

Project Progress report: July 2013

Previous work

My most recent Project Proposal (June 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 (F28335) microcontroller, 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
    },
};
```

The task, 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);
            }
        }
    }
}

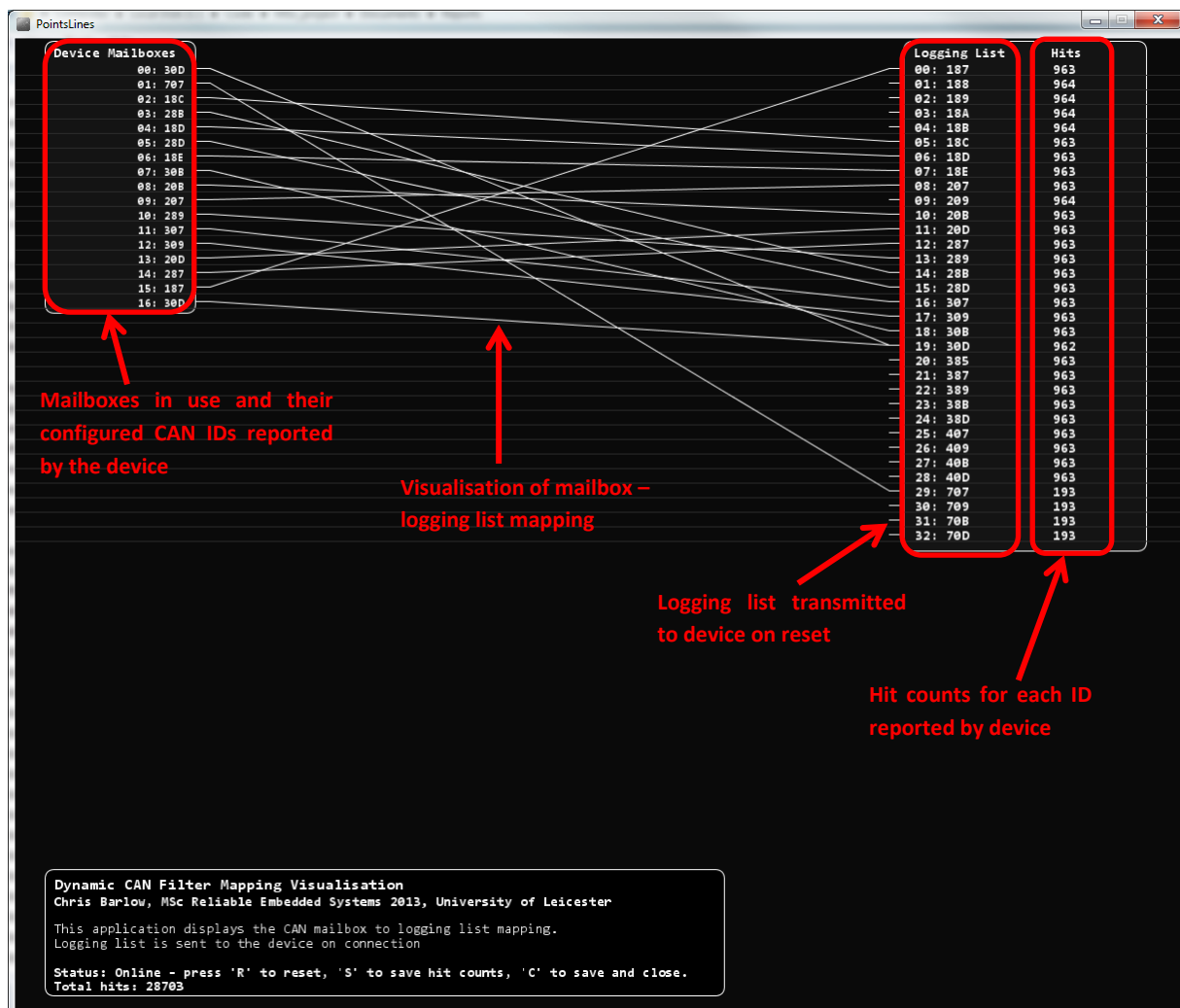
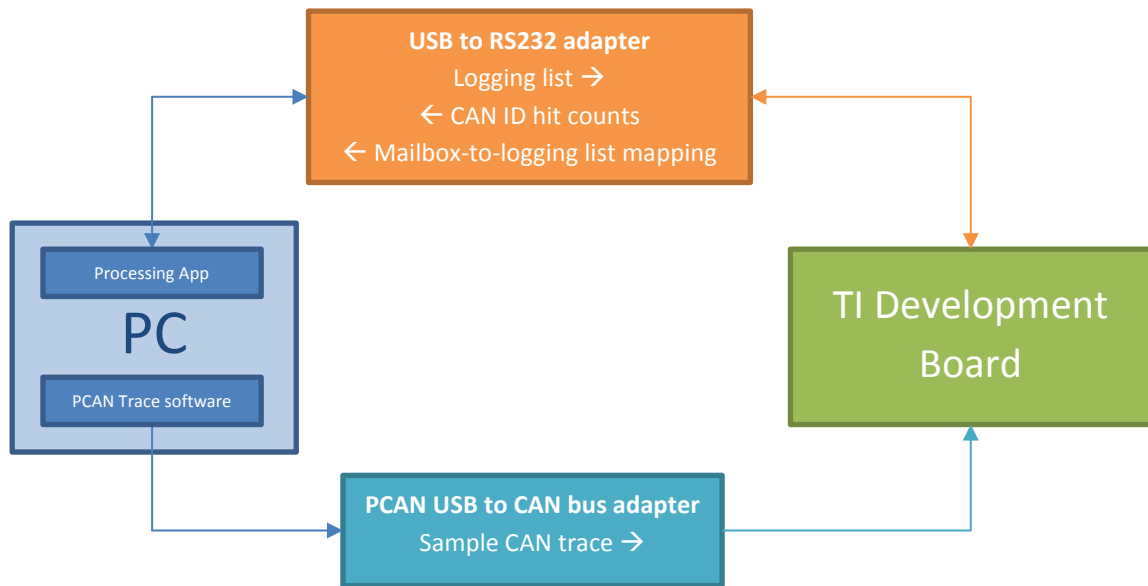
```

