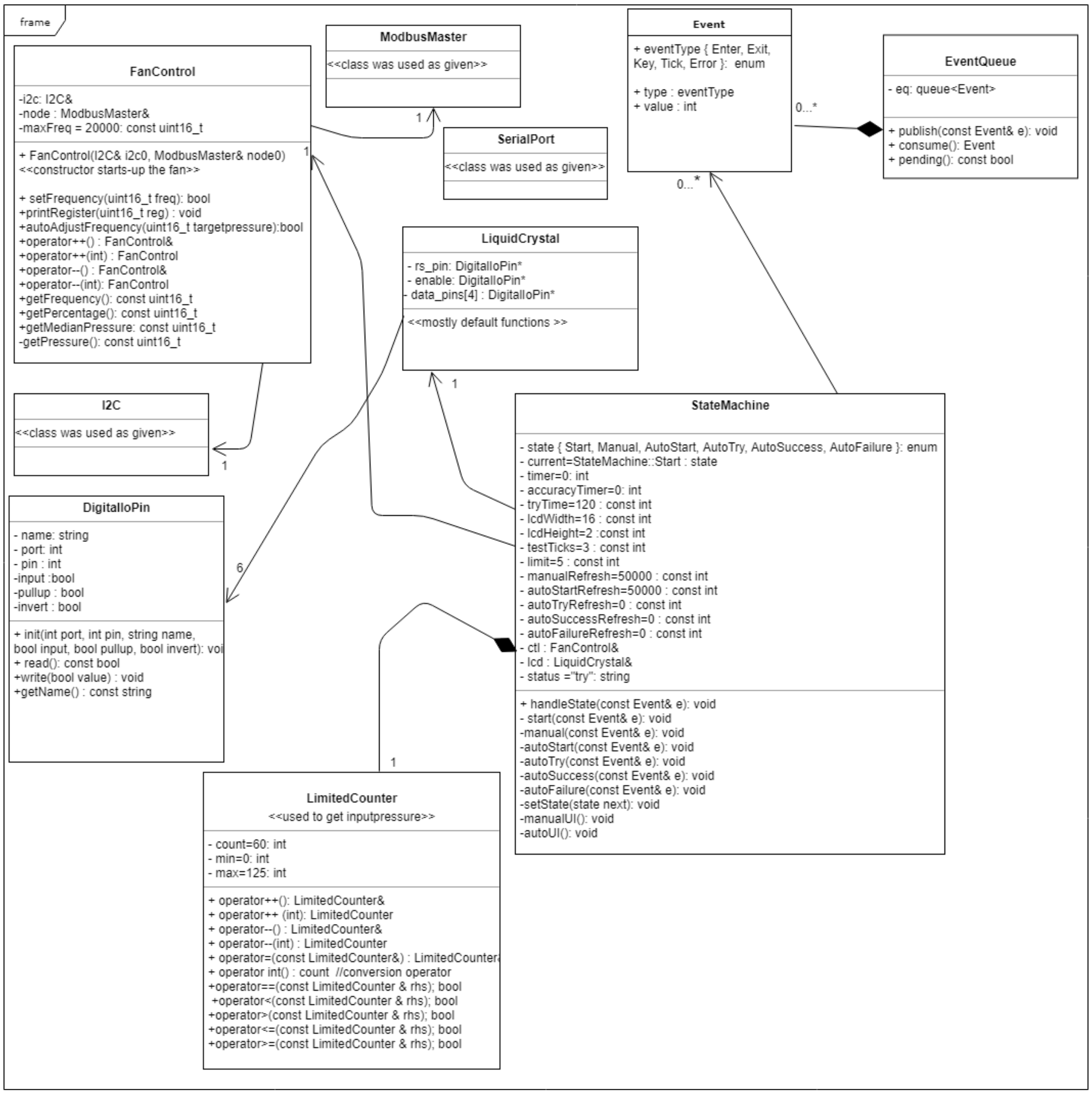
ABB Modbus Program documentation

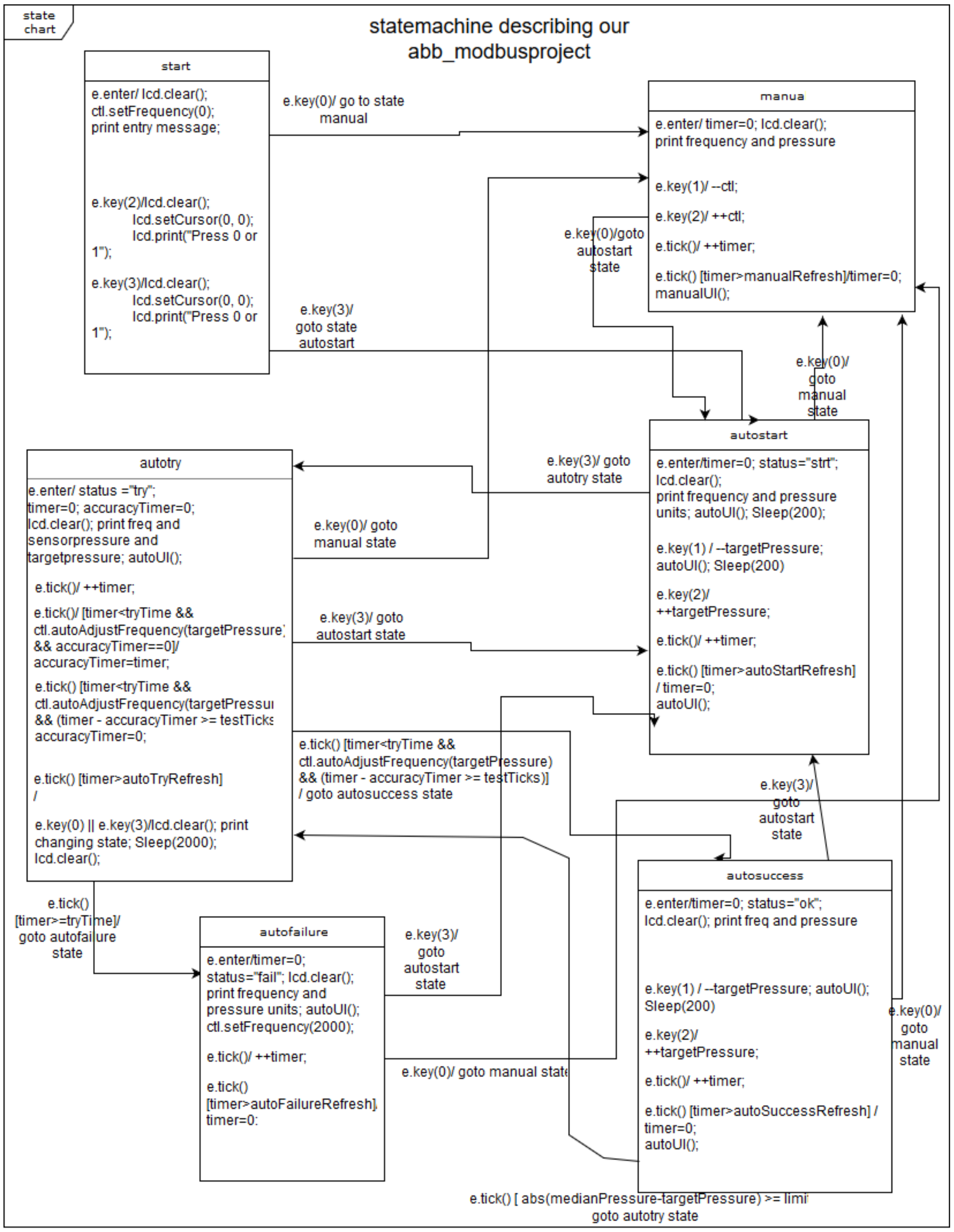
15.3.2018

Arvi Koivulehto and Lauri Solin

UML STYLE CLASS DIAGRAM


frame
ModbusMaster
<<class was used as given>>
1
Event
+ eventType { Enter, Exit, Key, Tick, Error }: enum
+ type : eventType
+ value : int
0...*
EventQueue
- eq: queue<Event>
+ publish(const Event& e): void
+ consume(): Event
+ pending(): const bool
0...*
FanControl
-i2c: I2C&
-node : ModbusMaster&
-maxFreq = 20000: const uint16_t
+ FanControl(I2C& i2c0, ModbusMaster& node0)
<<constructor starts-up the fan>>
+ setFrequency(uint16_t freq): bool
+printRegister(uint16_t reg) : void
+autoAdjustFrequency(uint16_t targetpressure):bool
+operator++() : FanControl&
+operator++(int) : FanControl
+operator--() : FanControl&
+operator--(int): FanControl
+getFrequency(): const uint16_t
+getPercentage(): const uint16_t
+getMedianPressure: const uint16_t
-getPressure(): const uint16_t
1
SerialPort
<<class was used as given>>
LiquidCrystal
- rs_pin: DigitalIoPin*
- enable: DigitalIoPin*
- data_pins[4] : DigitalIoPin*
<<mostly default functions >>
1
I2C
<<class was used as given>>
1
StateMachine
- state { Start, Manual, AutoStart, AutoTry, AutoSuccess, AutoFailure }: enum
- current=StateMachine::Start : state
- timer=0: int
- accuracyTimer=0: int
- tryTime=120 : const int
- lcdWidth=16 : const int
- lcdHeight=2 :const int
- testTicks=3 : const int
- limit=5 : const int
- manualRefresh=50000 : const int
- autoStartRefresh=50000 : const int
- autoTryRefresh=0 : const int
- autoSuccessRefresh=0 : const int
- autoFailureRefresh=0 : const int
- ctl : FanControl&
- lcd : LiquidCrystal&
