**A. Overview**

The objective of the contactor “project” (project in the STM32CubeMX terminology) is to control the contactors that connect the battery string to the motor inverter (DMOC). A key part of this is pre-charging the DMOC capacitor so that when the contactor is closed the current surge is with the limits of the contactor contact rating.

Two modes are accommodated. A two contactor mode is based on two contactors. Contactor #1 connects the battery string to the DMOC plus. Contactor #2 connects the DMOC minus to the battery string minus. The pre-charge resistor is across contactor #2 contacts and when it is closed it shorts the pre-charge resistor.

The one contactor mode has one contactor (designated #1) and a small pre-charge relay (designated #2). The pre-charge relay connects the battery string to the DMOC plus via a pre-charge resistor. When the DMOC capacitor is charged sufficiently, the contactor is closed. The DMOC minus connects directly to the battery string minus.

Also involved is to detect special conditions and failures. E.g. if the battery string voltage is too low a command to connect will put the ContactorTask state in a fault condition and not respond to connect commands.

This project organization is based on using STM32CubeMX and FreeRTOS, and be event driven.

**B. Hardware**

The program is based on the f103Ard pcb in the following directory which is a motherboard for the Blue Pill module-- ~/GliderWinchCommons/embed/svn\_sensor/hw/trunk/eagle/f103Ard

Three additional boards are connected to the f103Ard board: two fethe sub-boards which hold the isolated FET driver and FET for driving the contactor coil, and one custom module constructed on a perf-board with a Blue Pill module for isolated measurement of voltages of battery string and DMOC input.

1. f103ARd—Blue Pill Motherboard

The eagle files for this board can be found in--

~/GliderWinchCommons/embed/svn\_sensor/hw/trunk/eagle/f103Ard

A key reference to the board usage is in the following--

~/GliderWinchItems/contactor/README.pins

Some modifications have been made to the board.

The connector arrangement, top view, with CAN RJ 45 jacks at the lower left hand--

a) Serial port: above the dc-dc switcher module, is a six pin header for FTDI type uart<->usb, used for debugging and console monitoring.

1) Gnd – rightmost pin

2) nc

3) nc

4) tx (on ftdi module)

5) rx (on ftdi module)

6) nc

b) Programming port: 4 pin header at bottom of Blue Pill module

1) gnd – rightmost pin|connects to Discovery hdr pin 3

2) – connects to Discovery hdr pin 2

3) – connects to Discovery pin 4

4) +3.3v (not used if motherboard powered from CAN) | connects to Discovery 3.3v on 50 pin hdr, if 3.3v power needed.

Note:

- The pin order of pins 2 & 3 reverses between the Blue Pill and Discovery.

- Discovery pins 1,5, 6 are not connected.

- The two SWD jumpers are removed on the STLINK hdr

c) DMOC enable—three pin, keyed, approximately center

FET pull-down for DMOC hardware enable.

1) gnd

2) FET drain

3) available for diode in case pwm an inductive load used. The diode is not installed.

d) RJ11 jacks for external units

Top left: isolated uart input from Blue Pill isolated high voltage sensor module

Top right: Hall-effect current sensor for battery string current

Bottom left: drive for isolated FET driver for contactor #1

Bottom right: drive for isolated FET driver for contactor #2 (or relay)

2. fethe—isolated FET driver sub-boards

The eagle files for this board can be found in--

~/GliderWinchCommons/embed/svn\_sensor/hw/trunk/eagle/fethe

These boards hold the isolated FET driver, and power FET for driving the contactor coils.

R11 jack connects to the BP Motherboard

green led is on when Motherboard drive is high

green led is on FET drive plus is present

Connections

Three large holes, top, left--

gnd – contactor coil power ground (e.g. 12v minus)

+ - connect to 12v plus IF diode for pwm’ing coil is used

nearest FET – FET drain connect to contactor coil.

Note: instead of RJ11 cabling, direct wiring can be done using the viasadjacent to the RJ 11 jacks. Pass one wire through the via for strain relief and solder to the adjacent via, etc.

The right half of the fethe sub-boards are for various types of Hall-effect sensors and not use for this project.

3. Blue Pill isolated high voltage module

This board uses a Blue Pill module to measure four voltages with a common ground, with the ground isolated. A dc-dc module supplies 5v from the Motherboard. A photocoupler isolates the uart tx to the Motherboard.

The eagle files for this module can be found in--

~/GliderWinchItems/bpsensor/hw/eagle/bpsensor

The program layout is a clone of the contactor program with unused code removed.

The high voltage inputs go into a 2.2meg | 15K resistor divider.

**Note: when programming the Blue Pill on this module the Discovery board and usb to the PC are NOT ISOLATED!**

4. simulation module

The eagle files for this module can be found in--

~/GliderWinchItems/contactor/hw/eagle/simulator

This module is used for testing/debugging and simulates the two contactor modes. It does not simulate the auxiliary contact features and these have not bee tested in the program.

Connections to the module--

Note—the connectors are **NOT keyed. Be careful!**

top left: contactor #1

bottom left: contactor #2

gnd) connects to gnd (near FET on fethe subboard)

signal) connects to FET drain on fethe subboard

12v) connects to

a) fet driver supply pin

b) plus (diode) IF pwm of coil used

top middle: 12v supply

This supplies both the fet drive power and simulates a 12v battery string.

