CanModule 2.0.22

work done by Michael: hand merge non-official branch from 2.0.14 into master (“Piotr”)
16.aug.2022

Changelog.md (“a polite summary”)

2.0.22 [9.aug.2022]

- starting to merge and implement the changes proposed by piotr (based on 2.0.14) in branch piotr_canopen
- chrono and std:: cleanup, nanosleep etc are suppressed in favour of chrono. Some code streamlining.
- drop messages with extended IDs or data (do not truncate and send nevertheless)
- checked all thrown exceptions: they are indeed runtime_error and NOT logical_error
- getPortStatus(): UNIFIED PORT STATUS. when invoked by a client, it actually talks to the specific hardware for each vendor using the vendor API. It returns a unified port status as documented: first nibble=implementation, all other bits: whatever the vendor returns per implementation. No "abstraction" to an i.e. "CAN_PORT_ERROR_ACTIVE is possible. A port status change occurs (1) there is an error (2) a controlling action by the client has occurred. Errors are reported by LogIt or by thrown exceptions. So there is no need to report port-status changes through signals.
- piotr n slide8: what is status... refers to CAN bus. Depending on the vendor API CAN status is not available through the API! That is why getPosrtStatus() exists. IF we have a CAN error frame we send it per signal.

todo: LogIt linkage check signals for error frames piotr n: error handling slide 10

related to “Piotr’s work” and the refusal to validate 2.0.22 (“less polite, more clear”)

claims made in the document

- “sock” is the only option for Atlas: not true. see new CAN bridges coming, “an2”. Good engineers think ahead. Although this is “true” right now, and entirely Atlas’ responsibility. I just try to help.
- only “sock” is concerned: not true. CanModule is a standard API on top of all vendors, “sock” implementation can not divert. This is well known and can not be “ignored for the convenience of the argument”.

- branch difference with master: rather useless and incorrect counting, this relates to “master 2.0.14”, but we care about tags, not branches. Master branch should never be used for such kind of argument, this is well known as well (Atlas: “tag it and then we will look at it”)
- “many changes are agnostic”, meaning “there is at least one change in sock which breaks the API” ? Breaks in the sense of “change the semantics of an existing API method” (major version, all clients affected) or “add API methods” (medium version change) or “bug fixes” (minor version?

“proposals” made in the document

- general: code cleanup & quality

done, always ongoing. Need to keep BW compatibility and fulfil messy Atlas specifications which are most of the time “one year after the fact”. Had several forward-backward changes due to messy and late feedback from Atlas, but Atlas was always quick to pose new requests and constraints. This is why I have a very conservative “minimalistic” approach now in order to avoid a “waste of time”. Code and quality are “not sexy” C++22 standard, but stable, tested and functional. This is not a fantasy playground for “ambitious theoretical code design”, seeing also the windows/linux constraints.

- introduced proper can controller state machine

this idea is nice, but I reject it. It has only theoretical value for “sock” where such a state machine can be done easily. Even for “sock” the real added value of such a state machine is quite limited and the added code complexity is probably not worth it just to “have a state machine which reports it’s state and puts different names on the same reality”. Even though “following the theoretical CAN states” is a good theoretical idea, but it is “just an idea”, not more. CAN is a low level bus, CanModule is a standard API, any unneccessary extra “intelligence” in such technical layers should be avoided since it just adds confusion and complexity. For the other implementations “an”, peak”, “systec” it is impossible to animate such a state machine presently since the values for CAN errors tx, rx are not reported through the vendor API (this will improve with “an2”). Tx/Rx would therefore come from “software derived statistics” in those cases, and since the statistics is already reported such a state machine is not needed. The “textual representation” of a CAN state is therefore also dropped since it would be a minor convenience method only without extra benefit (except the convenience).

error handling (and status, statistics reporting)

EH and reporting is done on 5 levels in CanModule to reflect status and problems in the best possible way, as close as possible to the hardware and software reality:

1. **LogIt** on all init and runtime diagnostics on all relevant methods and modules: levels TRC (a lot), DBG (little), INF (standard runtime), ERR (terse production).
2. **error frames through signal/handler** subscription, replicating specific hardware errors reported by the vendor APIs, if available. The quality of error handling by the vendors varies (the underlying SW is 10+ years old!). Usually a signal on an error handler signifies rather an error close to the vendor bridge, and not related to CanModule itself.
3. **thrown exceptions**. Exceptions in a multithreaded linux/windows environment lead to program abortion in CanModule, there is no specific upper layer for “exception catching and treatment”. Exception throwing should NOT be used for “error reporting “ anyway, exceptions are fatal runtime conditions with a debugging trace leading to execution flow abortion, without heap deallocation in most cases (windows?). This is the “classical defensive interpretation” of C++ exceptions and if one wants code which works one should better stick to that and not use exceptions for program execution flow management. The CanModule client (~OPCUA server)

technically might choose to continue execution and reastart CanModule, but heap deallocation remains unclear in this case, this is relevant for a few mutexes which are needed to force serialize certain “delicate parts” of vendor API calls (like e.g. configuring ports etc) . 10+ years old vendor code is not expected to use “C++ smart auto-deallocation pointers for objects” in **all cases**. CanModule methods are re-entrant as much as possible (there is unknown vendor code underneath with mostly unknown characteristics so no guarantee can be given) and stability for reconnection has been tested, checked against memory holes and proven, so “CanModule is re-entrant” is true to a large part. Nevertheless the client should exit if CanModule throws an exception in order to provide maximum execution reproduction and stability. To be noted also: CanModule does offer well defined “reconnection behavior” for all implementations and therefore is able to safely recuperate from severe hardware faults and e.g. local power cuts without interrupting the execution flow.

