Blend Glitches - Sources and Solutions

Brian Lantz, Hugo Paris, and the Seismic Team
T1400284-v2, April 17, 2014

1 Summary

We describe and model the sources of the glitches affecting the blend switching and describe two things to help fix the issue. The glitches are caused by the transition between the different filter modules in the Blend Switch algorithm. The Blend Switch is described in detain in T1200126, the "Blend Switching User Guide" by Ruslan Kurdyumov and Chris Kucharczyk. The ramp which transitions from one blend to the next is a simple linear ramp, and the kinks at the start and end are introducing high frequency features. Those features are quite apparent because the T240 signals in the paths are large, and since we are using level 3 control, the controllers have high gain at high frequency. The product of (large input signal) * (kink) * (large high frequency drive) can cause glitches in the control output large enough to saturate the DAC and cause a watchdog trip.

The solution is two-fold. First, replace the linear ramp with the much smoother fifth-order polynomial ramp described in T1300510. This greatly reduces the generation of high-frequency components in the time series. Second, change the turn-on procedure so that we do not restore the tip and tilt targets for stage 1 of the BSC-ISI (the location of the offendingT240s). This will greatly reduce the low frequency amplitude of the T240 signals during the turn on process. We may also adjust the start times of the blend switcher to offset the effects of the various degrees of freedom. In figure 1 we show the example glitch used in this document. It is from a set of glitches described in the Hanford aLOG entry 10539 by Jeff Kissel. It is important to note that the largest glitches are in the horizontal direction.

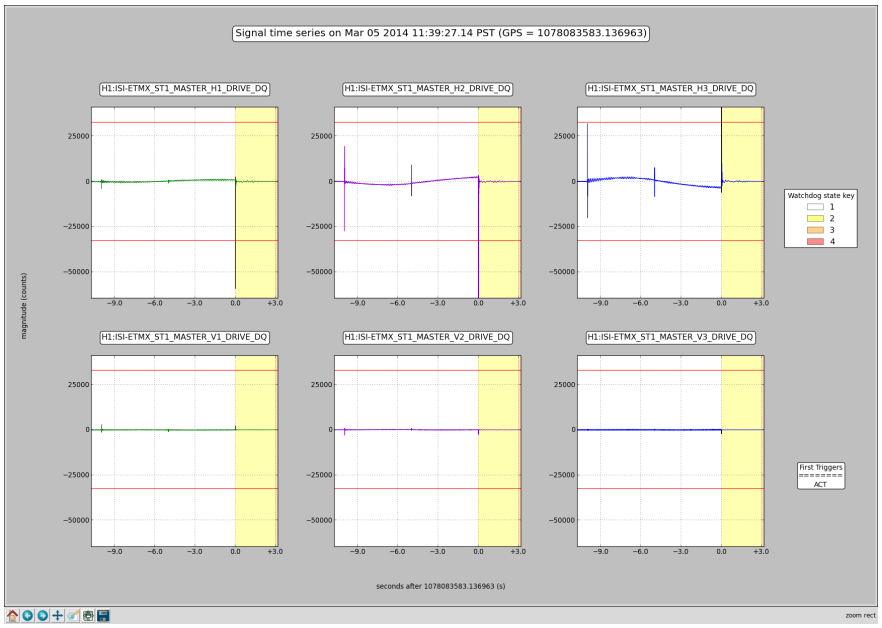

Figure 1: Example of blend-induced glitches, Hanford aLOG 10539

2 Problem Discussion

The relevant feature of the Blend Switcher is that there are 2 parallel paths, the Current (CUR) filter which is running at the start of the Blend Switch process, and the Next Filter (NXT) which has been select. When a switch is requested, the new blend filter in loaded into the NXT bank, the NXT bank is allow to settle for 5 seconds, and then a 5 second linear ramp is applied to move from the CUR to the NXT. The linear ramp has a sharp corner at the beginning and end, so there if there is a large difference between the outputs of the CUR and NXT filters, then a short, high-frequency signal will be imposed on the control signal. If there is significant gain in the controller at high frequency (which is especially true for the level 3 controllers, then a spike will be generated at the control output. This spike is the cause of the glitching of the Blend Filters. This has become apparent as:

1. The T240s are not used in the servo in the 'Blend Start'. As the system comes on, there is tilt on the table, especially when the Targets tilt levels are restored. This means that there are large, slowly moving signals in the T240 channels. The Blend start filter will have a 0 as the output of the T240 blend filter, by design, since the T240 is not used during start, but the output of the T240 NXT filter will typically not be 0 and so the difference can be quite large.
2. The ramp in use now is a straight linear ramp. As the Switcher moves from CUR bank over to the NXT bank, the output of the filter bank, 'Blend Out' for a sensor is

$$\text{Blend Out} = r \times \text{CUR} + (1 - r) \times \text{NXT} \tag{1}$$

   where r is the ramp function which moves, in this case, from 1 to 0. The ramp in use now is a linear ramp which moves from 1 to 0 in 5 seconds. Thus, it has distinct kinks at each end. To better illustrate the current problem, Blend Out can be rewritten as

$$\text{Blend Out} = r \times (\text{CUR} - \text{NXT}) + 1 \times \text{NXT} \tag{2}$$

3. The upper unity gain frequencies for the stage 1, level 3 control loops are about 40 Hz. To achieve this while compensating for the various roll-offs in the plant, the isolation controllers plus drive compensation filters have significant gain above 100 Hz.

3 Problem Simulation

We used NDS2 and LigoDV to pull data from the science frames. We pull the H2 drive channel (which caused the glitches) which is recorded at 512 samples/ sec. We pulled the outputs of the various blend filters, CUR and NXT, for the X channel CPS, T240, and L-4C. These data are from the epics OUT16 channels, and only saved at 16 samples per second. We resample them up to 4096. We load the X isolation filter and the H2 drive filters from the (somewhat later, but essentially the same) foton filters. We simulate the 4096 data into the X isolation control filter by applying the ramp function to the resampled data at the output of the CUR and NXT filters. This is an inexact method, but it clearly shows glitches in the simulated output at the same time we see glitches in the recorded output. As we will see, in the interest of getting the problem fixed quickly, the simulation calculations are only cursory. The matlab code for this is in {SeismicSVN}/seismic/BSC-ISI/Stanford/glitch_calc/plot_glitch_info.m.

Figure 2 shows the transfer function of the Level-3 Isolation controller for Stage 1 X of ETMX (non-boosted), the output filter for the stage 1, H2 coil driver, and the full path which the product

of the 2 control filters and the -2/3 projection element of the CART2DRIVE output matrix. We only plot the magnitude because it the large magnitude at high frequency which is the item of interest here.

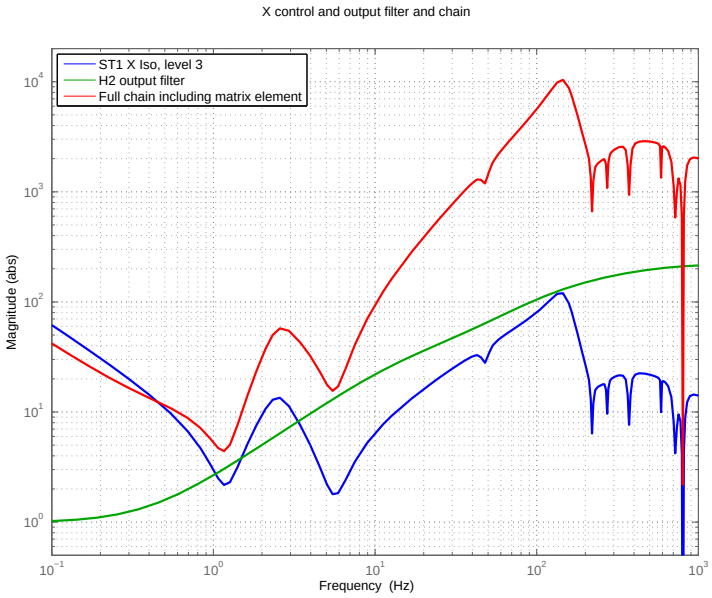

Figure 2: Magnitude of the isolation drive path. This includes the stage 1, Level 3, X controller, the -2/3 projection from X to H2, and the H2 drive output filter.