Bottom center—high voltage sensor connections

gnd – battery minus

1 – (hv1) battery string voltage

2 – (hv2) DMOC plus

3 – (hv3) DMOC minus

HV3 measures the voltage across the pre-charge resistor for the two contactor mode. (HV1-HV2) measures the voltage across the pre-charge resistor for one contactor (w pre-charge relay) mode.

Right most—discharge hdr

When the connect sequence ends the capacitor is charged and needs to be discharged for the next test cycle. A pushbutton plugged into the header allows for manually discharging the capacitor.

LEDs--

Dull green – 12v is present

Clear red (top) contactor #1 coil energized

Clear red (bottom) contactor #2 energized

Pre-charge capacitor simulation--

Resistor = 1K + 2.2K; Capacitor = 1000u

The 2.2K resistor was added to slow the charging to make it easier to observe what the program was doing.

Small resistors are in series with the 12v supply input and discharge hdr to limit smoke and dismay when bumbling about.

**C. Program layout**

STM32CubeMX generates ‘main’ plus the HAL and FreeRTOS routines. Added to the generated directories are the directories Ourwares and Ourtasks. Outwares holds our routines generally used across the winch applications. Ourtasks holds routines more specific to the project. The distinction is not rigid.

Makefile is generated by STM32CubeMX and is updated when ‘MX regenerates the project, if additional ‘MX files are added. To compile files from Ourwares and Ourtasks the lines are added to Makefile, but fortunately ‘MX does not overwrite the added lines. The list of files to be compiled in Makefile is obvious.

If floating point output in printf is used the Makefile LDFLAGS is modified to -u \_printf\_float instead of the -nano.

When main.c is generated by ‘MX the only task created is defaultTask. The initialization code is augmented with calls to files in Ourtasks and Ourwares for creation of tasks, buffers, etc. When there is a failure morse\_trap is called and the green LED on the Blue Pill flashes a numeric code. A search of the files will turn up the location of the code.

Upon successfully completing the initialization phase of main, main calls the FreeRTOS scheduler and at that point it the various tasks are running. The defaultTask toggles the led every second as an indication that at least the defaultTask is running OK. If the it doesn’t toggle a likely problem is that there is a crash. Running openocd and telenet localhost 4444 and looking at the pc is the starting point. There are some asserts in FreeRTOS which can hang the program, e.g. trying to set a software timer with a timeout count of zero.

Besides initialization in main.c there is initialization in the tasks, e.g. obtaining a buffer for serial output. These can fail and have morse\_traps for identification.

The main driver of the logic of the project is ContactorTask. The other tasks are more support for this task. The tasks running (the following in the order they are created in main initialization)--

1. defaultTask

One per second output of monitoring msgs, e.g. task stack usage, ADC readings, etc. Uses yprintf.

2. SerialTaskSend

Sending uart msgs from any task.

3. SerialTaskReceive

Receiving uart lines.

4. CanTxTask

Sending CAN msgs.

5. ContactorTask

Logic and operation of contactors.

6. MailboxTask

Sets up mailboxes for received CAN msgs and notifies task associated with the mailbox. can\_iface interfaces our handling of CAN tx and rx msgs with the STM32CubeMX HAL CAN routines. The incoming CAN msg interrupt results in notification of a task, whose task-handle was initialized in the control block for that particular CAN module. (In the Blue Pill there is only one CAN module, but most F4 series can have two, and there are three in the F413 and F767.)

7. ADCTask

Sums ADC readings for DMA ½ and applies initial calibration.

8. Software timer 1

Periodic timer for keepalive/command msgs. Incoming command msgs reset the keepalive and result in a a status CAN msg response. When it times out the periodic callbacks send a status msg as a heartbeat in the absence of keepalive/command msgs.

9. Software timer 2

One-shot timer for waiting for expected contact closure, opening, etc.

10. Software timer 3

Timer to detect failure to receive the isolated high voltage readings.

Note: this task might be eliminated by using timer 1 to check if high voltage readings are not arriving.

**D. ContactorTask specifics**

This task is the central command for the contactor operation.

Following the initialization the endless loop--

1. Wait for notification

This is similar to ‘select’ in unix. It waits for one or more notifications to arrive. When a notification arrives a bit is set in a ‘noteval’ that identifies the source of the notification. The following notifications are in the program--

/\* Task notification bit assignments. \*/

#define CNCTBIT00 (1 << 0) // ADCTask has new readings

#define CNCTBIT01 (1 << 1) // HV sensors usart RX line ready

#define CNCTBIT02 (1 << 2) // spare

#define CNCTBIT03 (1 << 3) // TIMER 3: uart RX keep-alive

#define CNCTBIT04 (1 << 4) // TIMER 1: Command Keep Alive

#define CNCTBIT05 (1 << 5) // TIMER 2: Multiple use delays

/\* MailboxTask notification bits for CAN msg mailboxes \*/

#define CNCTBIT06 (1 << 6) // CANID\_CMD: incoming command: cid\_cmd\_i

#define CNCTBIT07 (1 << 7) // CANID-keepalive connect command: cid\_keepalive\_i

#define CNCTBIT08 (1 << 8) // CANID-GPS time sync msg (poll msg): cid\_gps\_sync

2. Event handling

The notification is some sort of event. Each notification bit is checked and the associated notification is handled. E.g. a uart line received from the high voltage sensor requires extracting the three readings from the line and applying a calibration. The above notification list is also the event handling list.

