital output pins, or read some value. They are much simpler than the MASTER node's program (22 k bytes for MASTER; 9.4k for RoboteqEmulator, and 8.2k for Aux. The last of these, Aux, will grow, but about 1/3 of the memory use is for diagnostic stuff and will eventually get removed.) Polling rather than interrupts are used throughout because Roboteq MicroBasic does not have any interrupt handling.

****
****

If you want to see the details of how these things have been done, I'm afraid you'll have to study the schematics, pcbs and read through the code. I hope that at least some of you will. This is my first "deep" foray into embedded programming and I'm sure there are many improvements that can be made.