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’). CAN msgs are sent in CAN priority order.

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 task with the most delay 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. For this project ‘MailboxTask’, described later, is the main task “taking” msgs from the circular buffer; a gateway-to-PC task would be another common task with a need for dealing with all CAN msgs.

Multiple CAN modules, i.e. CAN1, CAN2, and CAN3 (‘f446 series), is handled by implementing a circular buffer for each CAN module. Partially redundant, the CAN msg struct used in the circular buffer carries the pointer to “our” CAN control block (which includes a pointer to the STM32 CAN control block), and that essentially identifies the CAN module as well.

A time-of-arrival time-stamp in the form of the 32b DTW system counter is included in the CAN msg struct in the circular buffer. 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’, these tasks are created “manually”, i.e. not by STM32CubeMX, and reside in the ‘Ourtasks’ directory. STM32CubeMX provides a skeleton task in ‘main’ when the task is specified in the ‘MX setup gui and ‘main’ quickly becomes very large. Most of the code in ‘main’ concerns initialization so having tasks as separate files helps modularize the project.

1. defaultTask

Created: STM32CubeMX. Priority: osPriorityIdle (-3)

The task is always setup by STM32CubeMX when FreeRTOS is selected. It resides in ‘main’. 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.

The keepalive notifications could be combined with polling if the DMOC keepalive duration can be set to span the 1/64th sec duration.

For the Master Controller (MC) the routine is interested in readings that might be a combination of the motors, therefore polling of the parameters from the DMOCs by the ‘MotorControlTask’ would be in order.

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 set to a high FreeRTOS task priority.

Influencing the approach in the following discussion is that it is desirable that ‘MailboxTask’ handle the Master Controller project as well as the ‘dynamometer’ and ‘dynamometer’ serves to refine and prove the approach.

From the ‘ControlTask’ viewpoint, the interest is in “readings,” not CAN msgs. Mailboxes based on readings is a knotty problem for the more general case where CAN msgs might carry more than one reading. E.g. shaft encoding carries a shaft encoder count (signed 32b int) and speed (32b float), combined into one payload since they occur at the same point in time. The CAN msg would need to update two mailboxes.

Rather than deal with the multiple readings per CAN msg issue, the CAN msgs could be limited to just one reading per CAN msg. This increases the CAN bus load a bit. In the case of battery cell voltages the inefficiency makes it a dubious tradeoff. A compromise is for the mailbox to allow for two readings.

Another complicating issue is if the control program is using a a mailbox, e.g., cable speed, it the source of the reading changes when the drum changes, i.e. the ‘MailboxTask’ would be updating the cable speed mailbox from different CAN msg IDs, conditional upon the active drum. However for the retrieve, where all drums might be active, (but in low tension, low speed), the control program would be interested in all mailboxes. This latter situation opts for multiple mailbox for the same class of reading, and the control program selects the mailbox based on active drum. (The same issue applies to multiple motors and battery banks.)

At this point, the approach to be taken is to take incoming CAN msgs; look them in a table which identifies them as associate with a mailbox. Those not found in the table are discarded. Tasks have a pointer to mailboxes and a mailbox can be used by multiple tasks, so it provides a means for distributing CAN msgs to unrelated tasks. (A separate task would be sending all CAN msgs to the PC for the gateway function, if the PC gateway is included. When all CAN msgs are to be used the ‘MailboxTask’ isn’t needed and the CAN msgs can be taken directly from the circular buffer receiving the incoming CAN msgs.) (Another consideration is the CAN bus gateway, where incoming CAN msgs on one bus are selectively sent to the other bus or buses if there are three CAN modules. This is a mailbox type of process where the CAN id has to be identified and either discarded or directed to another CAN bus.)

A CAN msg found in the table has its payload parsed, according to the code in the table. PAYLOAD\_TYPE\_INSERT.sql already exists and carries the name v code for the payload types (26) encountered so far. This table also has to associate the CAN msg with a mailbox. A list of mailboxes could be maintained by adding a .sql file to the database that associates a mailbox name with CANID, and payload reading. Multiple readings in a CAN msg would have multiple entries in this table. The project would have a table using the #defines generated from the database tables that specifies the mailbox name. In the case of multiple drums, there would be multiple tables of identical layout (i.e. the same struct) that can be selected by drum number.

Moving to some details about some in progress work on implementation--

Work on ‘MailboxTask’ implementation began with the a CAN msg with limited payload decoding mailbox rather than “readings” mailbox. 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. The mailbox struct includes an msg update counter, and pointer to a linked list of notification blocks; if more than one task needs to updated when the CAN msg arrives the linked list provides the task-handle and notification bit for the each of the task notifications.

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. For tasks that do not need a notification access can be made at any time and the time-of-arrival or update counter used to determine if the reading is stale or now. To do this 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, however if a notification is desired, the call to “add a mailbox” includes the notification bit to be used it is added to the linked list.

There are several levels of initialization involved. The first 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’ has started and is in the endless loop it receives a notification and 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, 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. For eight drums the list could in the 30-40 range. For a single drum the binary lookup wouldn’t pay. For the dynamometer, the list is probably about 6 per CAN bus and a binary search isn’t warranted. 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 calloc’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 interested in some 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.

Simplifications from compromises--

The pure form is to have a mailbox only contain a single reading, and be agnostic entirely to CAN msgs. With this a lot of details get push out of the “user” task and in to the infrastructure tasks. If instead of a mailbox holding just one reading, the mailbox holds the readings for a CAN msg, the naming and association of mailbox readings with CAN ids can be eliminated. That still leaves the problem of various payload formats, e.g. the reading is a float, but the four bytes for the float might start at payload[0], [1], or [2]. If the bytes preceding the float were jettisoned then a single four byte element in the mailbox would suffice. If a second four byte element is added, two readings could be accommodated and this would cover most cases. Furthermore, if the readings are all converted to ints/uints and floats and presented in a union the “user” task can access them, but needs to know which type to use. E.g. suppose during the task initialization it “adds” a mailbox, which returns a pointer to the mailbox, e.g. p\_tension. The mailbox might ad accessed like this --

ften\_belt = p\_tension->u.f1; // Get first float

ften\_mtr = p\_tension->u.f2; // Get second float

This requires the using program to know more about the payload than if the mailbox was simply a single reading of ‘tension\_belt’.

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 CAN bus mode and avoid the CAN1/2 issue, but requires two gateways or gatewayfunctions (i.e. one could run two usarts to the PC, each carrying traffic for one CAN bus).

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.