Most notifications do not generate any actions. One exception is the keepalive/command timer timeout. When that occurs a CAN msg response with the status is always returned.

3. States

Following the handling of events, the current state is checked, i.e. any event results in a pass through the current state logic. A switch statement on the current state code calls a routine that handles that state. (Since these states are rather simple, the routines for all the states are combined into one .c file. For complicated situation each state might warrant a separate file.)

4. Output updates

Following the handling of the states any udpates to outputs are executed. The execution of states sets bits in an output status word and the outputs are updated (e.g. de-energizing contactor coils).

The conclusion of the updates ends the wait loop. The word with the notification bits handled during the loop feed back into the next wait-for-notification, to reset the bits in the FreeRTOS internal notification word. This sequence takes care of the case of notifications arriving during the execution of the loop.

**E. States – brief**

The list of states with very terse and incomplete description--

1. DISCONNECTED

This the initial state and final state with the contactors open. If the battery is voltage is too low the state will move to FAULTING, and onward to FAULTED.

2. CONNECTING

This and its substates take care of the sequence of energizing the contactors, checking the charging voltage, and either ending up going to FAULTING and onward to FAULTED.

3. CONNECTED

Contactors closed. A command to disconnect or reset will exit this state to DISCONNECTING and onward to DISCONNECTED.

4. FAULTING,

Deals with de-energizing the contactors and the delay for opening.

5. FAULTED

Stuck in this state until a RESET command is received. A code specifying the reason for the fault shows up in the status msgs.

6. RESETTING

Almost the same as DISCONNECTING except for the resetting of the fault code.

7. DISCONNECTING

Similar to FAULTING except the end-point is DISCONNECTED rather than FAULTED.

**F. States – a little more description**

We start with contactors open, etc. and the state being "disconnected."

Abbreviation:

KA - KeepAlive/command msg

Aux = Auxiliary contact

**DISCONNECTED state**

This state is changed when:

1) Battery string voltage is less than require to start a launch--

Set code number for battery string low.

New state: FAULTING

2) If Aux contacts are present on contactor #1, AND Aux 1 does not show OFF--

Set timer2 for longest of the two contactors opening delay

New state: FAULTING with code number for this fault.

3) If Aux contacts are present on contactor #2, AND Aux 2 does not show OFF. (Obviously, when there is one contactor and a pre-charge relay, there would be no Aux 2 contacts)--

Set timer 2 for contactor #2 opening delay

New state: FAULTING with code number for this fault.

4) CAN command (keep-alive) msg "connect" is received.

Set software timer2 for contactor #1 closure delay--

(approx 25 ms and if it has aux contacts this parameter might have to include

some delay if the aux contact close a bit behind the main contacts)

Set connecting substate to C1

New state: CONNECTING

**CONNECTING state**

**substate C1** closing contactor #1

This state changes when:

1) Timer2 times out

Contactor #1 should now be closed, contactor #2 still open.

If aux1 present, and is not ON, then

Set code for aux1 not closing

New state: FAULTING

else

If two contactor config, and hv2 not approx equal hv1, then

Set code for contactor #1 didn’t close

New state: FAULTING

Set timer2 to minimum pre-charge duration (approx 3 secs)

New state: substate C2

**substate C2** charging DMOC capacitor

This state changes when:

1) Timer2 times out.

Set timer for limit to extended pre-charge duration (4 secs?)

New state: substate C3

**substate C3** charge DMOC cap until cutoff voltage

This changes state when:

1) If two contactors

hv3 is less than cutoff voltage (e.g. 10v)

else

(hv1-hv2) less than cutoff voltage

Set timer2 for contactor #2 closing delay

New state: substate C4

**substate C4** contactor #2 closing

This changes state when:

1) If aux2 present and aux2 not closed

Set fault code: contactor #2 aux not closing

New state: FAULTING

2) Timer 2 times out.

If one contactor config,

de-energize contactor #2 (pre-charge relay)

New state: CONNECTED

**CONNECTED state**

This changes state when:

1) KA disconnect or reset commands received

Set timer2 to longest of contactors open delays.

New state: DISCONNECTING

**DISCONNECTING state**

Changes state when:

1) timer2 times out.

New state: DISCONNECTED

**FAULTING state**

Changes state when:

1) timer2 times out

New state: FAULTED

**FAULTED state**

The fault code list--

enum CONTACTOR\_FAULTCODE

{

NOFAULT,

BATTERYLOW,

CONTACTOR1\_OFF\_AUX1\_ON,

CONTACTOR2\_OFF\_AUX2\_ON,

CONTACTOR1\_ON\_AUX1\_OFF,

CONTACTOR2\_ON\_AUX2\_OFF,

CONTACTOR1\_DOES\_NOT\_APPEAR\_CLOSED,

PRECHGVOLT\_NOTREACHED,

CONTACTOR1\_CLOSED\_VOLTSTOOBIG,

CONTACTOR2\_CLOSED\_VOLTSTOOBIG,

KEEP\_ALIVE\_TIMER\_TIMEOUT,

};

Changes state when:

1) KA command reset received

Clear fault code

New state: DISCONNECTED

**G. Parameter list**

Parameters are organized along the lines used for the bare metal and Derby database scheme. In the data base scheme parameters for a function are designated in PARAM\_LIST\_INSERT.sql and values for each instance of the function specified in PARAM\_VAL\_INSERT.sql. These files are found in

GliderWinchCommons/embed/svn\_common/trunk/db

A java program outputs a file with these values as a struct in a .c file, which is compiled and loads into high flash for semi-permanent storage of the parameters for the particular instance of the function. When the program boots it copies the value in high flash into a sram working struct.