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).

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.



```

/* The logging list. This is transmitted to the device for filter configuration */
int[][] loggingList = {
    {0x187,8,20},
    {0x188,8,20},
    {0x189,8,20},
    {0x18A,8,20},
    {0x18B,8,20},
    {0x18C,8,20},
    {0x18D,8,20},
    {0x18E,8,20},
    {0x207,8,20},
    {0x209,8,20},
    {0x20B,8,20},
    {0x20D,8,20},
    {0x287,8,20},
    {0x289,8,20},
    {0x28B,8,20},
    {0x28D,8,20},
    {0x307,8,20},
    {0x309,8,20},
    {0x30B,8,20},
    {0x30D,8,20},
    {0x385,8,20},
    {0x387,8,20}
}

```

Development

The visualisation in the desktop application highlighted a bug in the code that was 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;
        }

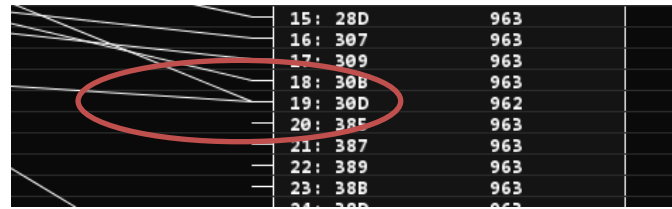
        for(j = 0; j < FILTERSIZE; j++){
            if(acceptanceFilter[j].canID == loggingSequence[i].canID){
                IDfound = TRUE;
            }
            else{
                IDfound = FALSE;
            }
        }

        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.



