A Fixed-Frequency-Trigger Veto for the ATLAS SCT

Alan Barr, Martin Postranecky, Matthew Warren (atlas-tim @ hep.ucl.ac.uk)

HEP Group, Department of Physics and Astronomy, University College London

ATL-INDET-PUB-2007-010
14 August 2007

1. Introduction

The Semi-Conductor Tracker (SCT) forms part of the ATLAS Inner-Detector and, as such, must perform in a strong magnetic field. SCT modules make use of wire-bonds to connect communications electronics to the unit. Triggers initiate increased activity in these parts of the front-end electronics as they transfer data, leading to large variations in the current flowing through the wire-bonds between read-out and idle times. As these currents flow through a magnetic field, the wire-bond experiences a force. At CDF trigger rates close to the mechanical resonant frequency have been seen to cause wire-bonds to oscillate to destruction[i].

Physics triggers are random, with a very low risk of damaging a wire-bond[ii], but calibration and test runs often use fixed trigger rates. To understand this problem in the context of the SCT, studies have been undertaken with test wire-bonds. Figure 1 shows the resonant effects seen in these studies[ii].

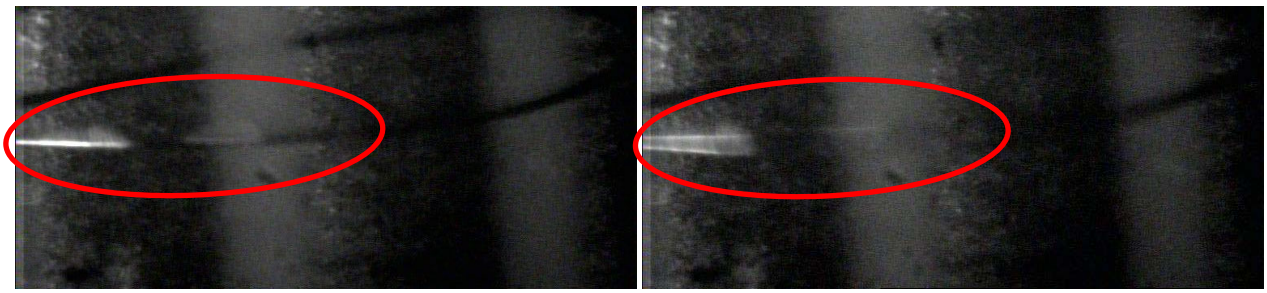

Figure 1. Photographs of a wire-bonds: left) off resonance; right) on resonance (17kHz).

Although the risks for the SCT were found to be low: the wire-bonds are oriented parallel to the magnetic field on the barrel and are short enough on the end-caps not to resonate within the trigger frequency range, protection from fixed-frequency triggers is still thought to benefit the life-time of the detector.

The Fixed Frequency Trigger Veto (FFTV) firmware module has been tasked with identifying these types of triggers, and stopping them. Although Fast Fourier Transforms seem ideally suited to the task, the large amount of logic resources required and the complexity of the system led us toward a simpler approach.

A single TTC Interface Module (TIM) in a Read-Out Driver (ROD) crate distributes triggers to ~12 ROD modules. This corresponds to ~400 front-end modules, all located in the same sub-detector partition. A TIM is also able to generate triggers, including those at fixed-frequencies from a trigger oscillator. This, coupled with a TIM being responsible for sending a busy to the Central Trigger Processor (CTP) that can inhibit triggers, makes it an ideal place to locate the sub-detector specific

FFTV module. However the ROD is also capable of autonomously sending triggers, suggesting that the FFTV module also needs to be located at this level. In the (more probable) case where fixed-frequency triggers are sent via the TIM, this would lead to all RODs engaging their FFTV's at the same time – generating a more confusing situation that if a single TIM was responsible.
The safest solution is to locate the module in both the TIM and ROD. The TIM will have the additional monitoring and logging tasks needed for global trigger operation, with the ROD FFTV providing the same protection, but with less resources consumed by monitoring logic.

2. Fixed-Frequency-Trigger Veto Operation Context

For event synchronisation reasons, arbitrarily ignoring triggers during non-standalone runs cannot be tolerated in ATLAS. Therefore, when TIM is using external TTC triggers (*Run-mode*), if fixed-frequency triggers are detected the only means of stopping them is by asserting the *ROD-Busy* signal and relying on systems upstream to stop sending triggers. To ensure the effects of this autonomous busy decision are understood, clock periods will be counted while TIM asserts *ROD-Busy* for FFTV reasons. An additional counter will log total busy time on TIM for comparison and allowing the veto contribution to dead-time to be monitored. In stand-alone mode TIM can veto triggers prior to generating trigger numbers, keeping the IDs in sync without busy feedback. This configuration is detailed in

Figure 2.

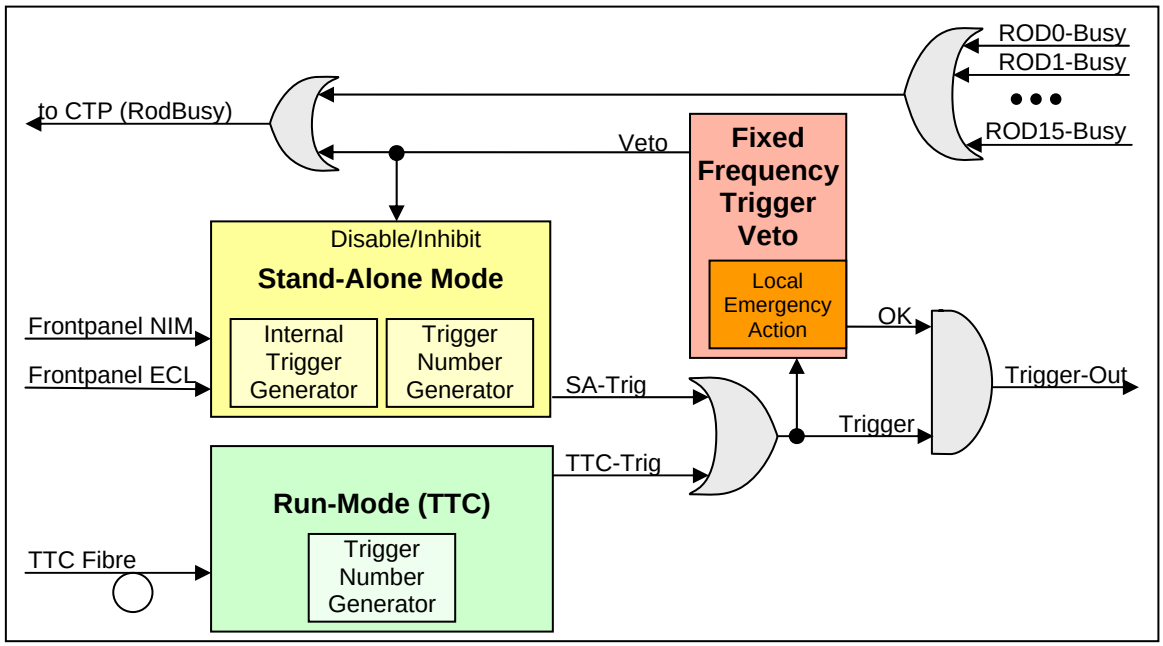

Figure 2. Schematic showing difference between *Standalone-mode* and *Run-mode* (TTC) FFTV operation.

In the case where triggers are received by TIM (or ROD) after the Veto has been asserted, a Local Emergency Action must prevent these triggers reaching the modules. In this case triggers will be discarded, the *ROD-Busy* will be set for an indefinite period, and flags raised to signal the run should be terminated. To allow for the round-trip time from sending the *ROD-Busy* signal to the CTP and triggers stopping, a short delay between Veto and Emergency Action is allowed for (~2µs). To ensure the FFTV is not accidentally disabled, TIM users are required to insert a jumper *and* set a bit in software before trigger vetoing is inhibited.