- status ="try": string
+ handleState(const Event& e): void
- start(const Event& e): void
-manual(const Event& e): void
-autoStart(const Event& e): void
-autoTry(const Event& e): void
-autoSuccess(const Event& e): void
-autoFailure(const Event& e): void
-setState(state next): void
-manualUI(): void
-autoUI(): void
DigitalIoPin
- name: string
- port: int
- pin : int
-input :bool
-pullup : bool
-invert : bool
+ init(int port, int pin, string name, bool input, bool pullup, bool invert): voi
+ read(): const bool
+write(bool value) : void
+getName() : const string
6
1
LimitedCounter
<<used to get inputpressure>>
- count=60: int
- min=0: int
- max=125: int
+ operator++(): LimitedCounter&
+ operator++ (int): LimitedCounter
+ operator--() : LimitedCounter&
+ operator--(int) : LimitedCounter
+ operator=(const LimitedCounter&) : LimitedCounter
+ operator int() : count //conversion operator
+operator==(const LimitedCounter & rhs); bool
+operator<(const LimitedCounter & rhs); bool
+operator>(const LimitedCounter & rhs); bool
+operator<=(const LimitedCounter & rhs); bool
+operator>=(const LimitedCounter & rhs); bool

UML STATECHART DIAGRAM


state chart
statemachine describing our abb_modbusproject
start
e.enter/ lcd.clear(); ctl.setFrequency(0); print entry message;
e.key(2)/lcd.clear(); lcd.setCursor(0, 0); lcd.print("Press 0 or 1");
e.key(3)/lcd.clear(); lcd.setCursor(0, 0); lcd.print("Press 0 or 1");
e.key(0)/ go to state manual
e.key(3)/ goto state autostart
manua
e.enter/ timer=0; lcd.clear(); print frequency and pressure
e.key(1)/ --ctl;
e.key(2)/ ++ctl;
e.tick()/ ++timer;
e.tick() [timer>manualRefresh]/timer=0; manualUI();
e.key(0)/goto autostart state