For contactor, the database/high-flash part is skipped and a routine with the parameter values hard-coded copies the parameters into the sram working struct. If the database/high-flash part is implemented (making possible online update of parameters) the only change for contactor would be to change the routine that hard-code initializes the working struct to one that copies the values from high-flash to the working struct.

For contactor there are two parameter lists. One is for the contactor function and the other is for the ADC. In the general sense the ADCs (as many as three modules) could serve multiple functions, however making a general multi-function interface is messy, e.g. one could use three of the ADCs, but only ADC1 accesses the internal voltage reference and temperature sensor, etc. Therefore, the parameters for the ADC were grouped separately from the contactor parameters.

The naming convention for the parameters is struct <NAME>LC (LC = Local Copy) giving rise to a .c and .h file. In this case--

adc\_idx\_v\_struct.c [hard code set values]

adc\_idx\_v\_struct.h [parameter struct definition]

contactor\_idx\_v\_struct.c [hard code set values]

contactor\_idx\_v\_struct.h [parameter struct definition]

A function instance requires more than just the parameters, so a struct is defined that contains the parameter struct plus the other other working variables. The naming convention for these structs is <NAME>FUNCTION.

adcparams.c

adcparams.h

ContactorTask.c

ContactorTask.h

In ContactorTask.h above the contactor function struc begins as--

struct CONTACTORFUNCTION

{

// Parameter loaded either by high-flash copy, or hard-coded subroutine

struct CONTACTORLC lc; // Parameters for contactors

struct ADCFUNCTION\* padc; // Pointer to ADC working struct

The first entry, struct CONTACTORLC lc; // Parameters for contactors is the parameter struct for contactor. The second entry is pointer (mostly for convenience) to the adc function working struct. With this layout a single pointer to the contactor function struct accesses “everything” that is either a variable or parameter. The only parameters that might be in the executed code would be some universal scaling constants.

When the program starts the initialization calls the .c files above which in the case of the parameter list loads the parameters values into the working struct. A subsequent call to the .c files for the function struct initializes the working parameters, such as converting specified parameters such as milliseconds into timer ticks, and converting floating point (e.g. voltages) into scaled integers, range checks on parameters, pre-computing ratios for ADC calibrations, etc.

**H. ADC**

The ADC scans six channels, storing the 12b readings in a DMA circular buffer. The DMA interrupts at the ½ and end points of storing in the buffer, and the buffer length holds sixteen scans (six reading per scan) in each half. Upon a DMA interrupt the sixteen readings are summed for each channel. Since the maximum reading possible is 4095, the maximum sum is 65520 which fits in a unit16\_t. The DMA interrupt notifies the ADCTask which carries out the summation and subsequent calibration of the readings.

The ADC calibration is based on using the internal voltage reference (Vref) rather than Vdd as the Vref is more stable. In the DiscoveryF4 Vdd can vary between about 2.85v and 3.3v depending whether the power source is the USB, as there is a diode drop involved. For the Blue Pill there is a 3.3v regulator and no diode involved, but the internal reference is more stable than the regulator.

The ADC routines in contactor deal with two types of measurements. One is absolute and the other ratiometric. The absolute measurement is based on the ratio of the ADC readings of the input to be measured and the Vref, and knowing the value of Vref. The result is independent of Vdd. The 5v and 12v supply voltages are examples. (For the high voltage sensor, the three high voltages would be absolute types of measurements.)

The ratiometric type of input is based on the ratio of the input to be measured and the ADC reading of the 5v supply. The Hall-effect current sensors use the 5v supply and output a signal from 0 – close to 5v. Their output is divided to limit the max voltage to the ADC to about 2.9v. The 5v supply is divided similarly. Knowing the ratio of the two dividers the result is independent of the 5v supply variations.

To simplify calibration, calibration parameters are specified in floating pt format, e.g. 5.03 volts. For computations the floating pt is scaled by 16b to uint32\_t.

A single pole iir filter, using int32\_t, and expecting 16b numbers is used to smooth the summed ADC readings. The filter time constant and integer scaling are parameter inputs for each ADC channel.

(NOTE: filtering is provided by currently being skipped and the unfiltered readings used.)

For the contactor states that check for voltages above/below some threshold value, the threshold that is specified in floating pt is converted to a scaled uint32\_t.

The internal temperature sensor is currently measured, but the formula to convert it to temperature is currently not coded. The code in F1 series is somewhat different than the F4 series that has calibration constants in the system memory. The temperature is chip temperature and as such not of much value for measuring ambient temperature, however it has use for compensating the internal reference voltage.

**I. CAN msgs**

ContactorTask.h has a summary of the CAN msg in the comments at the beginning of the file.

1. Function commands

The function command is a diagnostic/maintenance type of command. The early thinking was that one CAN id would be used to take of care of communication involving such things a online parameter update and retrieving variables that are not part of the operational aspect of the function. The hex CAN id is added to the database table CANID\_INSERT.sql, and used in the initialization of the contactor parameter struct (see contactor\_idx\_v\_struct.c).