The gain peaks around 150 Hz, so it is not surprising that small, high frequency glitches at the input will generate big peaks at the output.

In figure 3, we see a series of glitches in the drive signal for H2. There are 3 glitches, located at the WD trip, and 5 and 10 seconds before. NOTE that the timing here is a bit rough, since much of the data is reconstructed from epics channels. These times correspond with the start of the first ramp (-10 sec), the end of the first ramp (-5 seconds), and the start of the second ramp (zero seconds) as seen in other data not plotted (from the MIXSTATE channel). We are going to try simulate this first glitch.

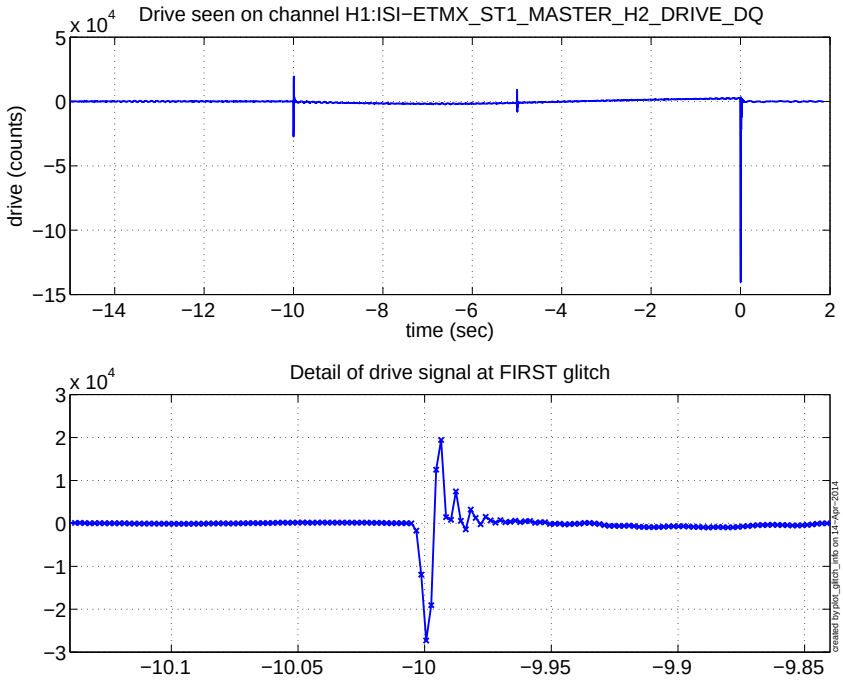

Figure 3: Glitches seen in the H2 drive channel. The detail of the first glitch is shown in the lower plot. The data samples are1/512 sec apart, so the glitch is very fast. Note that this channel has been down sampled for the framebuilder, so there is likely some signal distortion.

To simulate the glitch, we first make a reconstruction of the CUR and NXT outputs of the 3 sets of blend filters (CPS, T240, and L-4C). We start with the 16 Hz epics data and resample it up to 4096, with the matlab resample(channel, 4096, 16, 3) command. At the fast rate, we multiply it by the fast ramps and add the ramped CUR and NXT signals together to estimate the inputs to the Isolation filter. The 3 signals, and their sum, can be seen in figure 4. Figure 5 is a detail view of those inputs at the first glitch. The thing to note is that the T240 signal has a sharp glitch in it. This is largely due to the fact that the CUR blend filter is identically zero, since it is the 'Blend Start' and the NXT filter has a large input, and hence a large output. This means that the shape of the sum will look very much like the shape of the ramp.

