Project Progress report:

July 2013

# Previous work

My most recent Project Proposal (July 2013) outlined the following work that had been carried out on the project:

* Counter mechanism to compensate for varying message cycle times.

# Work this since previous report

This month, I have focussed on implementing the algorithms that I developed in the filtering simulation in hardware. The device chosen for this is a Texas Instruments C2000 \*\*\*put proper reference here\*\*\*, since I shall be working with this processor on other projects. I had already developed a mailbox handling library for this processor, which needed minor modifications to enable the mailbox ID’s to be modified after device initialisation.

The embedded implementation uses a Time-Triggered Hybrid scheduler, and operates on 0.5 ms ticks, with the 2 tasks CAN-related tasks operating from the ISR. The high tick frequency allows for fast updating of the mailboxes, and the Hybrid scheduler means that longer background tasks can still run without colliding with the CAN tasks.

**#define** TICK\_PERIOD\_us (500)

**volatile** task\_t Tasks[] = {

{

handleCAN\_update, /\* function pointer \*/

2, /\* period in ticks \*/

125, /\* initial offset in ticks \*/

*IN\_ISR*

},

{

receiveCAN\_update, /\* function pointer \*/

2, /\* period in ticks \*/

126, /\* initial offset in ticks \*/

*IN\_ISR*

},

{

controlSCI\_update, /\* function pointer \*/

2, /\* period in ticks \*/

125, /\* initial offset in ticks \*/

*IN\_SCHEDULER*

},

};

handleCAN is involved in updating the states of the mailboxes based on registry flags. receiveCAN contains the logic associated with copying the CAN data into memory from the mailboxes, and updating the mailbox CAN IDs.

This logic is, in the main, a direct port of that presented for the simulation. The only differences being the logging sequence array expects a maximum of 64 CAN IDs, and the number of mailboxes used is automatically adjusted to half of the number of IDs in the sequence (shown to be the optimum filter size in the previous report).

**void** **receiveCAN\_update**(**void**){

Uint16 mailBox, messagePointer;

**static** Uint32 totalcounter = 0;

**if**(updateSequenceRequired\_G == 1){

/\* set number of mailboxes to use \*/

filterSize\_G = numRxCANMsgs\_G/FILTERSIZE\_RATIO;

**if**((numRxCANMsgs\_G%2)!=0){

filterSize\_G += 1;

}

**for**(mailBox=0; mailBox<filterSize\_G; mailBox++){

updateFilter(mailBox);

updateSequenceRequired\_G = 0;

}

}

**else**{

/\* look through mailboxes for pending messages \*/

**for**(mailBox=0; mailBox<filterSize\_G; mailBox++){

**if**(checkMailboxState(CANPORT\_A, mailBox) == *RX\_PENDING*){

/\* Find message pointer from mailbox shadow \*/

messagePointer = mailBoxFilters[mailBox].messagePointer;

/\* Count message hits \*/

CAN\_RxMessages[messagePointer].counter++;

totalcounter++;

/\* read the CAN data into buffer

(nothing done with the data, but nice to do this for realistic timing) \*/

readRxMailbox(CANPORT\_A, mailBox, CAN\_RxMessages[messagePointer].canData.rawData);

/\* update the filter for next required ID \*/

updateFilter(mailBox);

}

}

}

}

In addition to these tasks, a serial communications task runs as a co-operative task, which acts as a good simulation of heavy background tasks running in a real system. The serial comms task communicates with a PC via the processor’s Serial Communications Interface (SCI) using a RS232 to USB cable.

The PC runs a desktop application written in the ‘Processing’ programming language ([www.processing.org](http://www.processing.org)).

The desktop application has 3 functions:

* Transmits the ‘logging list’ to the hardware, allowing ‘remote configuration’ of the CAN filter.
* Logs the number of hits registered for each CAN ID in the logging list.
* Visualises the mapping between the hardware mailboxes and the logging list in close-to-realtime.

