Overview

This presents some ideas on the ‘dynamometer’ project, based on STM32CubeMX, FreeRTOS, and interface & support routines developed in the ‘mxusartusbcan’ project (see ‘Brief.docx’ in ‘msusartusbcan’ directory for more detail). Some of the approaches take into account using the routines beyond the dynamometer project to the winch master controller and other CAN projects. All of this is tentative and details are likely to change as implementation progresses.

A. Infrastructure Tasks

The following outlines how these features developed in the ‘mxusartusbcan’ project are used for the ‘dynamometer’ project.

1. CanTxTask

Passes CAN msgs other tasks want to send to the interface routine (‘can\_iface.c’) to the HAL CAN routine (‘stm32F4xx\_hal\_can’).

The CAN msg carries the pointer to the HAL CAN module handle in “our” control block, which sends the CAN msg to the CAN1 or CAN2 modules and hence the CAN bus connected to the module..

2. CanRxTask

In ‘mxusartusbcan’ incoming CAN msgs are placed in a FreeRTOS queue by the functions in the ‘can\_iface.c’ routine. For ‘dynamometer’, ‘can\_iface.c’ is revised to use a circular buffer and notification to a ‘MailboxTask’ rather than a using a FreeRTOS queue. Besides lower overhead, there can be multiple “take” pointers to the circular buffer, alllowing multiple tasks can access buffered msgs. There is no overwrite protection so the buffer has to be large enough so that the “take” position in the circular buffer for the slowest/lowest priority task is not overwritten by “add”. The buffer is set during initialization for ‘can\_iface.c’. The ‘take’ pointers are added and initialized by tasks, thereby allowing multiple tasks access to the buffering without the need for a predetermined number to be compiled in to the code.

CAN msgs are “added” to the circular buffer during the callbacks for RX0, RX1, plus the loopback of CAN msgs that were sent if compiled with the option set. The circular buffer is loaded under interrupt and tasks “take” msgs from the buffer at FreeRTOS task level priority. ‘MailboxTask’ is described later.

The circular buffer with the CAN msgs includes more than the basic CAN msg (i.e. ‘struct CANRCVBUF’). The pointer to “our” control block, for which there is one for each CAN module, i.e. CAN1, CAN2, is added, and provides a CAN1/CAN2 identification. A time-of-arrival in the form of the 32b DTW system counter is added. Tasks can read this DTW count and check for “stale” readings. This is covered in the description below on ‘MailboxTask’.

B. Project specific tasks

Except for the ‘defaultTask’ the tasks are created “manually”, i.e. not by STM32CubeMX.

1. defaultTask

Created: STM32CubeMX. Priority: osPriorityIdle (-3)

It will be used for the lowest priority computing, such as formatting data for output monitoring on usart6 using ‘yprintf’ (wrapper with semaphores for ‘vsnprintf’), stack high water mark checking, etc.

2. ControlTask

Created: manually Priority: osPriorityNormal (0)

Handles Speed and Torque control using the readings of the two ADC readings. The ADC DMA callback does a 16 reading summation under interrupt followed by a notification. The ControlTask can then do further filtering, calibration, etc., triggered by the notification (see Task02 in ‘mxusartusbcan’).

The ControlTask sends the periodic “keepalive” notifications to the MotorControlTask. The MotorControlTask then shuts down the motors if the notifications are missed. Likewise, the MotorControlTask sending periodic commands to the DMOCs and the DMOCs shutdown if the commands are not received in time.

The ControlTask sends the polling CAN msg; 64/sec. The POD/CAN unit sends back two CAN msgs with the load-cell readings as calibrated floats.

Since the ControlTask is the source for polling cycle, it might also send out command msgs to the DMOCs to retrieve DMOC measurements of current, voltage, speed, torque, etc.

The ControlTask could be integrated with the MotorConrolTask, however the functions of this ControlTask will eventually be taken over by the Master Controller, (MC) so separation might help the transition to the MC.

3. MotorControlTask

Created: manually Priority: osPriorityNormal (0)

This tasks maintains the connection logic for each DMOC state, e.g. “ping” msgs from the DMOCs indicate the DMOC is operational but in a not connected state.

Notification from the ControlTask causes the MotorControlTask to send the speed & torque commands.

4. MailboxTask

Created: manually Priority: osPriorityNormal (3)

The main goal of the Mailbox task is to provide tasks with readings from CAN msgs such that the mailbox has the latest reading, plus a means for detecting if the reading is new, and/or too much time has elapsed since the last update, or a task or tasks, need immediate notification. This would be one of the highest FreeRTOS priority tasks as CAN msgs involved timing.

As described above in A.2., CanRxTask, the Mailbox task gets pointer to CAN msgs on the circular buffer of incoming CAN msgs. This CAN msg has the CAN id, dlc, payload, “our” control block pointer which identifies the CAN module, hence bus, plus the 32b DTW time-of-arrival timestamp.