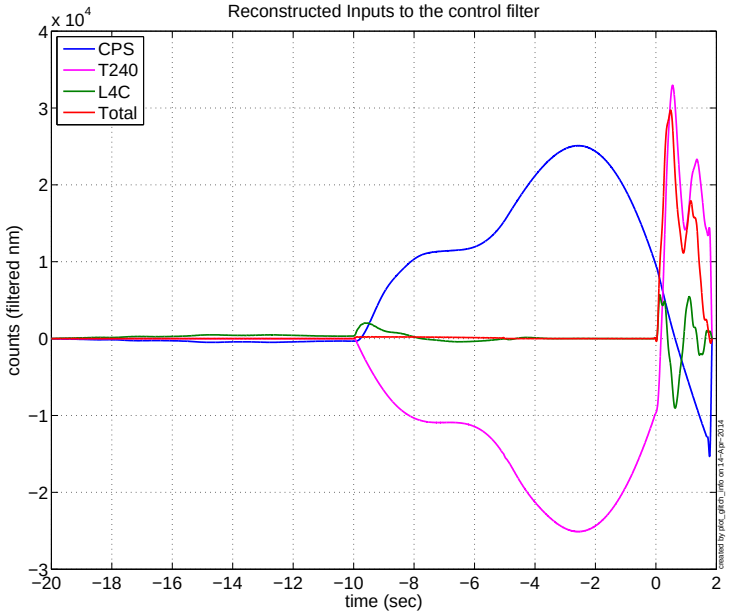

Figure 4: Estimated inputs to the isolation filter coming from the blend filters.

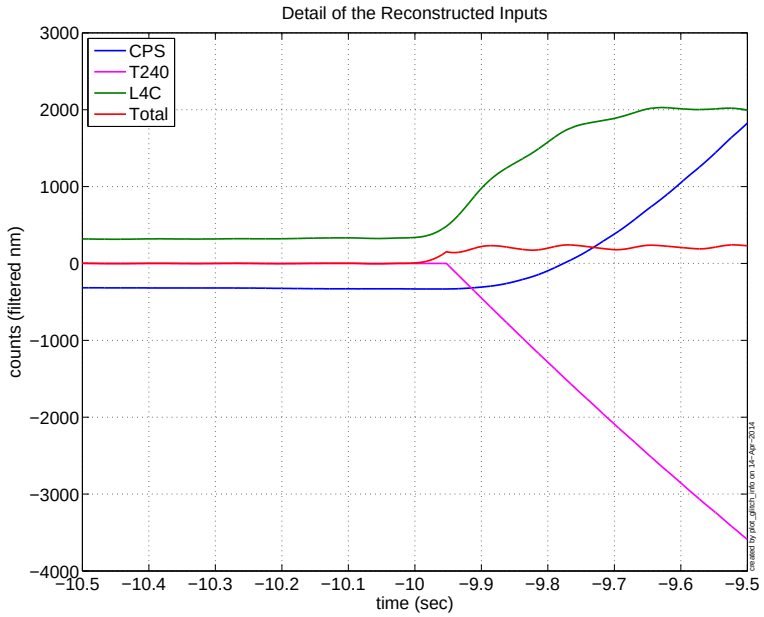

Figure 5: Detail of the estimated inputs to the isolation filter coming from the blend filters. Note the kink in the T240 signal.

Using these estimated inputs to the isolation filters, we were able to simulate the output of the drive chain. The measured output and the simulated output are reasonably close, as can be seen in figure 6. The most interesting thing to see in the simulation is that if we just use the T240 input

to the isolation filter, and calculate what the output will be, we see that the glitch clearly remains. This is good evidence that the glitch is, as we expect, coming mainly from the T240 part of the blends. The simulated output using only the T240s after the glitch is clearly not correct, which is what one would expect because the other sensors (mainly the CPS) are canceling the T240 signal at low frequencies.