For contactor retrieving raw ADC readings to be used in conjunction with an applied calibration voltage or current would be the main use of the function command.

cid\_cmd\_i (incoming) specifies a request in the first payload byte

cid\_cmd\_r (response) returns the requested readings

The first payload request byte code is specified in contactor\_cmd\_msg.c --

enum CONTACTOR\_CMD\_CODES

{

ADCRAW5V, // PA0 IN0 - 5V sensor supply

ADCRAWCUR1, // PA5 IN5 - Current sensor: total battery current

ADCRAWCUR2, // PA6 IN6 - Current sensor: motor

ADCRAW12V, // PA7 IN7 - +12 Raw power to board

ADCINTERNALTEMP, // IN17 - Internal temperature sensor

ADCINTERNALVREF, // IN18 - Internal voltage reference

UARTWHV1, // Battery voltage

UARTWHV2, // DMOC +

UARTWHV3, // DMOC -

CAL5V, // 5V supply

CAL12V, // CAN raw 12v supply

};

ADC requests send back a seven byte payload--

byte[0] : return of request code

byte[1]-[2]: 16b ADC sum reading

byte[2]-[6]: Float of calibrated reading

The high-voltage readings (essentially the same as ADC)--

byte[0] : return of request code

byte[1]-[2]: 16b sensor ADC reading sent via uart

byte[2]-[6]: Float of calibrated reading

CAL5V and CAL12V are the same as the ADC requests.

2. Polling

The contactor sends can send two CAN msgs in response to the CAN poll msg cid\_gps\_sync used for polling the POD. See contactor\_idx\_v\_struct.c--

// CANID\_HEX CANID\_NAME CAN\_MSG\_FMT DESCRIPTION

p->cid\_msg1 = 0x50400000; // CANID\_MSG\_CNTCTR1V : FF\_FF : Contactor1: poll response: High voltage1:Current sensor1

p->cid\_msg2 = 0x50600000; // CANID\_MSG\_CNTCTR1A : FF\_FF : Contactor1: poll response: battery gnd to: DMOC+, DMOC-

One is for battery string voltage 1 and battery current 1 and the other for voltages between battery ground and DMOC+, DMOC-. These are sent in an eight byte payload as a pair of floats with units, volts & amps.

3. Heartbeat

Heartbeat messages have the same payload format as the Polled msgs, but lower priority CAN ids.

p->cid\_hb1 = 0xFF800000; // CANID\_HB\_CNTCTR1V : FF\_FF : Contactor1: Heartbeat: High voltage1:Current sensor1

p->cid\_hb2 = 0xFF000000; // CANID\_HB\_CNTCTR1A : FF\_FF : Contactor1: Heartbeat: High voltage2:Current sensor2

4. Keepalive/command

The keepalive msg carries the command to connect/disconnect/reset and also serves to let the contactor know that there is communication with the controller.

cid\_keepalive\_i (incoming) command

cid\_keepalive\_r (response) status

The response sends the current status. It is sent in response to each incoming keepalive msg and also whenever the state changes. If the keepalive msgs cease the keepalive timer timeout causes a status msg to be sent, making it like a heartbeat msg with the status.

**J. cangateCON—cangate version for testing contactor**

cangateCON is located in--

~/GliderWinchCommons/embed/svn\_discoveryf4/PC/sensor/cangateCON/trunk

This version of cangate has some commands specific for testing the contactor. To compile cangateCON

cd to the above directory and type ./mm<enter>

To execute cangateCON (after setting up a gateway and hub-server)--

cd ~/GliderWinchCommons/embed/svn\_discoveryf4/PC/sensor/cangateCON/trunk

./cangateCON 127.0.0.1 32123

More on setting up the whole test rig later.

**command k**

k - CONTACTOR: keep-alive

ka - Begin sending keep-alive msgs

kx - Stop keep-alive msgs

k0 - disconnect

k1 - connect

k2 - reset

This command handles the keepalive CAN msgs to/from the contactor.

Entering ‘k’ starts the command and keepalive msgs from contactor are decoded and displayed. The msgs from contactor may be either responses to keepalive msgs sent by cangateCON, or heartbeat msgs which are sent when the contactor keepalive timer has timed out, i.e. contactor was not receiveing keepalive msgs.

The keepalive msgs from contactor carry the status which is displayed, both with the numeric fields and ascii text for the codes that show the status of the contactor program progress.

After starting command ‘k’,

ka – starts sending keepalive msgs. The rate of status msgs from contactor will be faster than the heartbeat msgs.

kx – stops sending keepalive msgs.

k0 – disconnect command

returns contactor to disconnected state

k1 – connect command

starts the sequence for connecting

k2 – reset fault

When contactor goes into a fault state it remains until ‘k2’ sets the reset bit in the command msg.

**command b**

b - CONTACTOR: display polled msgs

Entering ‘b’ alone displays options

bp<enter>

CONTACTOR msg gps msg POLLED

default CAN IDs 50400000 50600000

bp aaaaaaaa bbbbbbbb<enter>

a = CAN ID msg1, b = CAN ID msg2

bh<enter>

CONTACTOR msg gps msg HEARTBEAT

default CAN IDs FF800000 FF000000

bh aaaaaaaa bbbbbbbb<enter>

a = CAN ID msg1, b = CAN ID msg2

The default CAN id for two CAN msgs from contactor are displayed. These can be overridden by entering ‘bp’ followed by the two CAN addresses. Note, one could display CAN payloads from any CAN msg(s) where the payload is two floats, i.e. the display is rather generic.