When a CAN msg is added to the circular buffer in ‘can\_iface.c’, a notification to ‘MailboxTask’ is made. ‘MailboxTask’ takes a CAN msgs from the circular buffer and does a lookup in the list of CAN bus, and CAN ids. If it appears in the list, it converts the payload and loads the mailbox with the payload converted to readings, plus the time-of-arrival DTW and increments an update counter in the mailbox. If the mailbox has the task handle and notification bit setup, a notification is made.

Since there can be multiple tasks that need notification when a mailbox is updated, a linked list for each mailbox provides a means of stepping down the list making notifications for each task. If notifications are not needed, any number of tasks can access the mailbox and check the time-of-arrival and update counter, however to access the mailbox each tasks needs to have the pointer to the mailbox. That pointer can be retrieved by a call to “add a mailbox”. The call to “add a mailbox” includes the CAN id, so the initialization can check for duplicates, the existing pointer is returned and a new mailbox not created. If a notification is desired, the call to “add a mailbox” includes the notification bit to be used.

There are several levels of initialization involved. One is the creation of ‘MailboxTask’ which is performed in ‘main’ before the task scheduler is started. The initialization would include a call to ‘can\_iface.c’ to add a “take” pointer for the circular buffer of CAN msgs. When ‘MailboxTask’ receives a notification it does a ‘while’ to remove CAN msgs until it is caught up.

The list of CAN bus and CAN ids to be selected is generated by the tasks during initialization, either after the task starts and before the endless loop, or after the task was created but before the scheduler started. The list is sorted so that the binary lookup can executed rapidly. The task level initialization includes the notification bit to be used by the task if notifications are used, and as such requires that the task handle has been created, hence this is most conveniently done after the task has just started.

When a task polls, it may want to know if the mailbox reading is too old, in which case it can check the current DTW time count versus the one stored in the mailbox. Or, it may need to know if there is a new reading, in which case the task compares the mailbox update count with the previous count which was saved locally. Any number of tasks can be using the CAN mailbox in these two modes.

If the notification handle is NULL or the notification bit word zero, ‘MailboxTask’ would not execute a notification. However, if given the task and notification bit was set, the task would be notified. This would unblock the task waiting with on ‘NotifyWait’ and the mailbox would be dealt without further delay. The notification bit(s) that are found to be set when the task unblocks identify the mailbox(es) that were in the notifications.

A further detail in the implementation is to have an array of pointers to the mailbox structs (which were malloc’d). When the mailboxes are created by calls at the task level initialization the mailbox is loaded with the CAN id and CAN control block pointer (CAN module identification), and the pointer added to the array. The array of pointers is sorted on CAN id. The array of pointers is then used for the lookup with CAN that was taken off the circular buffer. CAN msgs not in the list are simply ignored for mailbox updating.

Another possibility: Do this as two steps. The first step is to select CAN msgs for a task or set of related tasks, and have a second level deal with conversion to mailboxes, or something else. If the processor/board is handling more than one winch function then some of the tasks are not interest in sme of the CAN msgs. Rather than having each function go through their list of CAN msgs for each arriving CAN msg, this approach would supply each task with only the CAN msgs of interest to the task.

The situation of multiple winch drums raises the issue of CAN msgs with different CAN ids being used for the same basic function, i.e. the control loop routine would deal with, e.g. cable tension, but the mailbox for “cable tension” would be updated by CAN msgs with different CAN ids and would depend on the drum that was active. If in this example a single mailbox is used for cable tension, then the CAN msg selection would take place in the ‘MailboxTask’. The alternative would be have mailboxes for each drum cable tension, and control loop that is using the mailbox readings select the mailbox associated with the active drum. For retrieve where the cable tension is being measured for all drums there would need to a mailbox for each drum.

5. GatewayTask

Created: manually Priority: osPriorityNormal (0)

This task takes incoming CAN msgs from the circular buffer with CAN msgs; converts them to the ascii/hex format and sends them out on the high speed usart (usart2 @ 2E6 baud) to the PC . (‘gateway\_CANtoPC’, used in Task03, in ‘mxusartusbcan’.)

Ascii/hex CAN msgs coming in from the PC are converted to the binary CAN format and passed on to CanTxTask for sending.

To deal with the CAN1/CAN2 issue ascii/hex format robs the low two bits from the sequence number of the CAN msg and uses that for a CAN1/2/3 identifcation (STM32F4 series has two CAN modules, and the STM32F7 has three) . NOTE: this requires some modification of ‘cangate’ to either have a CAN1/2/3 version, or way to avoid the original ‘cangate’ from flagging a sequence number error.

An alternative is to use the gateway to the PC in a one bus mode.

Use of the usb CDC (virtual com port) for gateway purposes requires some additional work, as the buffering/interrupt/polling is somewhat different with the STM HAL routine for usb-cdc and the usart routines.