For the Pixel detector, which uses the same physical RODs and TIMs as the SCT, but is not susceptible to fixed-frequency trigger damage, different firmware on the former and different software/jumper on the latter allow it to be disabled.
It is very likely that early tests at the LHC will operate with only one bunch in the machine with a trigger rate of approximately 11.2kHz. As rates around 17kHz have been seen to be destructive[†], a lower frequency limit of 15kHz has been incorporated into the FFTV algorithm.

3. FFTV Hardware Implementation

The full featured FFTV module, when built in programmable logic on a TIM FPGA, uses 1% of resources on a 600k gate device. A *Veto-Busy-Timer* and an overall *Busy-Timer* are implemented as 48bit counters (81 day rollover period), and a *Veto-ID* (number of veto's asserted) counter is implemented as a 32 bit counter. The ROD version, has smaller counters for an even smaller resource footprint (0, 24 and 8 bit counters respectively).
All parameters, except for the *match-level* (see Section 4, below), are fixed and cannot be modified. The *match-level* can be modified over a range of 2-10, allowing for the FFTV to be characterised at the higher veto rates achieved with lower *match-level* values.

4. The FFTV Algorithm

The algorithm developed (based on the type used at CDF[ii]) is constrained by the physical capabilities of the hardware.
The core of the algorithm quantifies trigger frequency by logging the time between successive triggers. This is counted in units of the system clock (25ns) by the *period-counter*. Successive trigger periods are compared and a counter incremented if they match (within a programmable *period-match-tolerance*). If this *match-counter* exceeds a pre-determined level, the veto signal is generated. In cases where trigger periods do not match, or each time the *period-counter* exceeds a maximum limit (*period-rollover*) and resets to zero, the *match-counter* is decremented.
By assigning a maximum trigger-period, above which the triggers are ignored, low frequency triggers can be passed. Likewise, applying a minimum period causes high-frequency triggers to be ignored. This is particularly useful during double-trigger tests, where the lower frequency components can be identified amongst the otherwise widely varying successive trigger periods.
Although these upper and lower frequency limits are by no means a proper band-pass filter, they are capable of isolating triggers that would be missed by the CDF version of the algorithm, as well as passing useful low frequency periodic triggers.
The operation of the algorithm in an FPGA is best demonstrated by looking at the simulation results detailed in Section 5 below.

[†] This frequency was observed for 3 mm bond wires, orientated so that the oscillations were out of the plane of the loop. In the SCT the bond wires will either be orientated so that oscillations would be in the plane of the loop (barrel) or to have much shorter bond wires. In either case this results in much high resonant frequencies.

5. FFTV Module Behavioural Simulation

The code developed for the FPGA has been simulated using the ModelSim‡ software package. In all cases the system was configured with the values show in Table 1, below. These values were chosen to illustrate the functionality of the algorithm only.

Table 1. Parameter values for the FFTV algorithm in the FPGA.

Quantity	Value	Units	Value in real terms
Period upper limit	2850	Clock cycles (25 ns)	14.035 kHz
Period lower limit	2200	Clock cycles (25 ns)	18.182 kHz
Period roll-over	4100	Clock cycles (25 ns)	9.756 kHz
Period match tolerance (ΔT)	127	Clock cycles (25 ns)	+/- 315 Hz
Period match threshold	5	-	6
Veto duration	8000	Clock cycles (25 ns)	200µs

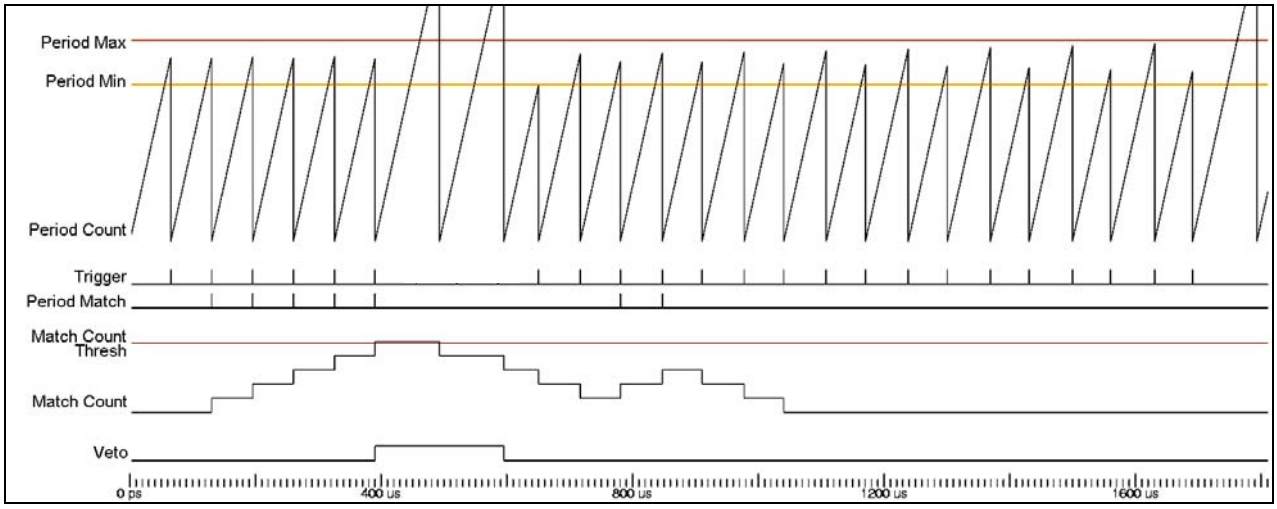

Figure 3. Demonstrating a Veto generated after 5 successive triggers, as well as trigger matching some (slightly) non identical trigger periods. Note that during a Veto the period-counter continues counting beyond the scope of this plot.

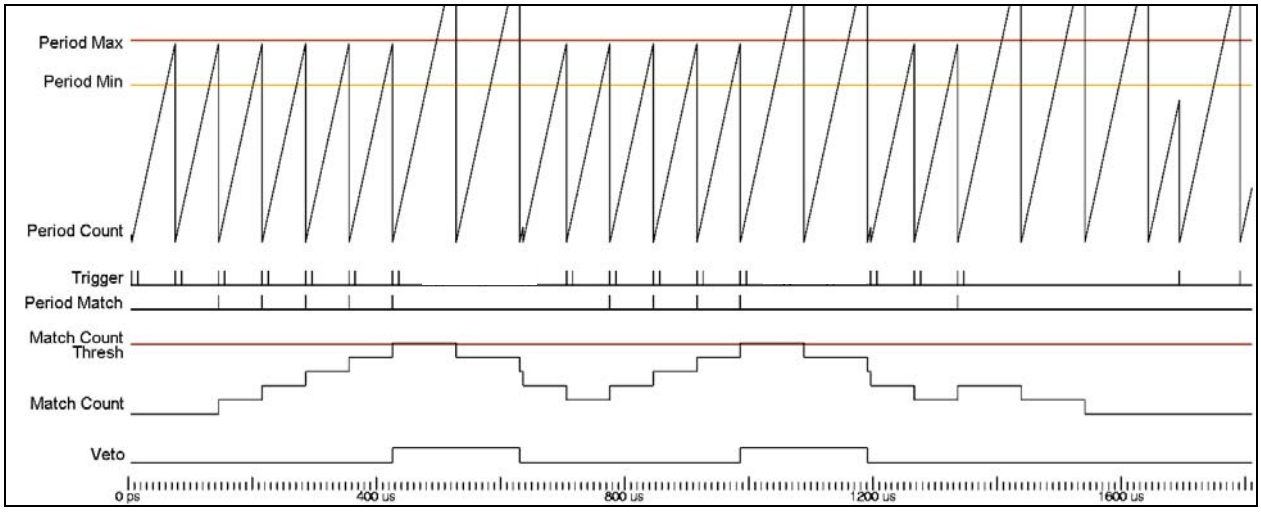

Figure 4. Demonstrating that with short-period double triggers a repetitive trigger is still detected.