e.key(0)/ goto manual state
autostart
e.enter/timer=0; status="strt"; lcd.clear(); print frequency and pressure units; autoUI(); Sleep(200);
e.key(1) / --targetPressure; autoUI(); Sleep(200)
e.key(2)/ ++targetPressure;
e.tick()/ ++timer;
e.tick() [timer>autoStartRefresh] / timer=0; autoUI();
e.key(3)/ goto autotry state
autotry
e.enter/ status ="try"; timer=0; accuracyTimer=0; lcd.clear(); print freq and sensorpressure and targetpressure; autoUI();
e.tick()/ ++timer;
e.tick()/ [timer<tryTime && ctl.autoAdjustFrequency(targetPressure) && accuracyTimer==0]/ accuracyTimer=timer;
e.tick() [timer<tryTime && ctl.autoAdjustFrequency(targetPressure) && (timer - accuracyTimer >= testTicks accuracyTimer=0;
e.tick() [timer>autoTryRefresh] /
e.key(0) || e.key(3)/lcd.clear(); print changing state; Sleep(2000); lcd.clear();
e.key(0)/ goto manual state
e.key(3)/ goto autostart state
e.tick() [timer<tryTime && ctl.autoAdjustFrequency(targetPressure) && (timer - accuracyTimer >= testTicks)] / goto autosuccess state
e.tick() [timer>=tryTime]/ goto autofailure state
autofailure
e.enter/timer=0; status="fail"; lcd.clear(); print frequency and pressure units; autoUI(); ctl.setFrequency(2000);
e.tick()/ ++timer;
e.tick() [timer>autoFailureRefresh] timer=0:
e.key(3)/ goto autostart state
e.key(0)/ goto manual state
e.key(3)/ goto autostart state
autosuccess
e.enter/timer=0; status="ok"; lcd.clear(); print freq and pressure
e.key(1) / --targetPressure; autoUI(); Sleep(200)
e.key(2)/ ++targetPressure;
e.tick()/ ++timer;
e.tick() [timer>autoSuccessRefresh] / timer=0; autoUI();
e.key(0)/ goto manual state
e.tick() [abs(medianPressure-targetPressure) >= limi goto autotry state

