# Project Progress report: May - June 2013

## Previous work

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

- Reading around caching behaviour suggested use of counters / timers to more intelligently replace ID's in the filter.

## Work this since previous report

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.

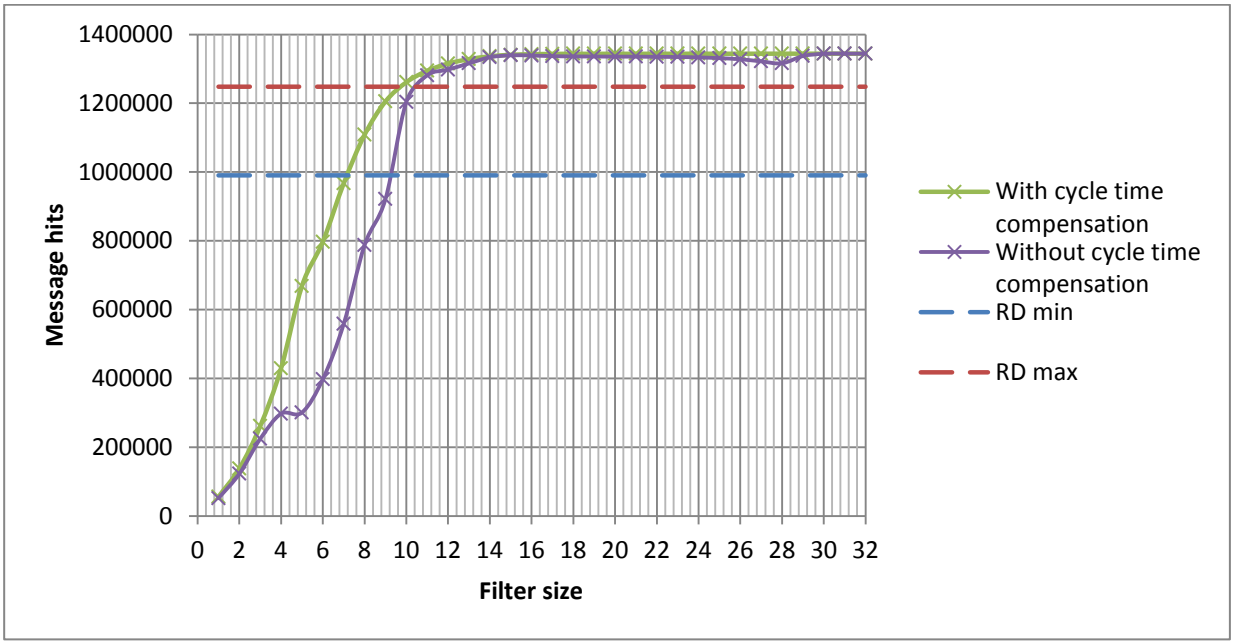


**FIGURE 1: HIT RATE VS FILTER SIZE FOR 32-ID LOGGING LIST**

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.

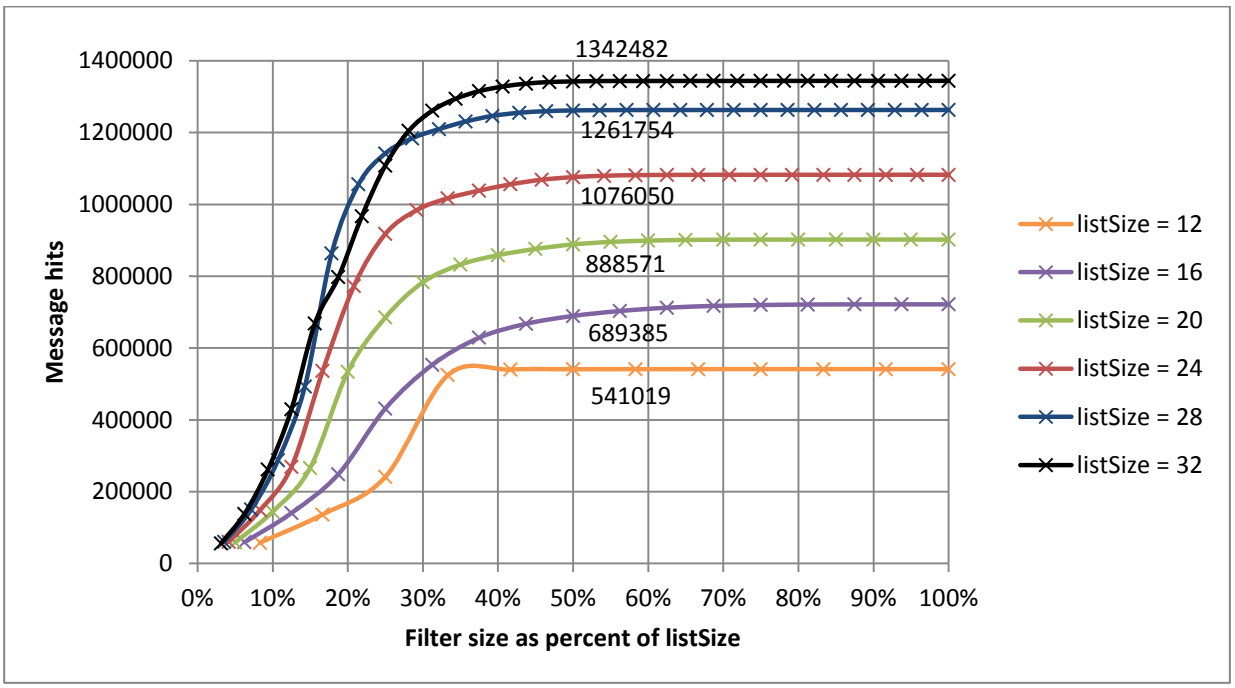


**FIGURE 2: MESSAGE HITS VS FILTER SIZE FOR VARYING LIST SIZES**

| 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: TBD |
| 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 |