Entering **‘**bp’ without additional data uses the default CAN addresses and displays polled msgs contactor sends in response to the gps/time sync msgs (64/sec). The polled response is two msgs that carry the three high voltage readings plus the current, all as floats in volts, or amps.

For example entering bp displayed, HV1, current, HV2, HV3 at 64 lines/sec--

Display two payload float pairs using CAN IDs: 50400000 50600000

12.468 0.000 0.027 0.027

12.458 0.000 0.027 0.027

…

Note that each line is payload from two CAN msgs and expects both msgs.

bh – same display as bp, but for heartbeats.

**command s**

s - CONTACTOR: command request to retrieve all readings

Send the maintenance commands and display the responses. CangateCON sends CAN msg to contactor. The first payload byte contains the code number for data. The response repeats the request code and adds the data.

ID: CAN node sends to us: E3600000 We send to CAN node: E360000C

TX: e360000c 1 0 : RX E3600000 7 00 FB F9 00 00 00 00 0: 63995 0.00 ADCRAW5V, PA0 IN0 - 5V sensor supply

TX: e360000c 1 1 : RX E3600000 7 01 BB B1 00 00 00 00 1: 45499 0.0 ADCRAWCUR1, PA2 IN2 - Current sensor: total battery current

TX: e360000c 1 2 : RX E3600000 7 02 C3 90 00 00 00 00 2: 37059 0.0 ADCRAWCUR2, PA4 IN4 - Current sensor: motor

TX: e360000c 1 3 : RX E3600000 7 03 75 3F 00 00 00 00 3: 16245 0.0 ADCRAW12V, PA7 IN7 - +12 Raw power to board

TX: e360000c 1 4 : RX E3600000 7 04 27 69 B6 23 0D 42 4: 26919 35.3 ADCINTERNALTEMP, IN17 - Internal temperature sensor

TX: e360000c 1 5 : RX E3600000 7 05 32 5D 00 00 00 00 5: 23858 0.000 ADCINTERNALVREF, IN18 - Internal voltage reference

TX: e360000c 1 6 : RX E3600000 7 06 38 05 DE 55 47 41 6: 1336 12.46 UARTWHV1, Battery voltage

TX: e360000c 1 7 : RX E3600000 7 07 03 00 ED 77 DE 3C 7: 3 0.03 UARTWHV2, DMOC +

TX: e360000c 1 8 : RX E3600000 7 08 03 00 69 C7 DE 3C 8: 3 0.03 UARTWHV3, DMOC -

TX: e360000c 1 9 : RX E3600000 7 09 00 00 00 00 00 00 9: 0 0.000 CAL5V, 5V supply

TX: e360000c 1 10 : RX E3600000 7 0A 00 00 00 00 00 0010: 0 0.000 CAL12V, CAN raw 12v supply

TX: e360000c 1 11 : RX E3600000 7 0B

Two instances of cangateCON, one with command k running, and one with command b running allow monitoring the pre-charging and status.

**K. Running the test setup**

The following three units are needed to run the test setup. Other units can also be on the CAN bus.

**1. CAN bus**

**a)** gps/logger generating time sync msgs

To observe the polled contactor msgs it needs time sync CAN msgs from the gps/logger.

**b)** gateway

**c)** contactor BP Motherboard

**2. USB**

A usb hub is needed for the following--

**a)** gateway FTDI module (for hub-server/socat to CAN bus)

**b)** contactor BP Motherboard: console monitoring

**c)** Discovery F4 STLINK (for reprogramming contactor)

**3. contactor wiring**

**a)** Wired to simulator, BP high voltage sensor modules and fethe subboards as described earlier.

**b)** Set jumpers for mode--

all to left = two contactor mode

all to right = contactor mode

**c)** Set use/ignore high voltage readings jump

not jumpered = use high voltage readings

jumper present = ignore high voltage reading logice

**d)** 12v to simulator module (which also powers the fethe boards).

**e)** Check that the following LEDs are “working”--

Both Blue Pill modules on the Montherboard and high voltage sensor should have a steady red led (showing it is powered) and one per second flashing green led (showing the program is running OK).

Each fethe subboard should have a steady bright green LED on (12v fet driver power).

Simulator module should have the dull green LED on (12v power).

**4. Open window and Start hub-server and cangateCON**

**a)** Determine port number of FTDI from gateway.

List serial ports devices--

ls /dev/ttyUSB\*

Pull USB plug from the hub for gateway.

Up arrow and run above command again.

Note which device missing, e.g. dev/ttyUSB1

**b)** Start hub-server

hub-socat\_b 32123 /dev/ttyUSB1 2000000

32123 = port number to be used

/dev/ttyUSB1 = gateway port as determined above

2000000 = baud rate for gateway<->socat

**c)** Check that hub-server is working

nc localhost 32123

CAN msgs should be seen ripping by.

