**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.

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

**B. Hardware**

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

1. Several contactor configurations are accommodated--

hwconfig in the parameter list (contactor\_idx\_v\_struct.h) has bit positions for the configurations.

a. Two contactors

1). With, or without, auxiliary contacts.

2). With, or without, pwm of the coil drive.

Contactor #1 connects the battery string plus to DMOC plus.

Contactor #2 connects battery string minus to DMOC minus.

Pre-charge resistor is across contactor #2 contacts.

b. One contactor, one small pre-charge relay

1). Contactor with, or without, aux contacts and with or without pwm.

The contactor connects the battery string plus to DMOC plus.

2) Small relay connects battery string plus to pre-charge resistor and the pre-charge resistor connects to the DMOC plus.

The DMOC minus connects directly to the battery string minus.

Note: For program purposes, the contactor is designated as contactor #2 (since it closes the high current path to the DMOC after pre-charge), and the relay designated as contactor #1 since closing it begins the pre-charge.

2. Isolated high voltage

High voltage is measured and sent to the ContactorTask via an isolated uart RX (usart3). Three voltages are from a common ground on the battery string are measured and sent as a line (maybe binary).

a. “hv1” - battery string plus

b. “hv2” - DMOC plus

c. “hv3 ” - DMOC minus

hv3 measures the voltage across the pre-charge resistor when there are two contactors, i.e. the voltage across contactor #2 contacts.

The difference between hv1 and hv2 measures the voltage across the pre-charge resistor in the one contactor configuration.

3. DMOC enable

A small fet pulls DMOC enable line, or possibly a small relay for daisy-chaining a DMOC enable.

4. CAN

A non-isolated CAN driver.

5. Current sensors

Provision for two Hall-effect current sensors.

a. Battery string current

b. spare

6. Console output uart (usart1)

Debugging, and monitoring with PC.

**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.

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.