TI Development Board

PC

**PCAN USB to CAN bus adapter**  
🡪 Sample CAN trace 🡪

**USB to RS232 adapter**

🡪 Logging list 🡪

🡨 CAN ID hit counts 🡨

## Discoveries

The application’s visualisation highlighted a bug in the code ported from the Simulation. The algorithm that searched through the ID’s in the ‘filter’ to check for duplicates would result in IDfound = *FALSE* whenever there was no match for the very last filter location:

boolean\_t **updateFilter**(**unsigned** **int** filterPointer){

**static** **int** last\_i = -1;

**int** i, j;

flag\_t result = *FALSE*, IDfound = *FALSE*;

i = last\_i;

**do**{

**if**(i<(listSize-1)){

i++;

}

**else**{

i=0;

}

**for**(j = 0; j < FILTERSIZE; j++){

**if**(acceptanceFilter[j].canID == loggingSequence[i].canID){

IDfound = *TRUE*;

}

**else**{

IDfound = *FALSE*;

}

}

**if**(IDfound == *FALSE*){

loggingSequence[i].timer--;

}

**if**(loggingSequence[i].timer<=0){

result = *TRUE*;

}

}**while**((result == *FALSE*)&&(i != last\_i));

This meant that duplicate IDs were allowed in the filter and was clearly visible from the visualisations as more than one line leading to the same logging list location.

What is more interesting is that, when I removed this erroneous else{} the effectiveness of the algorithm reduced quite dramatically, giving a much higher standard deviation of the hit rates. This suggested that allowing duplicate ID’s would be beneficial as long as I could control them to prevent the same low frequency ID’s clogging the filter.

The reason for this could be that without the duplicates, IDs falling out of sequence will be missed, since the algorithm will only look for them after the expected preceding messages have arrived. By having two mailboxes configured to the same ID, even when one copy of the message has arrived, there is still another mailbox waiting for the next one, regardless of its sequential position in the CAN stream.

In the hardware implementation, the equivalent function uses the ‘.timer’ property of the CAN\_RxMessages struct. By allowing the timer to be decremented into minus values, we can control how many duplicates of the same ID are allowed in the filter, as follows:

#### In header:

**#define** DUPLICATES\_ALLOWED (1)

#### Main body:

**void** **updateFilter**(**unsigned** **int** filterPointer){

**static** int16 last\_sequencePointer = -1;

int16 sequencePointer;

boolean\_t result = *FALSE*;

**if**(updateSequenceRequired\_G == 1){

last\_sequencePointer = -1;

}

/\* Find next required CAN ID in sequence \*/

sequencePointer = last\_sequencePointer;

**do**{

/\* Wrap search \*/

**if**(sequencePointer<(numRxCANMsgs\_G-1)){

sequencePointer++;

}

**else**{

sequencePointer=0;

}

/\* .timer used to track number of duplicate ID’s in mailbox \*/

**if**(CAN\_RxMessages[sequencePointer].timer >= (0-DUPLICATES\_ALLOWED)){

CAN\_RxMessages[sequencePointer].timer--;

}

/\* ID ready to be inserted \*/

**if**((CAN\_RxMessages[sequencePointer].timer >= (0-DUPLICATES\_ALLOWED))

&&(CAN\_RxMessages[sequencePointer].timer <= 0)){

result = *TRUE*;

}

}**while**((result == *FALSE*)&&(sequencePointer != last\_sequencePointer));

/\* New ID found for mailbox \*/

**if**(result == *TRUE*){

/\* Message scheduling \*/

last\_sequencePointer = mailBoxFilters[filterPointer].messagePointer;

CAN\_RxMessages[last\_sequencePointer].timer = CAN\_RxMessages[last\_sequencePointer].timer\_reload;

/\* Real ID replacement \*/

configureRxMailbox(CANPORT\_A, filterPointer, ID\_STD, CAN\_RxMessages[sequencePointer].canID,

CAN\_RxMessages[sequencePointer].canDLC);

/\* ID replacement in shadow \*/

mailBoxFilters[filterPointer].canID = CAN\_RxMessages[sequencePointer].canID;

mailBoxFilters[filterPointer].messagePointer = sequencePointer;

}

last\_sequencePointer = sequencePointer;

}

This method, although having slightly lower hit rates, gives much more consistent results across all CAN IDs in the sample trace, indicated by the low standard deviation. Since the emphasis in this project is in making the hit rate more predictable, the slight reduction in hit rate is acceptable, giving a guaranteed 99.5%