**d)** Start an instance of cangateCON

Go to cangateCON directory--

cd ~/GliderWinchCommons/embed/svn\_f~/GliderWinchCommons/embed/svn\_discoveryf4/PC/sensor/cangateCON/trunk

./cangateCON 127.0.0.1 32123

The menu should come up--

### CANGATECON ### 08/03/2019

BAUDRATE is FIXED AT: 2000000 baud

argc 3

0 ./cangateCON

1 127.0.0.1

2 32123

Socket opened OK. IP: 127.0.0.1 PORT: 32123

CANID sql file opened!: ../../../../../svn\_common/trunk/db/CANID\_INSERT.sql

Control C to break & exit

a - ascii monitor of a CAN unit

d - list raw msgs

f - display fix: (e.g. f<enter>, or f E2600000<enter>

h - Photodetector level histogram from sensor

l - list unix time/date in heartbeat time msgs (l e1000000)

n - list msg id's and msg ct during 1 sec (coarse computer timing)

u - list msg id's and msg ct between CAN 1 sec time mgs (e.g. u 00600000)

m - list msgs for id entered 'm xxxxxxxx (CAN ID as 8 hex digits)'

q - Identify received msgs from CANID.sql file

r - send high priority RESET

s - CONTACTOR: command request to retrieve all readings

k - CONTACTOR: keep-alive

ka - Begin sending keep-alive msgs

kx - Stop keep-alive msgs

k0 - disconnect

k1 - connect

k2 - reset

b - CONTACTOR: display polled msgs

c - request & display launch parameters

w - list msgs float (wf) or integer (wi) payload with payload byte offset (wi1 E1800000)

x - cancel command

Control C to quit program

Enter command u--

CAN msg counts should appear if gps/logger is putting out time sync msgs. If not,

Try command n.

Enter command k for contactor. Something similar to the following should begin--

Using DEFAULT CAN IDs: Send E3800000 Rcv E3C00000

to use different CAN IDs, enter exactly in the following format

k aaaaaaaa bbbbbbbb<enter>

where: a = ID PC sends, b = ID PC receives

KA response (status): E3C00000 3 80 00 03 : 0 3 DISCONNECTED : NO\_FAULT

KA response (status): E3C00000 3 80 00 03 : 0 3 DISCONNECTED : NO\_FAULT

KA response (status): E3C00000 3 80 00 03 : 0 3 DISCONNECTED : NO\_FAULT

…

**e) Open a window and start minicom for console monitoring**

Do a ‘ls /dev/ttyUSB\*’, then pull the usb plug to the hub for the console FTDI module and repeat the command to see which port the console is on. Start minicom.

The console output depends on the sections of code enabled in the ‘main.c’ task--

void StartDefaultTask(void const \* argument)

For example, the following updates at about once per second--

# 7 Unused Task stack space--

#Stack: defaultTask--- 109

#Stack: SerialTaskSend 92

#Stack: CanTxTask----- 84

#Stack: MailboxTask--- 64

#Stack: ADCTask------- 85

#Stack: ContactorTask- 76

#Stack: SerialReceiveTask 70

#GetFreeHeapSize: total: 7200 free 600 8.3% used: 6600

63995 45457 37097 16271 27127 23872 : 27129 372.9

UART ctr: 531

uart\_hv 1337 1376 3

dscale0.0093252 0.0090523 0.0090649

hvcal 1222 1186 1188

hvc 1632592 1631936 3564

hv 1336 1376 3 <= from BP voltage sensor

dhvc 12.47 12.46 0.03 <= after calibration

ibattlow: 965 fbattlow: 9.00 hv[0]: 1336 battnow: 12.46

iprechgendv: 66 hv[IDXHV3].hvc: 3

HV by-pass pin: 1

THRES: 1752 78625 1631370 1633122 0

**5. Open terminal window to observe contactor voltages/current**

Start cangateCON. ‘cd’ to directory, as above, and start with same command line.

Enter command ‘bp’

The readings should be running past--

HV1 | current : HV2 | HV3

Press pushbutton to discharge capacitor. HV2 will likely drop somewhat. The latest offset was 0.027 volts.

**6. Operate contactor**

In the window with the ‘k’ command, running ‘ka’, it should be showing DISCONNECTED : NO\_FAULT

If it is showing a fault, enter ‘k2’ to reset the fault state.

Enter ‘k1’ to start the connection.

Two contactor mode--

HV2 should jump to the batttery string voltage (i.e. the 12v supply).

HV3 should jump from near zero to nearly equal to the HV1, the battery string (12v supply) reading and begin dropping as the capacitor charges.

The status msgs should show “CONNECTING”.

The substate code will step to 1 for the minimum duration pre-charge, regardless of voltage, followed by step 2 which begins checking for the voltage threshold for closing contactor #2. If the threshold is not reached before the time expires it goes to a fault state and with the status message showing the fault.

Varying the timing and voltage parameters and recompiling/flashing can be done to demonstrate the various fault conditions.

**L. Program compiling/flashing**

**1. Connect a DiscoveryF4 or Nucleo board to the Blue Pill programming port.**

**a)** Remove the two jumpers on the STLINK header on the DiscoveryF4 board.

**b)** Connect the three wires from the SWD (six pin) header on the DiscoveryF4 board to the Blue Pill header as detailed earlier. Note that the wire sequence reverses for one pair.

**c)** USB cable from usb hub on PC to STLINK connector of DiscoveryF4

**2. Open a window and cd to the project code--**

cd ~/GliderWinchItems/contactor

Start compilation--

./mm

On my computer success produces--

deh@ASUS:~/GliderWinchItems/contactor$ ./mm

make: Nothing to be done for 'all'.

make: Nothing to be done for 'all'.