NOTES ABOUT THE CODE

Early on we decided to implement state machine.

First key objects from ModbusMaster, I2C and FanControl are initialized in order to make Modbus and I2C connections to work, and power-up the fan.

DigitalIoPins, LCD, EventQueue and StateMachine are also initialized, and the starting state will be StartState.

Mainloop deals with event generation and event consumption: tickEvents, and buttonpressEvents mainly. StateMachine handles state based on the incoming events from queue. Currently tickEvents are generated as fast as the mainloop runs, assuming no other delays from one of many causes.

StateMachine was implemented with the one-method per state, and there are currently 6 states implemented. (start, manual, autostart, autotry, autosuccess, autofailure).

LimitedCounter class is only really used for getting the inputPressure values from the user in autoStart state, and for holding that value of course as a targetPressure value for autoAdjustingFrequency().

IMPLEMENTATION OF AUTO-ADJUST-MODE

Most challenging in the project for us was the implementation of the automatic mode for autoAdjusting the frequency based on sensorPressure, in order to match the inputPressure. Also the balancing act between adjusting multiple Sleep() delays was quite challenging in the project overall.

Our implementation of the autoAdjustFrequency was a two-geared-adjustment system with if-statements, starting with more coarse frequency increment/decrements (4%) and ending up with 1% frequency accuracy in increment/decrement, when the sensorPressure and inputPressure are closer together than 30 pascals. (the pressurevalue is defined in threshold const int in FanControl). The frequency increases are also definable in FanControl.h.

The good part about our implementation of the autoTry mode was that the automatic frequency adjustments happened at a rapid rate in the tickEvents. The function was also programmatically simple to make, compared to the alternative of a PID controller and tuning PID coefficients.

AutoAdjustingFrequency was tested and we found that typically the function is able to reach reasonable goals within 1-2 minutes of starting. The timer expiring guard condition in the autoTry state should cause failure in under 2 minutes, if the desired pressure goals were not reasonable and could not be achieved.

There is also a stabilization feature added into the autoTry state with StateMachine datamember accuracyTimer. In autoTry state, in the tickEvent, we check that there have been 3 tickEvents such that the inputPressure and sensorPressure were equal. And only then does the autoTry state succeed in going to autoSuccess state.

AutoFailure state was implemented at a fairly late stage in the development and its main purpose was to be a stopgap state, where the frequency is manually set to reasonable value (freq=2000), and the user is notified with LCDprintout. The user can acknowledge the autoFailure state, and navigate to useful states with buttons.

AutoSuccess state was designed keep the current frequency at the steady state, and monitor further changes. Typically, the device would succeed from adjusting the frequency in the autoTry state, and that would activate state change to AutoSuccess state. AutoSuccess state monitors the current sensorPressure level, and compares it to the previously given inputPressure. If the sensorPressure changes up or down by too much (about +- 5 pascals, defined in limit in statemachine), then the state changes back to the AutoTry state.

SensorPressure itself is gotten with getMedianPressure() function and we used sample size of 10 values with median filtering of pressure values.

All the automatic mode states have currently button-enabled state changes to other states, primarily autoStart state and Manual states.

We had to add more delay to the buttonPress events in the AutoTry mode, because otherwise it was difficult to make a smooth and controlled exit from the AutoTry state, and react fast enough by releasing the button, so you don’t immediately enter into another state when you exit. We suspected that the reason was maybe the delays incurred from AutoTry tickEvents. We didn’t have any busyloops with buttonpolling, but we used delays.

Many crucial functions such as setFrequency() and getFrequency() were wrapped in our FanControl class, and we have some FanControl operator overloads such as decrement and increment frequency by 1%.