‡ Mentor Graphics, http://www.model.com/

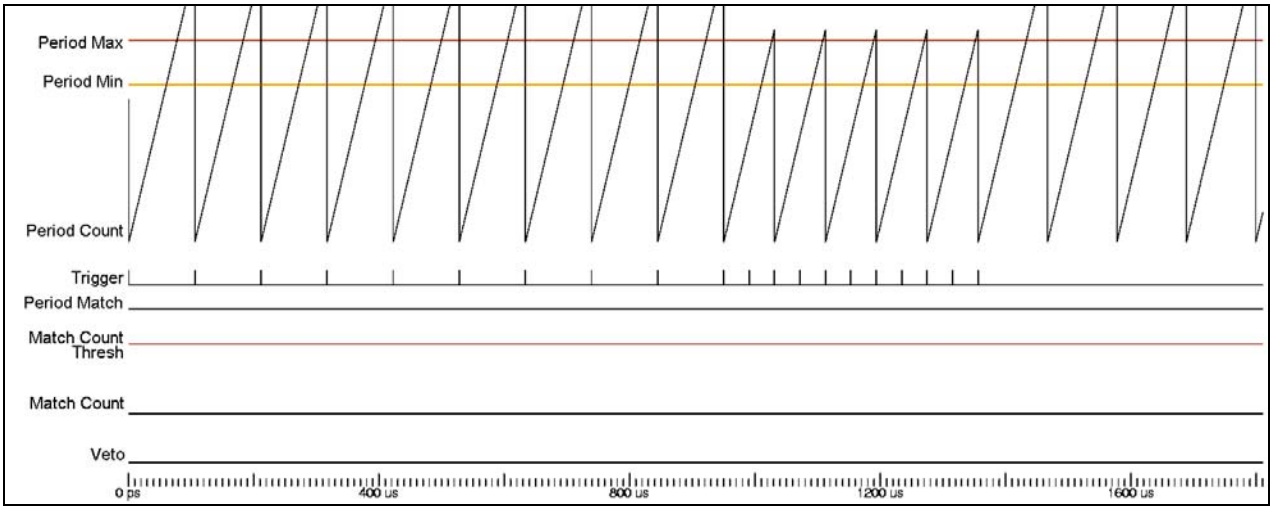

Figure 5. Demonstrating trigger rates below and above threshold being ignored. Note that short periods are ignored and used as part of a longer period component.

6. Oscilloscope Traces

Using test-points on the TIM, aspects of FFTV operation can be displayed on an oscilloscope. To best utilise the display on the oscilloscope, the rate and ranges have been modified by approximately a factor of 10 from that required. These are shown in Table 2.

Table 2. Parameters used for FFTV oscilloscope tests.

Quantity	Value	Units	Value in real terms
Average Frequency	150	kHz	-
Period lower limit	66	Clock cycles (25 ns)	606 kHz
Period upper limit	364	Clock cycles (25 ns)	110 kHz
Period match tolerance (ΔT)	1	Clock cycles (25 ns)	25 ns
Period match threshold	16	-	17
Veto duration	40000	Clock cycles (25 ns)	1 ms

Two tests were undertaken. The first (results shown in Figure 6 left) demonstrates the *match-counter* in operation; triggers are unaffected until the FFTV module is enabled, and then it takes 16+1 successive triggers to assert the veto.

The second test (results shown in Figure 6 right) demonstrates random triggers being ignored, then, after switching to fixed frequency triggers, the veto being asserted.

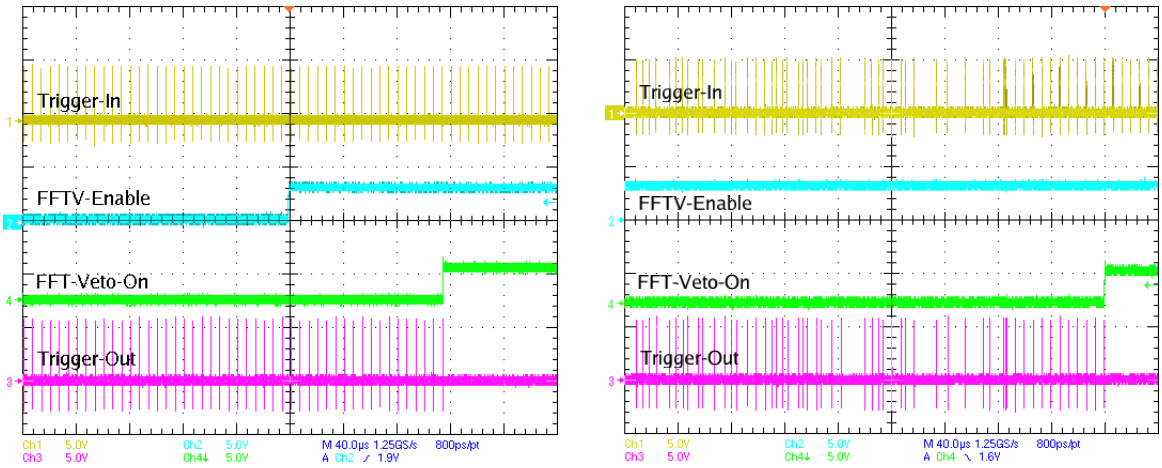

Figure 6. left) Oscilloscope display showing a 150kHz fixed frequency trigger being passed until the FFTV is enabled and then counts 16+1 successive matching periods.
right) Oscilloscope display showing a random trigger averaging 150kHz switching to a fixed frequency and thus generating a veto.

7. Reliability

To understand the long-term effect of the FFTV on detector dead-time, tests have been undertaken using the TIM with a random trigger source. To verify these results a simple Monte Carlo has also been written. Specifically we wanted to show the probability of a trigger generating a Veto.
Initial tests used parameters based on those used at CDF, that have been fine tuned (and now hard-coded in firmware) for SCT needs. These final parameters are detailed in Table 3.

Table 3. Parameters used for Monte Carlo and TIM random veto test.

Quantity	Value	Units	Value in real terms
Average Frequency	75	kHz	-
Period lower limit	80/85	Clock cycles (25 ns)	500kHz/471kHz
Period upper limit	2666	Clock cycles (25 ns)	15 kHz
Period match tolerance (ΔT)	40	Clock cycles (25 ns)	1 μs
Period match threshold	10	-	
Veto duration	40000	Clock cycles (25 ns)	1 ms

To understand the results properly, tests were needed to both verify and calibrate the hardware operation of the algorithm and supporting systems.
Generating randomly spaced triggers is not trivial in firmware. Using a pseudo-random number generator (PRNG) is a good compromise. On TIM the PRNG is constructed using a linear-feedback shift-register (LFSR). The LFSR generates seemingly unrelated numbers in a non-repeating sequence of $(2^{N-1}-1)$ length, where N is the number of bits in the binary number generated. TIM's LFSR is 38 bits long. As each bit now has an effective 50% chance of randomly being a 0 or a 1, a single bit can be used as pseudo-randomly spaced trigger at an average rate of 50% the LSFR shift rate (which, in our case, is the LHC bunch-crossing rate of 40MHz). By demanding more bits to be a known state, the average rate is halved with each additional bit, allowing the 40MHz to be divided down to generate triggers at rates of 80kHz and below. On TIM the randomiser frequency setting is used to control the number of bits demanded in the trigger generating equation (see the *BunchSpacing=1* plot in Figure 7, below).

To further complicate matters, the LHC will often run during commissioning with collisions spaced much further apart than during normal data taking. In particular, certain machine tests require spacings of 22*25ns and 82*25ns. To characterise the FFTV, triggers of this structure are generated by gating the trigger output of the TIM randomiser with a 25ns wide periodic pulse, of programmable duty-cycle. Obviously this reduction in number of possible triggers affects the average trigger rate. To quantify this, frequency scans at various bunch spacing have been performed, see below in Figure 7.