15:	28D	963
16:	307	963
17:	309	963
18:	30B	963
19:	30D	962
20:	38F	963
21:	387	963
22:	389	963
23:	38B	963
24:	38B	963

What is more interesting is that, when I removed this erroneous `else{}` the effectiveness of the algorithm reduced, giving a much higher variation in 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 more than one mailbox configured to the same ID for the more sporadic messages, 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. By trial and error experimenting with different approaches I have found it most effective to apply a 'soft limit' strategy as follows:

- Duplicate ID's are caused by inconsistency in the message order, or missed messages causing the ID to be inserted late into a mailbox.
 - Each ID is initially allowed only one duplicate in the filter (total of 2 occurrences).
 - When the next message arrives that matches one of these IDs, it is replaced as before, the previous occurrence is left in the filter.
 - The algorithm now allows a further 2 occurrences of the ID.
 - Each time this ID is matched, the matched mailbox is replaced, and 2 additional occurrences are allowed.

This method was found to be more effective than simply only ever allowing 2 occurrences of each ID. One theory is that if a particular ID, or group of ID's is in a particularly inconsistent order, the algorithm will respond to this by allowing more occurrences in the filter. This will mean that priority is given to the ID's with the most missed messages, allowing them to 'catch up' in terms of hit rates. The results below show that this allows for a much more consistent hit rate, albeit a little lower for some ID's.

It is worth noting that this is a slightly risky strategy, since it relies heavily on the cycle time compensation timers being correct. Dictating the wrong cycle time for any of the lower frequency messages causes the duplication algorithm to 'snowball' and fill block all of the mailboxes.

```

#define DUPLICATES_ALLOWED      (1)

void updateFilter(unsigned int filterPointer){
    static int16 sequencePointer = -1;
    int16 last_sequencePointer, last_messagePointer;
    boolean_t result = FALSE;

    if(updateSequenceRequired_G == 1){
        sequencePointer = -1;
    }
    /* Find next required CAN ID in sequence */
    last_sequencePointer = sequencePointer;
    do{
        /* Wrap search */
        if(sequencePointer < (numRxCANMsgs_G-1)){
            sequencePointer++;
        }
        else{
            sequencePointer = 0;
        }

        /* ID not already in mailbox, decrement 'schedule' timer
        (timer set to -1 whilst ID is 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_messagePointer = mailBoxFilters[filterPointer].messagePointer;
        CAN_RxMessages[last_messagePointer].timer = CAN_RxMessages[last_messagePointer].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;
    }
}

```

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% across all of the higher frequency messages.

CAN Message		Hit Rate			
Cycle time	CAN ID	Previous Algorithm	No duplicates	1 duplicate	2 duplicates
~20 ms	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%
~100 ms	0x707	99.42%	99.257%	99.046%	99.157%
	0x709	99.41%	99.346%	99.046%	99.124%
	0x70B	99.37%	99.479%	99.057%	99.124%
	0x70D	99.45%	99.501%	99.046%	99.135%
Standard Deviation		0.0326%	0.1151%	0.0055%	0.0157%

Short-term plan (This month)

- On-going reading around related subjects.
- Update duplication algorithm in the simulation code to match this month's findings in the hardware.
- Freeze code development for simulation, embedded software and desktop app
 - Bugfixes still allowed if necessary.
- Run and document formal test for comparison between simulation, new hardware implementation and existing telemetry device.

Long-term plan (To project end)

Task	July 2013	Aug 2013	Sept 2013	Oct 2013	Nov 2013	Dec 2013
Module B1			Exam: 16 th			
Module B2			Course: 16 th to 20 th			Exam: 9th
Literature review / background research		31 st				
Desktop feasibility Simulation & formal write-up						
Hardware Implementation and testing		31 st				
Thesis draft (deadline)				29 th		
Thesis Submission (deadline)						29 th