---- PATHS for OPENOCD ----

OPENOCD\_SCRIPTS: /usr/share/openocd/scripts

OPENOCD\_FLAHSER: /usr/share/openocd/scripts/interface/stlink-v2.cfg

OPENOCD\_BOARD: /usr/share/openocd/scripts/board/olimex\_stm32\_h103.cfg

FLASH FILE NAME: /contactor.srec

\_FLASHNAME: /usr/bin/make

DIR\_SREC: /home/deh/GliderWinchItems/contactor/build

Open On-Chip Debugger 0.9.0 (2018-01-24-01:05)

Licensed under GNU GPL v2

For bug reports, read

http://openocd.org/doc/doxygen/bugs.html

Info : auto-selecting first available session transport "hla\_swd". To override use 'transport select <transport>'.

Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD

adapter speed: 1000 kHz

adapter\_nsrst\_delay: 100

none separate

Info : Unable to match requested speed 1000 kHz, using 950 kHz

Info : Unable to match requested speed 1000 kHz, using 950 kHz

Info : clock speed 950 kHz

Info : STLINK v2 JTAG v34 API v2 SWIM v0 VID 0x0483 PID 0x3748

Info : using stlink api v2

Info : Target voltage: 2.889882

Info : stm32f1x.cpu: hardware has 6 breakpoints, 4 watchpoints

target state: halted

target halted due to debug-request, current mode: Thread

xPSR: 0x01000000 pc: 0x08009f58 msp: 0x20005000

auto erase enabled

Info : device id = 0x20036410

Info : flash size = 64kbytes

Info : Padding image section 0 with 4 bytes

target state: halted

target halted due to breakpoint, current mode: Thread

xPSR: 0x61000000 pc: 0x2000003a msp: 0x20005000

wrote 62464 bytes from file /home/deh/GliderWinchItems/contactor/build/contactor.srec in 3.646678s (16.728 KiB/s)

shutdown command invoked

deh@ASUS:~/GliderWinchItems/contactor$

During the flashing phase the green LED on the Blue Pill module board will stop. If the program doesn’t crash when it starts up the green LED begins one second on/off flashing again.

Most errors are trapped with the ‘morse.c’ which flashes a two digit code number on the green LED. That code number can be found in the source code to locate where it went wrong. These are generally initialization faults.

If the green LED doesn’t flash there is likely FreeRTOS not happy. The most common problem is there is not enough sram for either system stack or the heap. The allocation can be adjusted

at line 106 in--

~/GliderWinchItems/contactor/Inc/FreeRTOSConfig.h

However, re-running STM32CubeMX will overwrite this if the code is regenerated. Therefore, if a new value is settled upon, go to STM32CubeMX and change it in the FreeRTOS section.

**2. Modify parameters**

Parameters can be modified to test the functioning of different fault conditions with the simulation module. Modifications will be necessary to adapt it to the real winch.

The parameters of interest are located in the directory--

~/GliderWinchItems/contactor/Ourtasks

The parameters are loaded into a sram struct during initialization. The values are currently loaded by executing the code in--

contactor\_idx\_v\_struct.c

Later, this file could be replaced with loading from a flat file in high-flash that can be updated via CAN, and is independent of re-flashing the program code.

Other files associated with contactor\_idx\_v\_struct.c are--

ContactorTask.h

contactor\_func\_init.c

hwconfig has bit positions for the configurations. All bits off is the default. These are defined in--

contactor\_idx\_v\_struct.h

fbattlow Defines the voltage below which will cause a low battery fault when in the disconnected state.

The section on timings are in milliseconds, which is converted to parameters to timer ticks during the execution of contactor\_func\_init.c during initialization.

Of particular interest is--

ddiffb4 – defines the voltage across the contactor (difference before closing) that will end the pre-charging state. If the timer times out during this state, it means the voltage had been expected to be reached, but failed; therefore, it goes to a fault state.

fdiffafter – defines the voltage across final contactor after it has been energized and after a short delay for closure it should be closed, above which results in a fault, which means the contactor may not have closed. There is an issue here with pwm’ing the coil. If the contactor is pwm’ed and the pwm percentage is too low the contactor could drop out. The state logic doesn’t cover this situation.

prechgmin\_t – Duration for the minimum pre-charge time. The first phase of pre-charge is always this duration.

prechgmax\_t – Following the above minimum duration, this duration times the state that the voltage is checked for terminating the pre-charge. The pre-charge state will be terminated if the voltage threshold is reached, in which case it was successful, or if the timer times out, in which case it failed.

If the jumper for ignoring the high voltage readings is in place, the pre-chargemax time will take place and it will not go to a fault state when the timer times out. Net—the pre-charge duration would be the sum of prechgmin\_t plus prechgmax\_t.

close1\_t, close2\_t – These are set long to be able to observer the LED on the simulator module.

PWM parameters are in percentage (0-100). The coil is energized at full ON for close1\_t or close2\_t duration. At the end of that time duration the FET is pwm’ed at the parameter percentage. The simulator coils drop out at around 40%.

fpwmpct1, fpwmpct2 – percent pwm ON time.

Note: A minor simplification could be made by not specifying if the coils are to be pwm’ed, but instead always pwm the coil (after the initial 100% pull duration), and specify the pwm percentage as 100% (which results in continuous ON) if no pwm is desired.

Calibrations

The Blue Pill high voltage sensor sends ADC measurement, and these are then filtered, and adjusted for offset and scale to yield voltage. The defaultTask in contactor ‘main.c’ lists the raw adc readings which can be used to adjust the calibration for the three voltages. The present calibration was done with the 12v supply and can expected to be close for voltages in the 300-400v range.