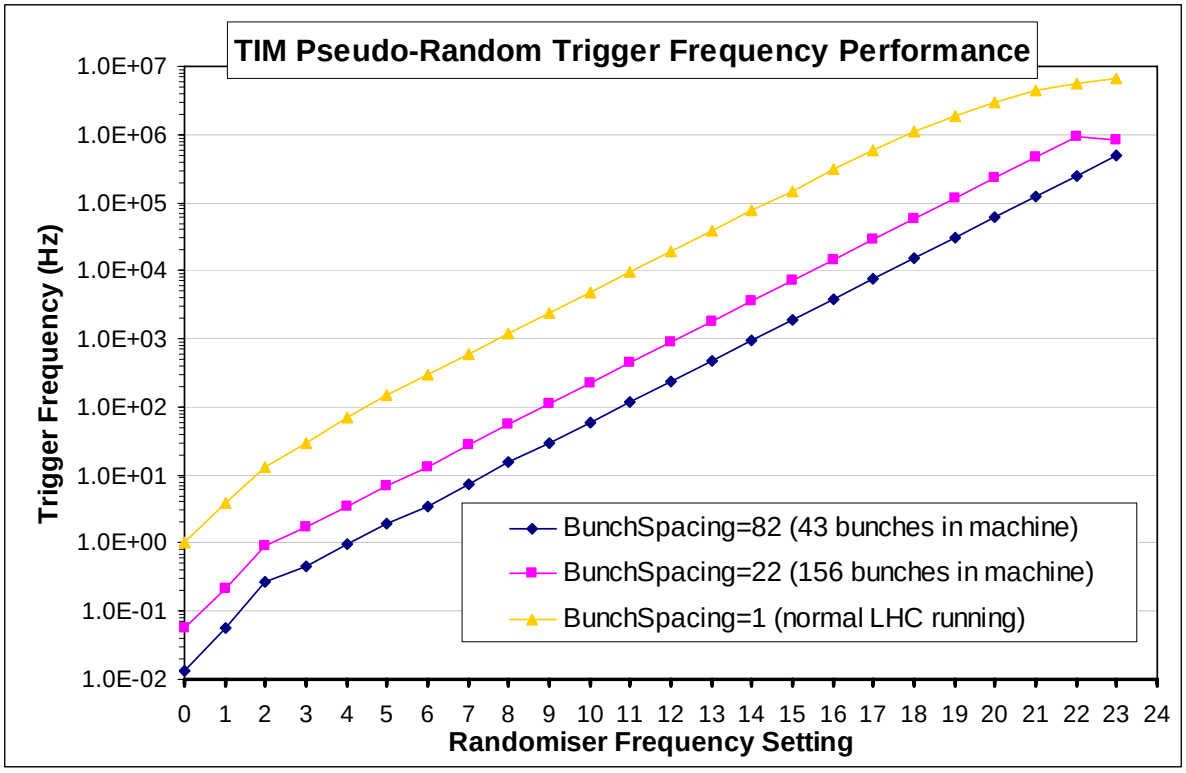

Figure 7. TIM Pseudo-Random Trigger Frequency Performance: By scanning over the randomiser frequency and bunch-spacing settings, average trigger rates are verified.

To evaluate FFTV performance, the TIM (pseudo) random trigger source was used. The TIM was setup to veto its own triggers (i.e. *Standalone-mode*). Triggers and the number of vetoes generated were counted. These numbers are used to calculate the probability of generating a veto. By scanning through the setting for the number of matching triggers required to generate a veto (*match-level*), the relationship between this setting and veto-probability can be understood. This is useful for predicting veto rates at higher match-levels as at these higher levels the veto is only asserted at the order of 1 per day or less, making it difficult to get good statistics.

Initial tests showed that average trigger frequency affected FFTV performance, with Figure 8 demonstrating the effect of frequency settings approximately at, above and below the specified ATLAS average trigger rate of around 75kHz.

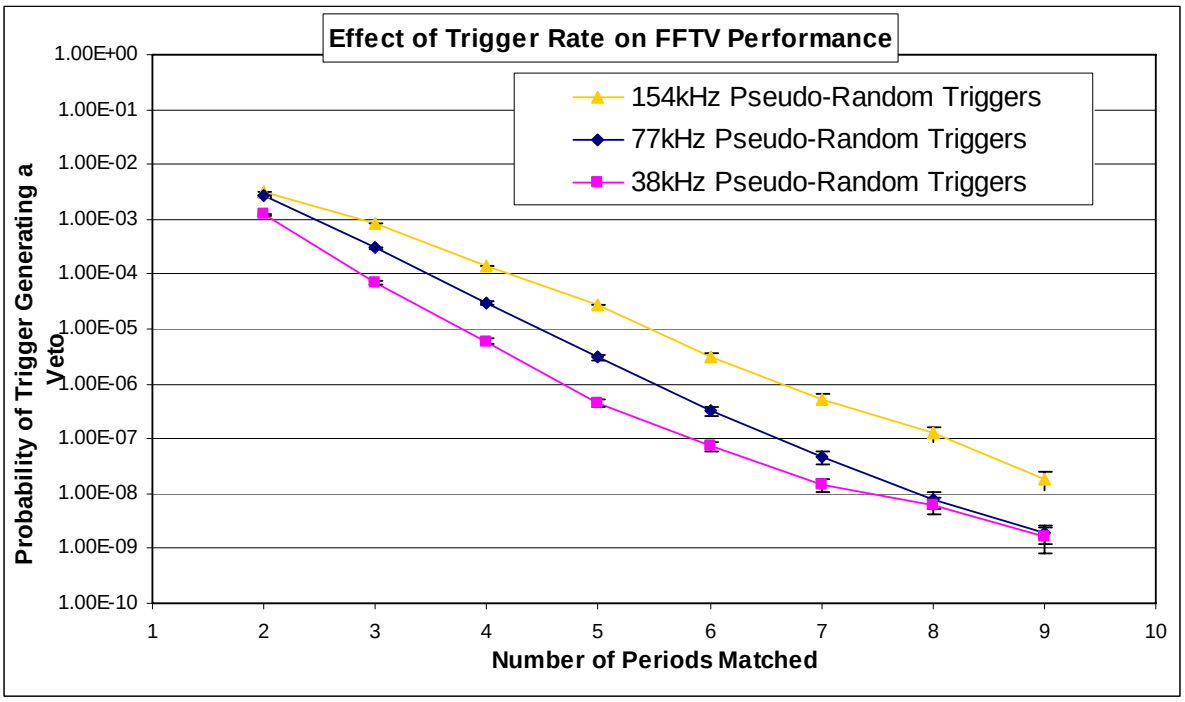

Figure 8. Effect of trigger rate on FFTV performance: Shows the variation in performance due to bunch-spacing and trigger-rate.

As the bunch-spacing setting affects randomiser trigger rate, and trigger-rate affects FFTV performance, FFTV performance at different bunch-spacings is difficult to compare directly. Figure 9 shows FFTV performance at the 3 most common bunch-spacing settings. Note that the average trigger frequency achieved during the test is also shown.