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Previous Algorithm | | No duplicates allowed | 1 duplicate allowed | 2 duplicates allowed |
| 0x187 | | 99.76% | 99.856% | 99.550% | 99.585% |
| 0x188 | | 99.52% | 96.469% | 99.541% | 99.577% |
| 0x189 | | 99.76% | 99.898% | 99.543% | 99.579% |
| 0x18A | | 99.59% | 96.404% | 99.541% | 99.577% |
| 0x18B | | 99.76% | 99.909% | 99.541% | 99.577% |
| 0x18C | | 99.57% | 96.402% | 99.541% | 99.577% |
| 0x18D | | 99.76% | 99.936% | 99.546% | 99.581% |
| 0x18E | | 99.60% | 96.389% | 99.541% | 99.577% |
| 0x207 | | 99.77% | 99.911% | 99.550% | 99.585% |
| 0x209 | | 99.76% | 99.942% | 99.543% | 99.579% |
| 0x20B | | 99.75% | 99.936% | 99.541% | 99.577% |
| 0x20D | | 99.76% | 99.953% | 99.546% | 99.581% |
| 0x287 | | 99.76% | 99.925% | 99.550% | 99.585% |
| 0x289 | | 99.76% | 99.967% | 99.543% | 99.579% |
| 0x28B | | 99.75% | 99.973% | 99.541% | 99.577% |
| 0x28D | | 99.75% | 99.829% | 99.543% | 99.581% |
| 0x307 | | 99.77% | 99.967% | 99.550% | 99.585% |
| 0x309 | | 99.76% | 99.984% | 99.543% | 99.579% |
| 0x30B | | 99.76% | 99.971% | 99.541% | 99.577% |
| 0x30D | | 99.76% | 99.960% | 99.546% | 99.581% |
| 0x385 | | 99.59% | 96.353% | 99.539% | 99.574% |
| 0x387 | | 99.77% | 99.996% | 99.550% | 99.585% |
| 0x389 | | 99.76% | 99.982% | 99.543% | 99.579% |
| 0x38B | | 99.76% | 99.996% | 99.541% | 99.577% |
| 0x38D | | 99.77% | 99.998% | 99.546% | 99.581% |
| 0x407 | | 99.77% | 99.987% | 99.550% | 99.585% |
| 0x409 | | 99.76% | 99.980% | 99.543% | 99.579% |
| 0x40B | | 99.76% | 99.996% | 99.541% | 99.577% |
| 0x40D | | 99.76% | 99.996% | 99.546% | 99.581% |
| Standard Deviation | | 0.0726% | 1.3648% | 0.0035% | 0.0035% |

Additionally, there is very little difference between allowing 1 duplicate and 2, which supports my ‘before and after’ theory above. 2 duplicates (3 mailboxes per ID) would have little benefit since both early and late messages are already covered by having 2 per ID.

Following the previous research I have implemented a counter mechanism in the filter replacement strategy, which uses knowledge of the cycle times to replace the CAN ID’s in the filter. This works as follows:

The software is provided with a ‘logging list’ of CAN ID’s and their cycle times (in real life this would be known from the CAN spec by the respective manufacturer.

**typedef** **struct**{

**int** canID;

**unsigned** **long** counter;

**unsigned** **long** loggedCounter;

**int** timer;

**int** timer\_reload;

} logging\_Sequence\_t;

logging\_Sequence\_t loggingSequence[BUFFERSIZE];

**typedef** **struct**{

**int** canID;

**unsigned** **int** cycleTime;

} logging\_list\_t;

logging\_list\_t loggingList[]={

{ 0x187 , 10 },

{ 0x188 , 10 },

{ 0x189 , 10 },

{ 0x18A , 10 },

/\*\* Some IDs omitted for clarity \*\*/

{ 0x38D , 10 },

{ 0x407 , 10 },

{ 0x409 , 10 },

{ 0x40B , 10 },

{ 0x40D , 10 },

{ 0x707 , 50 },

{ 0x709 , 50 },

{ 0x70B , 50 },

{ 0x70D , 50 }

};

During initialisation, the software reads all of the cycle times, and calculates a counter value for each ID based on the cycle time of the ID divided by the minimum cycle time in the list:

**void** **buildSequence**(**void**){

**int** i, cycleTime\_min;

cycleTime\_min = 0xFFFF;

**for**(i=0;i<listSize;i++){

**if**(loggingList[i].cycleTime<cycleTime\_min){

cycleTime\_min = loggingList[i].cycleTime;