4. **unified port status**. The vendor CAN port status from the hardware (vendor API) is made available in a 32bit pattern which also codes the implementation. This is a general philosophy which is compatible to all present and future implementations/vendors. This is extensively documented.
5. **statistics**. CanModule monitors message flows and derives statistics from this information, which are available through fast API calls. Documentation has been improved, methods are tested and should work. A minor functional change, related to the observation time interval, might be appropriate: presently an interval is defined as the time lapse between two successive calls (which can be months apart), an option for setting a fixed interval might be useful [not implemented, low priority].

The “proposed error handling relating to the state machine” is superfluous since the state machine is not implemented anyway, and since the existing status and error handling is very complete. I checked CanModule 2.0.22 nevertheless and added a few more error signals where appropriate, but this was minor.

uniform std::exceptions “logic_error”. After close inspection of the code, logic_error is not needed. Only std runtime_error are thrown. There is no real case to distinguish that, unless one wants to implement a big and useless water-head of a largely theoretical state machine (see above).

aliasing rules of “select”. This was a very valuable contribution which we detailled in collaborative discussion, and which got implemented in 2.0.14 (or later… I don’t remember) and fixed for both CC7 and CS8. Reporting it here again is obsolete, unlsess a bug/erroneous behavior was detected in CS8 (only “sock” concerned). In this case an Issue should be reported in the standard way to ICS.

statistics module misuse: some cleanup was done in 2.0.14 of statistics, it is unclear what this proposal refers to. Statistics have been slightly unhappy in previous, I hope it is OK now.

gettimeofday/nanosleep/chrono: cleaned up to C11 standard a long time ago, went though it again, found 1 or 2 additional places. Only relevant for “code beauty”, but was easily accepted as a good proposal.

reduced code redundancy: code redundancy is very practical if sections of code which are presently similar are expected to divert from each other in the future. Redundancy can provide code flexibility therefore and keep complexity low at the same time. Not everything you lean in code-schools is actually a good idea in the long run. Therefore “just because code could be streamlined and modularized” does not mean this is (a) an improvement (b) and intelligent approach to long term

maintenance. Code diversion in CanModule can be generally expected, due to linux/windows evolution, vendor API evolution and bug fixes, the addition of new bridges and changes in functionality. Therefore “highly modularized code” is actually quite a bad idea for CanModule. Please think before proposing bs.

8 byte message restriction handling. that was a clear bug (without consequences until now), corected, thanks !

11bit ID restriction: same, a clear bug, corrected!, thanks !

rewrite algorithm which maps CAN ports to network interfaces. The present algorithm works and was tested, also foor “sock” using peak modules (where the port numbering is undetermined and depends on the plugin sequence, this is fixed). Rewriting an algorithm which works for the sake of proposing a better algorithm which is not scoped and tested to cover all use cases and contexts: sorry, this is not worth spending time on it. If you make a clear JIRA issue and maybe propose a comfortable merge then we might agree. Like this it is rather a waste of time just “beautyfying code” without added value (except the programmer’s ego gets bloated).

build systems. cmake build chain is tricky for CanModule, and any contribution for improvement is welcome. Went though this again and improved some places. The “as-static-as-possible” build option exists since a loooong sime (2.0.3 I think), and for “sock” this is easily done, including boost as well, but keeping the dynamic loading at init. The dynamic (-static) loading of vendor libs at init is a historical key feature of CanModule. To my mind this key feature should be replaced with full static linking, only, the vendor libs are delivered as static but non-relocateable libs (-fPIC missing). So the possibilities for full static linking are limted (tis is why it is called “as-possible”). The new “an2” implementation with modernized bridge should make full static link possible at least in the “an2” case, for linux and windows. Ripping out the CanModule build chain so that it works statically for “sock” for an OPC UA server, and reporting that as a “proposal for CanModule improvement”, is of course possible but misses entirely the point of CanModule: vendor and OS compatibility. While certain hints were gladly taken, the underlying idea is too limited and therefore can not be accepted presently.

remarks

- **exceptions** are bs in that context, see above
- **cleaned up headers:** yes, good idea, thanks for the initiative. cleaned up much.
- **port status relocation.** ever actually read and understood “unified port status”? not understood the point of this remark.
- **minor data type corrections.** yes, mee too. ca me fatigue. code cleanup is always good but should not become an obsession for ambitious engineers. We are CERN (aging prototypes, poor, long term maintenance resources thin) and not google (3 lines of high quality code per week, lots of money & resources, maintenance done by global rip-out-and-rewrite) make a proper JIRA case.
- **current “reconnection API” is incompatible with socket CAN behavior?** Well-it works and it is tested ! CANX ! did you **really** miss out completely on all of this?! It might be of course incompatible with the proposed state machine… and who cares about “a model” - this is of no practical value, just “ideas in one’s mind” which are sometimes rather a hindrance than a

help ! CanModule is NOT in the conception or design phase! CanModule is in the “tedious and very conservative long term maintenance” stage!

- **further improvements/changes likely to be necessary**. Sure- if you want to push all of your proposals this is a lot of work and a lot of consequences. Otherwise this is a trivial and useless statement with a menacing and uncollegial undertone. poubelle alors.
- **user documentation should be extended.** I made another round and added text on : standard API, reentracy, blocking. There are no callbacks (=stand alone GUI methods), this is a low-level software layer, non-GUI. There are (boost) signals and handlers and they do the error reporting. The condition is: if there is something to report, send a signal. I don’t get the point of this comment. Error and status reporting are extensively documented. There is no “error handling” since errors etc are “reported” but nothing is done about them (that is why they are errors). So no handling of errors, and also no model on “error handling”. Take a beer and relax, man.