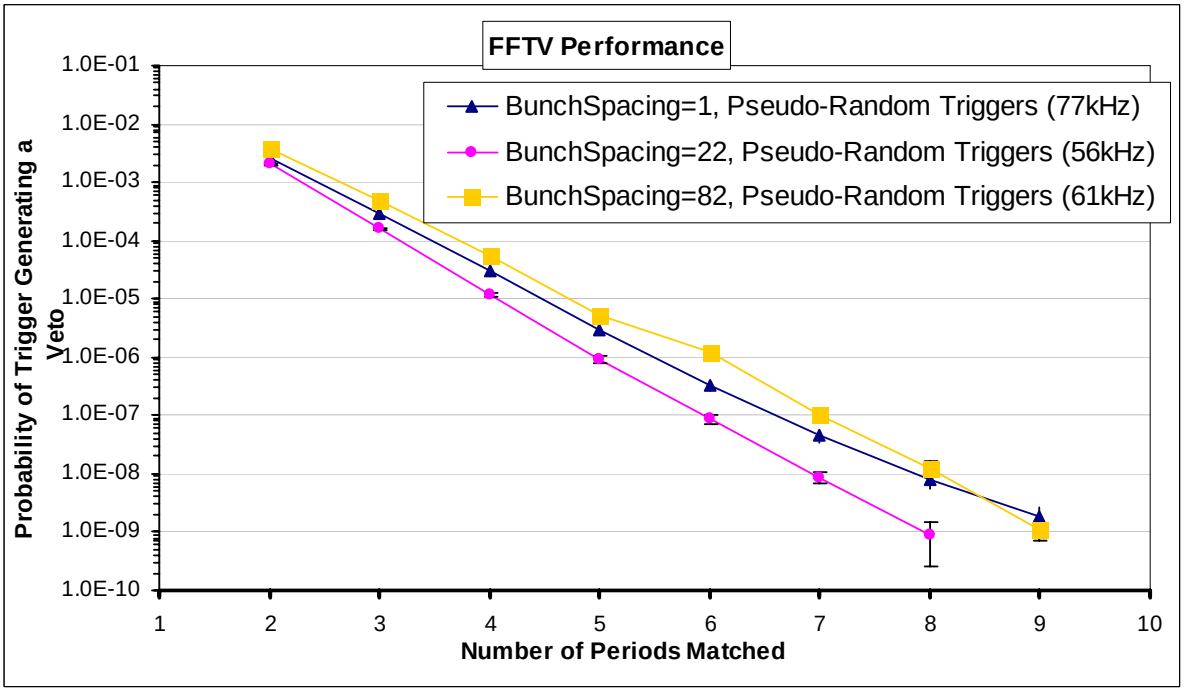

Figure 9. Overall FFTV Performance.

The FFTV algorithm has a lower-period (*period-min*) level below which triggers are ignored. This had been set to 80 clock-periods (i.e. 500kHz) as this is twice the *match-tolerance* and well above

the wire-bond resonant frequency and physics trigger rates. As a bunch spacing of 82 (i.e. 43 bunches in the machine) falls inside this range, we expected a much higher level of vetoing. To safeguard against this the lower-period setting was moved to 85 clocks (471kHz).

Test results comparing the 2 lower-period settings were surprising, though, as the effect of the change was inconclusive (see Figure 10). To ensure that the setting has indeed changed and had some effect, a scan of fixed-frequency trigger frequencies was undertaken. The results, seen in Figure 11, show clearly that the FFTV does affect trigger rates at the *period-min*/*max* boundaries.

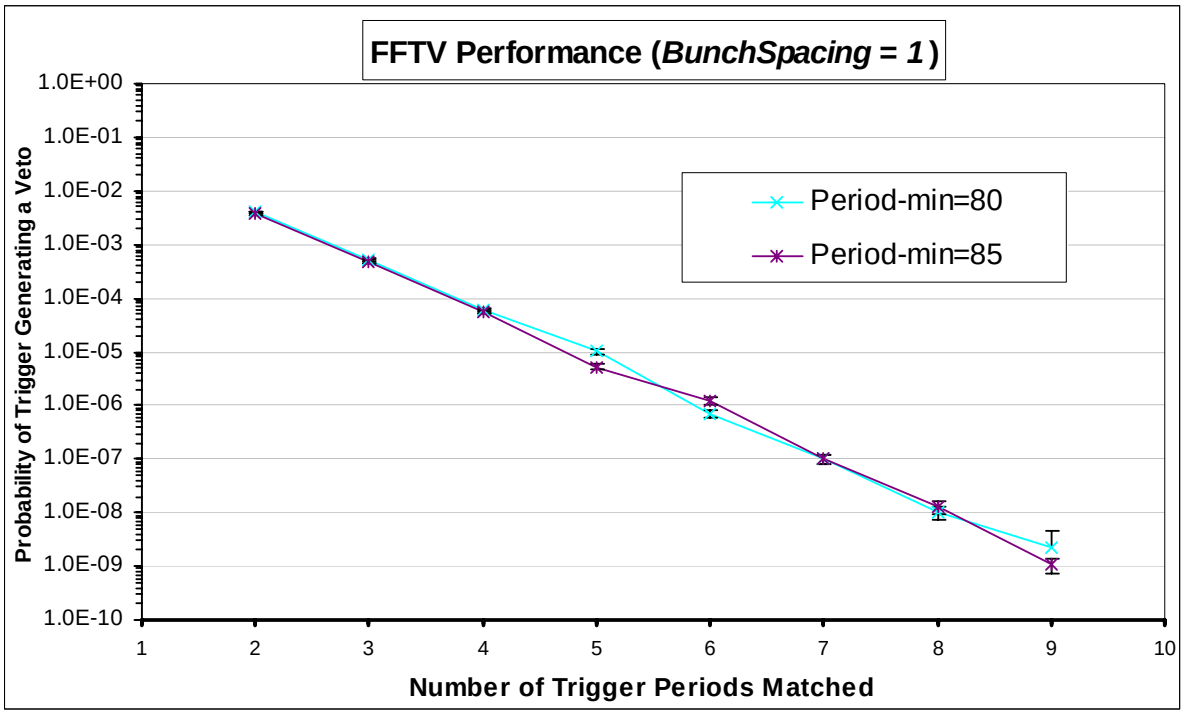

Figure 10. Comparing FFTV performance at different lower-period-threshold values.

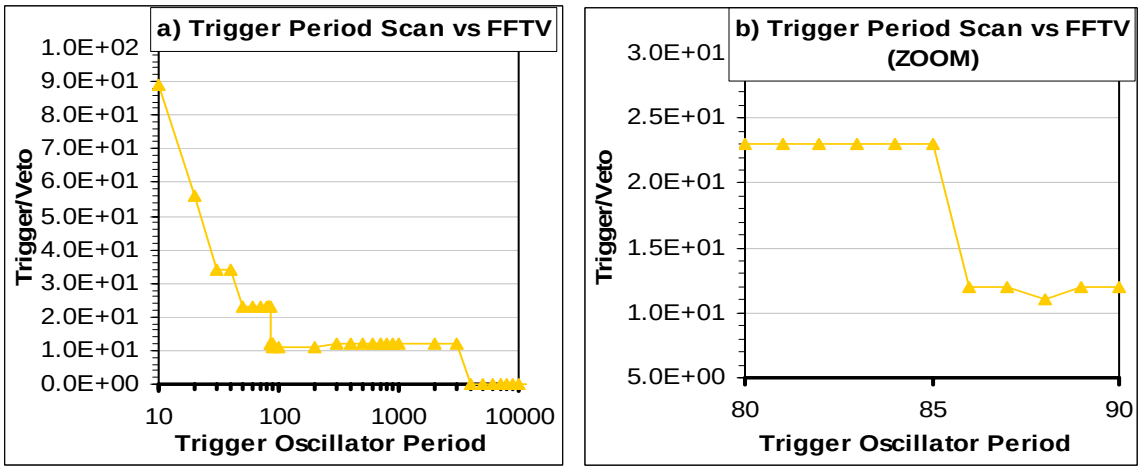

Figure 11. Fixed Frequency Trigger Period Scan. a) Shows the full FFTV operating range, while b) shows a zoom in to around the 82 bunch-spacing mark.

Using a simple Monte Carlo, developed by Tony Weidberg (listed in Appendix 13), the theoretical probability of the FFTV algorithm vetoing random triggers is quantified. We compared the results from the initial tests to those from TIM, and noticed some discrepancies. The only algorithmic

difference between the software and hardware systems was the type of random number generator used. A PRNG was coded for the Monte Carlo to emulate the method used in TIM hardware. Figure 12 shows that the PRNG Monte Carlo agrees nicely with TIM data.