}

}

**for**(i=0;i<BUFFERSIZE;i++){

**if**(i<listSize){

loggingSequence[i].canID = loggingList[i].canID;

loggingSequence[i].timer\_reload = loggingList[i].cycleTime/cycleTime\_min;

loggingSequence[i].timer = 1;

}

}

}

The sequence replacement now works in a similar manner to a TT scheduler. In the replacement algorithm the next ID is found that isn’t already included in the filter. The timer for this ID is decremented and if it has reached zero, the ID is used. If not, the next ID is interrogated:

boolean\_t **updateFilter**(**unsigned** **int** filterPointer){

**static** **int** last\_i = -1;

**int** i, j;

flag\_t result = *FALSE*, IDfound = *FALSE*;

i = last\_i;

**do**{

**if**(i<(listSize-1)){

i++;

}

**else**{

i=0;

}

**for**(j = 0; j < FILTERSIZE; j++){

**if**(acceptanceFilter[j].canID == loggingSequence[i].canID){

IDfound = *TRUE*;

}

**else**{

IDfound = *FALSE*;

}

}

**if**(IDfound == *FALSE*){

loggingSequence[i].timer--;

}

**if**(loggingSequence[i].timer<=0){

result = *TRUE*;

}

}**while**((result == *FALSE*)&&(i != last\_i));

**if**(result == *TRUE*){

last\_i = i;

loggingSequence[i].timer = loggingSequence[i].timer\_reload;

acceptanceFilter[filterPointer].canID = loggingSequence[i].canID;

acceptanceFilter[filterPointer].sequencePointer = i;

acceptanceFilter[filterPointer].loggedFlag = *FALSE*;

}

**return** result;

}

This strategy has resulted in much more predictable results, and proves that the unusual behaviour seen before was due, in part at least, to the lower frequency messages blocking the higher frequency ones in the filter. This is contrary to some previous, less formal, testing that I mentioned in my previous report, but I believe the results of these new tests to be much more trustworthy and conclusive.

Figure 1 shows a comparison between the new and old replacement strategies. RD min and RD max show the range of hits that the existing telemetry device was achieving (see previous reports). Most notably, it can be seen that the line ‘with cycle time compensation’ is much smoother than the one without. The hit rate no longer drops between a filter size of 23 and 30.

Figure 1: Hit rate vs filter size for 32-ID logging list

Figure 2 shows that the number of message hits levels off when the filter size is around 50% of the total size of the logging list. If this simulation is accurate, this means that I can successfully use this mechanism to read CAN data with less than 5% loss (worst case for listSize = 16) with a CAN acceptance capacity of half the total number of IDs. This could be particularly useful for devices such as the Texas Instruments microcontrollers, whose native CAN controller is ‘mailbox’ based – mailboxes can be dynamically changed in software to suit the CAN bus, and messages stored in larger buffers in RAM.

Figure 2: Message hits vs filter size for varying list sizes

|  |  |  |  |
| --- | --- | --- | --- |
| List size | Total messages expected | Total hits when  filter size = list size/2 | Percent hits |
| 12 | 541318 | 541019 | 99.94% |
| 16 | 721755 | 689385 | 95.52% |
| 20 | 902192 | 888571 | 98.49% |
| 24 | 1082631 | 1076050 | 99.39% |
| 28 | 1263068 | 1261754 | 99.90% |
| 32 | 1344249 | 1342482 | 99.87% |

# Short-term plan (This month)

* Ongoing reading around caching and related behaviour.
* Port the code to an embedded implementation, taking a logging list from a ‘remote’ application over a serial port.
  + Need to make sure, the hardware can report which IDs it has ‘seen’ without disrupting the timing of the software.

# Long-term plan (To project end)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Task | **June 2013** | **July 2013** | **Aug 2013** | **Sept 2013** | **Oct 2013** | **Nov 2013** | **Dec 2013** |
| Module B1 |  |  |  | Exam: 16th |  |  |  |
| Module B2 |  |  |  | Course: 16th to 20th |  |  | Exam: 9th |
| Literature review / background research |  |  | 31st |  |  |  |  |
| Desktop feasibility Simulation & formal write-up | 30th |  |  |  |  |  |  |
| Hardware Implementation and testing |  | 31st |  |  |  |  |  |
| Thesis draft (deadline) |  |  |  |  | 31st |  |  |
| Thesis Submission (deadline) |  |  |  |  |  |  | 31st |