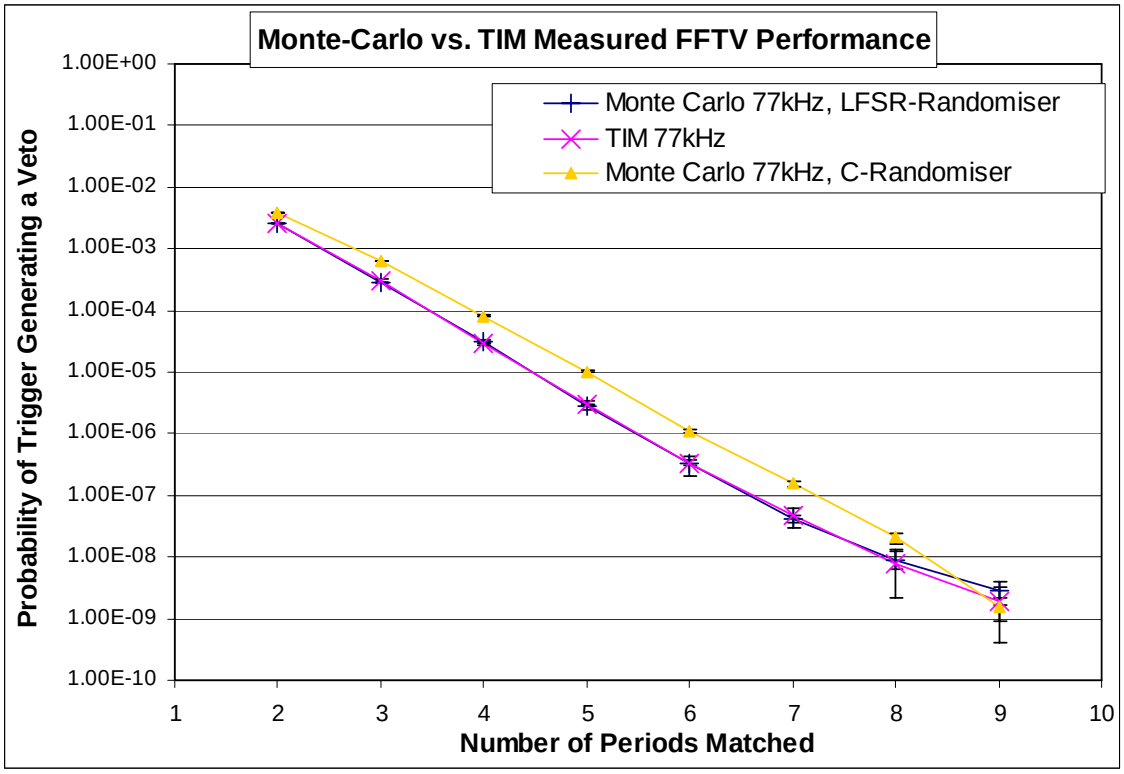

Figure 12. TIM vs. M-C performance.

8. Conclusion

A system for protecting the SCT from fixed frequency triggers has been implemented. It can be introduced into the ATLAS trigger stream with negligible effect on data-taking efficiency.
Performance has been evaluated using both the hardware derived and Monte Carlo generated data. These agree.

9. Acknowledgements

Many people have been invaluable. Tony Weidberg at Oxford for all of his input, code and useful conversation. John Hill (Cambridge) for his guidance and efforts in getting the affected groups (Trigger, Pixel and SCT) to accept the change (not easy!). John Joseph at LBL, ever accommodating, and for making the inclusion of the module into the ROD so easy.

10. References

[i] G. Bolla et al., Wire-bond failures induced by Resonant vibrations in the CDF silicon tracker, Nucl. Instr. Meth. A518 (2004) 277.
[ii] T. J. Barber et al., Resonant Bond Wire Vibrations in the ATLAS SemiConductor Tracker, Nucl. Instr. Meth. A538(2005) 442-457.

Appendix

1. FFTV Implementation Specifics: TIM

- FFTV firmware located in the TIM trigger-out path, i.e. ALL triggers destined for RODs, regardless of source or TIM configuration will be monitored and vetoed if necessary (see *Local Emergency Action* below)
- TIM detects "fixed frequency" triggers at rates:
  - above 15 kHz
  - below 471 kHz (thus allowing periodic double-trigger patterns to be still be detected as fixed frequencies)
- On receiving 10 triggers (presetable in range 2-10 only) of the same frequency (+/-1 MHz), a Veto signal is generated.
- In an attempt to stop triggers, TIM asserts the RodBusy output to the CTP (via front-panel NIM connection) for 1 ms.
  - TIM also disables all non-TTC trigger sources (NIM, ECL and internal).
- The TIM expects the CTP to stop sending TTC triggers from 2μs after asserting the busy.
- The Crate-Controller can read the following status information from TIM:
  - RodBusy output asserted (Busy Status3 Reg, addr. 0x5A, bit 10)
  - Veto Asserted (Busy Status3 Reg. addr. 0x5A , bit 14)
  - 'Latched' bits, for cases where a status bit is only asserted for a short period, bits are stored in the Busy Status3 Latch Reg. (addr. 0x5E)
  - Total time spent with RodBusy (to CTP) is asserted (Overall Busy Count Reg. addr. 0x60,62,64)
  - Time spent with RodBusy output (to CTP) asserted due to Veto. This allows Veto contribution to overall busy to be monitored (FFTV Count Reg. addr. 0x74,76,78).
  - Number of times the Veto signal is asserted (Veto ID Reg. addr. 0x84/86)
  - Number of triggers received (and ignored in stand-alone mode) while the Veto signal is asserted (Veto ID Reg. addr. 0x84/86)
  - RODBusy Status Reg. (addr. 0x20)
  - RODBusy Latch Reg. (addr. 0x22)
  - RODBusy Monitor Register (addr. 0x24)
- Veto system can be disabled by insertion of a jumper AND writing to the relevant bit in a register (allows Pixels to disable the Veto).
- For any case where triggers continue after a Veto is active, imposed *Local Emergency Action* to stop these triggers leaving TIM.

2. FFTV Implementation Specifics: ROD

The ROD will use an identical FFTV module to that used in TIM (same source code) and will be filter all ROD generated triggers. Monitoring data will be is reduced to save resources.

- Reduced fftv-busy-time-counter and fftv-id-counter sizes to 24 and 8 bits respectively, with roll-over status bits just in case.
- Serial trigger command detection algorithm ("0110"). This is needed as the FFTV is located after DSPs that generate serial trigger commands internally.
- Connect *fftv_disable_i* signal line to *I-Am-A-Pixel-ROD* line.
- Configuration bits in ROD Reg-Block:
    - FFTV-Match-Level: 4 bits, but values 2-10 valid only, other values default to 10.
    - Emergency-Clear: 1 bit, '1' clears and emergency condition*
    - Count-Clear: 1 bit, '1' clears Busy-Time-Counter*
    - ID-Clear: 1 bit, '1' clears Veto ID counter*

    * Need to write a zero afterwards to allow triggers again.
- Status bits/words for Register Block:
    - 20 bit Busy-Time-Counter
    - 12 bit ID-Counter
    - 1 bit Busy-Time-Counter-Rollover (latching)
    - 1 bit ID-Counter-Rollover (latching)
    - 1 bit Busy Active
    - 1 bit Busy Active (latching)
    - 1 bit Emergency Active
    - 1 bit Emergency Active (latching)?

3. Local Emergency Action

Describing the case where triggers continue to arrive at TIM after the Veto has been asserted:

- The FFTV system determines that a fixed frequency is present and generates a Veto (lasting 1ms).
- Triggers are allowed to pass without action for ~2µs (this number equates to 471kHz, which is the upper limit of the FFTV algorithm detection window.)
- After 2µs ANY trigger destined for the RODs (regardless of source) will be stopped and TIM will initiate Local Emergency Action, stopping triggers INDEFINITELY. The RodBusy signal to the CTP will remain asserted
- This condition is flagged by bit ‘LEA-active’ (Busy Status3 Reg. addr. 0x5A, bit 13).
- TIM will de-assert the ‘TIM-OK’ backplane signal (which is sent to all BOCs & RODs in the crate, and indicates that the TTC clock is selected and 'good' and that TIM is ready for TTC triggers.)
- To clear Local Emergency Action the Run Control needs issue a Clear-LEA command to TIM (write to Control Reg. addr. 0x46, bit 0). This command will be ignored for the duration of the 1ms Veto active period, though.

4. Monte Carlo source code

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\
* Toy MC to study estimate CDF algorithm of vetoing fixed frequency triggers  *
* v1: generate triggers according to exponential distribution                 *
* v2: use module 25 ns for trigger times                                      *
*     implement Matt Warren's algorithm                                       *
* v3: reworked by Matt to verify both algo and stats                          *
* v4: Option to use LSFR randomiser added by Matt                             *
*                                                                             *
* This version: 9 September 2006                                              *
\* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

#define DEBUGLVL            0          /* control amount of debug. Values 0,1,2 best. */
#define BINS_FACTOR         200000     /* >200k exceeds int at some point */
#define USE_LFSR            0
#define TRIG_RATE           77850.0    /* = TIM rate + triggers lost to deadtime */
#define LFSR_TRIG_RATE      14         /* integer, as used in TIM code/register list = 77k */
#define MATCH_LEVEL_MAX     3
#define MATCH_LEVEL_MIN     2
#define MATCH_TOLERANCE     1.000e-06  /* 1.0  us          */
#define LOWER_TIME_CUT      2.125e-06  /* 2.125us = 470kHz = 85 */
#define UPPER_TIME_CUT      66.66e-06  /* 66.7 us ~  15kHz  */
#define PERIOD_ROLLOVER     75.66e-06  /* 75.7 us ~  13kHz  period counter roll over level*/
#define VETO_DURATION       1.000e-03  /*  1.0 ms */
#define VETO_LATENCY       75.000e-09  /* 75.0 ns */
#define TRIG_DEADTIME_CLKS  5
#define TRIG_DEADTIME     125.000e-09  /* 125ns */

double gen_trigger   (const int rand_type);
int    rand_lfsr     (const int freq_id);
void   lfsr_step     (int steps);
void   print_int2bin (const unsigned int val);
float  time_min      (double *x, int n, int m);
float  time_max      (double *x, int n, int m);
double rand_exp      (float x);
/* globals */
unsigned int lfsr_high = 0;
unsigned int lfsr_low  = 0;
int exit_loop = 0;

/* ========================================================================== */
void sig_handler (int sig) {
   exit_loop = 1;
   printf ("\nExiting loop ASAP\n") ;
}
/* ========================================================================== */
int main(void) {
   unsigned int i;
   unsigned int match_level, triggers_in_max;
   unsigned int veto_id  [(MATCH_LEVEL_MAX+1)],
                used_triggers[(MATCH_LEVEL_MAX+1)],
                vetoed_trigs [(MATCH_LEVEL_MAX+1)],
                triggers_in  [(MATCH_LEVEL_MAX+1)];
   int    match, veto_on, good_trig, upper_time_cut_clks, lower_time_cut_clks, time_int;
   double error_prob,delta_deltat;
   double prob_veto, current_delta_t, prev_delta_t, delta_t;
   double run_time_tot[(MATCH_LEVEL_MAX+1)];
   char  filenm[50];
   FILE *ofp, *file_dest;

   /* ------------------------------------------------------------------- */
   printf("Starting fftv_mc\n");

   srand(time(0));
   lfsr_high = (int)(rand()*0xffffffff);
   lfsr_low  = (int)(rand()*0xffffffff);
   upper_time_cut_clks = (int)(UPPER_TIME_CUT/0.000000025);
   lower_time_cut_clks = (int)(LOWER_TIME_CUT/0.000000025);

   /* initialize counters */
   for(i=0;i<=MATCH_LEVEL_MAX;++i) {
      veto_id[i]=0;
```

```
   used_triggers[i]=0;
   vetoed_trigs[i]=0;
}

/* Trap the next ^C in sig_handler to interrupt the loop without an error */
if (signal (SIGINT, sig_handler) == SIG_ERR)
   printf ("OS signal error (trying to catch ^C)\n") ;
exit_loop = 0;

for (match_level=MATCH_LEVEL_MIN; match_level<=MATCH_LEVEL_MAX; ++match_level) {
   printf("\n");
   triggers_in_max = BINS_FACTOR*(pow(3,match_level));
   run_time_tot[match_level] = 0;
   match=0;
   veto_on=0;
   current_delta_t = 0;
   prev_delta_t    = 0;
   i = 0;

   while ( (exit_loop==0)&&(i<triggers_in_max) ) {
      i++;
      /* ensure triggers with period<deadtime are not used but add to overall time) */
      good_trig = 0;
      delta_t = 0;

      while (good_trig==0) {
         delta_t += gen_trigger(USE_LFSR);
         if (delta_t>TRIG_DEADTIME) good_trig=1;
      }

      run_time_tot[match_level] += delta_t;
      current_delta_t += delta_t;
      if ( (i%10000==0)||(i==triggers_in_max) ) {
         printf("N=%d/%d, ",      match_level, MATCH_LEVEL_MAX);
         printf("%4us, ",         (int)run_time_tot[match_level]);
         printf("trig %2u/%uM, ", i/1000000, triggers_in_max/1000000);
         printf("veto %9d, ",     veto_id[match_level]);
         printf("prob %2.2e, ",   ((double)veto_id[match_level])/((double)i) );
         printf("rate %2.2ek\r",  ((double)i)/((double)run_time_tot[match_level])/1000 );
         if (DEBUGLVL>2) printf("\n");
      }

      /* -------- Veto ON action ------------- */
      if(veto_on==1) {
         if(current_delta_t > (VETO_DURATION+VETO_LATENCY) ) veto_on=0;
         else vetoed_trigs[match_level] += 1;
      }

      /* -------- Veto OFF action ------------- */
      if (veto_on==0) {
         used_triggers[match_level] += 1;
         if(current_delta_t>=LOWER_TIME_CUT) {    /* period not too short */
            if(current_delta_t>=UPPER_TIME_CUT) { /* period too long */
               match -= (current_delta_t/PERIOD_ROLLOVER);
               if (match<0) match = 0;
               current_delta_t = 0;
            }
            else {                                /* period just right! */
               delta_deltat = fabs(current_delta_t - prev_delta_t);
               if (delta_deltat<=MATCH_TOLERANCE) ++match; else --match;
               if (match<0) match = 0;
               if (DEBUGLVL>2||((DEBUGLVL>1)&&(match_level==2)))
                  printf("ddt %.9lf, dt1 %.9lf dt0 %.9f, tol %.9lf, match %d\n",
                         delta_deltat, current_delta_t, prev_delta_t,
                         MATCH_TOLERANCE, match);
               prev_delta_t = current_delta_t;
               current_delta_t = 0;
            }
         }
         /* check for a veto           */
         /* -------------------------- */
         if(match>=match_level) {
            ++veto_id[match_level];
            veto_on = 1;
         }
      }
   }
   if (signal (SIGINT, SIG_DFL) == SIG_ERR)  /* Treat the next Control-C in default way */
      printf ("Os signal error (trying to un-catch ^C)\n") ;
   triggers_in[match_level] = i;
```

```
      printf("\n");
   }
   printf("\n");
   time_int =  (int)time(0)/60;
   sprintf(filenm, "fftv_mc_summary_%d.txt", time_int);
   ofp = fopen(filenm,"w");

   for (i=0; i<=1; i++) {
      if (i==0) file_dest=ofp; else file_dest=stdout;
      /* print final stats */
      fprintf(file_dest,"\nfftv_mc Results %d\n", time_int);
      fprintf(file_dest,
        "================================================================================\n");
      fprintf(file_dest,"Environ: Delta-t %.3e, Tveto %.3e\n",
                        MATCH_TOLERANCE, VETO_DURATION);
      fprintf(file_dest,"         Upper-time-cut %.9lf %3.0fkHz %4dclks\n",
              UPPER_TIME_CUT, (1/UPPER_TIME_CUT/1000), upper_time_cut_clks);
      fprintf(file_dest,"         Lower-time-cut %.9lf %3.0fkHz %4dclks\n",
              LOWER_TIME_CUT, (1/LOWER_TIME_CUT/1000), lower_time_cut_clks);
      fprintf(file_dest,"         Max-Records %u\n",triggers_in_max);
      if (USE_LFSR==1)
         fprintf(file_dest,"Using LFSR randomiser, Freq ID %d\n", LFSR_TRIG_RATE);
      fprintf(file_dest,
         "--------------------------------------------------------------------------------\n");
      fprintf(file_dest,
              "N Time   TrigsOUT   Vetos   VetodTrg  TrigsIN    Prob     Err    Rate \n");
      fprintf(file_dest,
         "- ----- ---------- -------- -------- ---------- ------- -------- -----\n");
      for(match_level=MATCH_LEVEL_MIN;match_level<=MATCH_LEVEL_MAX;++match_level) {
         prob_veto  = ((double)veto_id[match_level])/((double)triggers_in[match_level]);
         error_prob = prob_veto/sqrt(veto_id[match_level]);
         fprintf(file_dest,  "%x ", match_level);
         fprintf(file_dest, "%5u ", (int)run_time_tot[match_level]);
         fprintf(file_dest,"%10u ", used_triggers[match_level]);
         fprintf(file_dest, "%8u ", veto_id[match_level]);
         fprintf(file_dest, "%8u ", vetoed_trigs[match_level]);
         fprintf(file_dest,"%10u ", triggers_in[match_level]);
         fprintf(file_dest,"%.1e ", prob_veto);
         fprintf(file_dest,"%.1e ", error_prob);
         fprintf(file_dest,  "%4u",
           (int)((double)triggers_in[match_level]/run_time_tot[match_level]) );
         fprintf(file_dest,"\n");
      }
      fprintf(file_dest,
        "--------------------------------------------------------------------------------\n");
   }
   fclose(ofp);
   return 0;
}
/* ============================================================================ *\
\* ============================================================================ */
double gen_trigger(const int rand_type) {
   unsigned int trig_int;
   double clk_to_sec  = 0.000000025;
   double sec_to_clk  = 40000000;
   double ti, trig_double;

   if (rand_type == 0) {
      ti=rand_exp(TRIG_RATE);
      trig_int= (int)(ti*sec_to_clk);
   }
   else trig_int = rand_lfsr(LFSR_TRIG_RATE);
   trig_double = ((double)trig_int)*clk_to_sec;
   if (DEBUGLVL>1) printf("ti %.9lf, t_double %.9lf\n", ti, trig_double);
   return trig_double;
}
/* ---------------------------------------------------------------------------- */
/* ---------------------------------------------------------------------------- */
double rand_exp(float x) {
/* exponential random number generator */
   double xi,xj;


   xi= (double) rand() / (double) RAND_MAX;
   xj=log(xi);
   return (-xj / x);
}
/* ---------------------------------------------------------------------------- */
/* ---------------------------------------------------------------------------- */
int rand_lfsr(const int freq_id) {
```

```
   int value = 0, done = 0;
   unsigned int ll, lh;
   lh = lfsr_high;
   ll = lfsr_low;

   while (done == 0) {
      value ++;
      lfsr_step(1);
      switch (freq_id) {
         case 31:if(((lh&0x00)==0x00)&&((ll&0x00000000)==0x00000000))done=1;break;
         case 30:if(((lh&0x00)==0x00)&&((ll&0x00000000)==0x00000000))done=1;break;
         case 29:if(((lh&0x00)==0x00)&&((ll&0x00000000)==0x00000000))done=1;break;
         case 28:if(((lh&0x00)==0x00)&&((ll&0x00000000)==0x00000000))done=1;break;
         case 27:if(((lh&0x00)==0x00)&&((ll&0x00000000)==0x00000000))done=1;break;
         case 26:if(((lh&0x00)==0x00)&&((ll&0x00000000)==0x00000000))done=1;break;
         case 25:if(((lh&0x00)==0x00)&&((ll&0x00000000)==0x00000000))done=1;break;
         case 24:if(((lh&0x00)==0x00)&&((ll&0x00000000)==0x00000000))done=1;break;
         case 23:if(((lh&0x00)==0x00)&&((ll&0x00000000)==0x00000000))done=1;break;/* 40M*/
         case 22:if(((lh&0x00)==0x00)&&((ll&0x00080000)==0x00080000))done=1;break;/* 20M*/
         case 21:if(((lh&0x00)==0x00)&&((ll&0x10000200)==0x10000200))done=1;break;/* 10M*/
         case 20:if(((lh&0x01)==0x01)&&((ll&0x00100040)==0x00100040))done=1;break;/*  5M*/
         case 19:if(((lh&0x20)==0x20)&&((ll&0x04004004)==0x04004004))done=1;break;/*2.5M*/
         case 18:if(((lh&0x04)==0x04)&&((ll&0x08081010)==0x08081010))done=1;break;/*1.2M*/
         case 17:if(((lh&0x10)==0x10)&&((ll&0x40810202)==0x40810202))done=1;break;/*600k*/
         case 16:if(((lh&0x41)==0x41)&&((ll&0x04104104)==0x04104104))done=1;break;/*300k*/
         case 15:if(((lh&0x41)==0x41)&&((ll&0x04124104)==0x04124104))done=1;break;/*150k*/
         case 14:if(((lh&0x41)==0x41)&&((ll&0x04124105)==0x04124105))done=1;break;/* 80k*/
         case 13:if(((lh&0x41)==0x41)&&((ll&0x04124145)==0x04124145))done=1;break;/* 40k*/
         case 12:if(((lh&0x41)==0x41)&&((ll&0x04125145)==0x04125145))done=1;break;/* 20k*/
         case 11:if(((lh&0x41)==0x41)&&((ll&0x04165145)==0x04165145))done=1;break;/* 10k*/
         case 10:if(((lh&0x41)==0x41)&&((ll&0x05165145)==0x05165145))done=1;break;/*  5k*/
         case  9:if(((lh&0x41)==0x41)&&((ll&0x45165145)==0x45165145))done=1;break;/*2500*/
         case  8:if(((lh&0x51)==0x51)&&((ll&0x45165145)==0x45165145))done=1;break;/*1200*/
         case  7:if(((lh&0x51)==0x51)&&((ll&0x45165155)==0x45165155))done=1;break;/* 600*/
         case  6:if(((lh&0x51)==0x51)&&((ll&0x45165555)==0x45165555))done=1;break;/* 300*/
         case  5:if(((lh&0x51)==0x51)&&((ll&0x45175555)==0x45175555))done=1;break;/* 150*/
         case  4:if(((lh&0x51)==0x51)&&((ll&0x45575555)==0x45575555))done=1;break;/*  75*/
         case  3:if(((lh&0x51)==0x51)&&((ll&0x55575555)==0x55575555))done=1;break;/*  40*/
               /*if(((lh&0x55)==0x55)&&((ll&0x55575555)==0x55575555))done=1;break;    20*/
         case  2:if(((lh&0x55)==0x55)&&((ll&0x55575555)==0x55575555))done=1;break;/*   0*/
         case  1:if(((lh&0x55)==0x55)&&((ll&0x75577555)==0x75577555))done=1;break;/*   5*/
               /*if(((lh&0x55)==0x55)&&((ll&0x77577555)==0x77577555))done=1;break;   2.5*/
         case  0:if(((lh&0x5D)==0x5D)&&((ll&0x77577555)==0x77577555))done=1;break;/* 1.2*/
      }
   }
   return value;
}
/* ---------------------------------------------------------------------------- */
void lfsr_step (const int steps) {
   /* LFSR random number generator */
   /* from the VHDL: shiftreg(39 downto 2) <= shiftreg(38 downto 1);
                     shiftreg(1)           <= not (shiftreg(39) xor shiftreg(35)); */
   unsigned int i, bit_carry, bit31, bit34, bit_feedback;

   for (i=0; i<steps; i++) {
      bit_carry    = ((lfsr_high >>  6) & 0x1);
      bit34        = ((lfsr_high >>  2) & 0x1);
      bit_feedback = ((~(bit_carry ^ bit34)) & 0x1); /* ^ == xor */
      bit31        = ((lfsr_low  >> 31) & 0x1);
      lfsr_high    = ( (lfsr_high <<  1)  | bit31        );
      lfsr_low     = ( (lfsr_low  <<  1)  | bit_feedback );
   }
}
/* ---------------------------------------------------------------------------- */
void print_int2bin(const unsigned int val) {
   int i;

   for (i=31; i>=0; i--) {
      if (((val >> i) & 0x1) == 1) printf("1"); else printf("0");
   }
}
/* ============================================================================ *\
\* ============================================================================ */
```