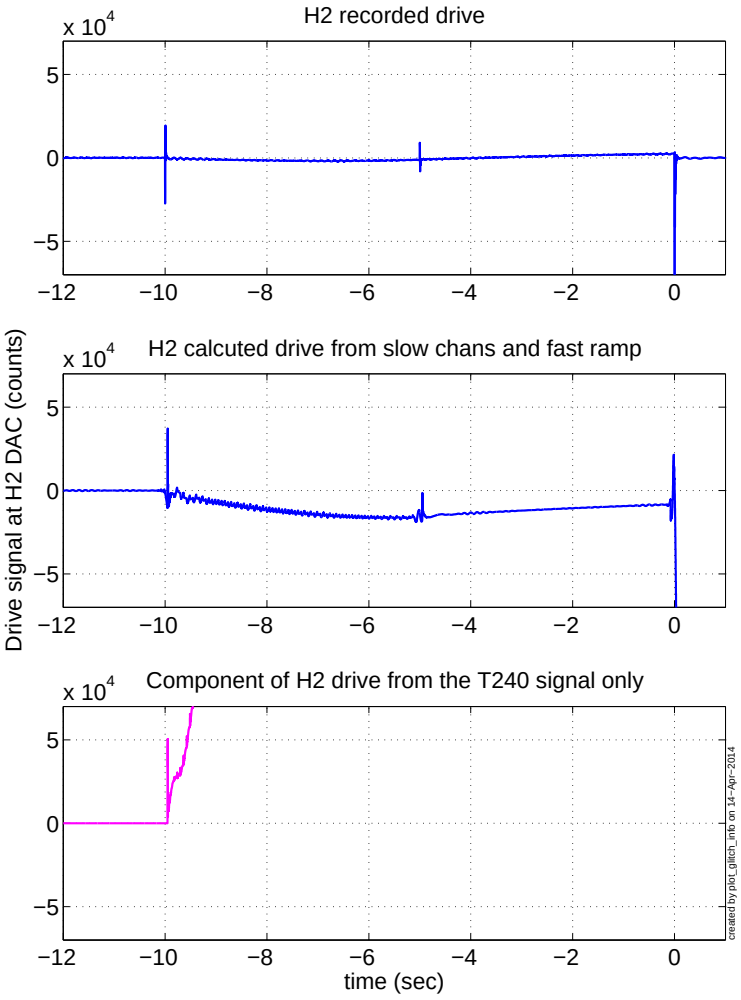

Figure 6: Measured (top) and estimated (middle) outputs of the X to H2 drive path. Note the glitches are very similar. The bottom trace shows the output of the filter chain if only the T240 signal is used. The glitch is nearly the same, indicating that most of the trouble is coming from the T240 components of the blend switch (in this case, at least).

4 The P5 ramp

We have developed a smooth, fifth order polynomial ramp where the initial and final velocities and accelerations are 0. This was developed for the bias restore function, and is called the P5 ramp. See

T1300510 for more information. We reran the simulation using that ramp to estimate the inputs to the blend filters. The comparison between the linear ramps and the P5 ramps can be seen in figure 7.

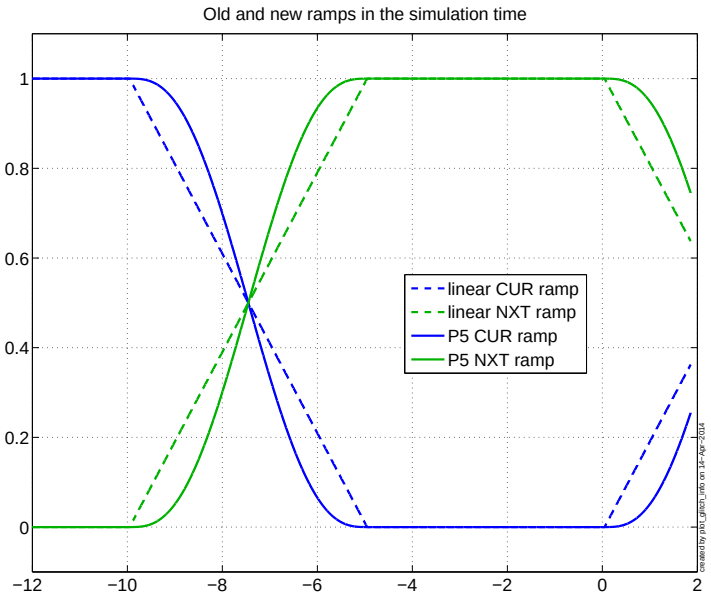

Figure 7: The time history of the linear blend ramps (dashed line) and the new P5 ramps. The P5 ramps are much smoother.

Finally, we use the P5 ramps to simulate the input to the isolation filters during a blend switch. This estimation is only accurate at the very beginning of the blend switch process because we are not using a plant model. However, if we look at figure 8 what happens to the glitch in the output of the H2 driver, in the one case using the T240 input with the linear ramp, and in the other using the T240 input with the P5 ramp, it is clear that the P5 ramp eliminated the big glitch.

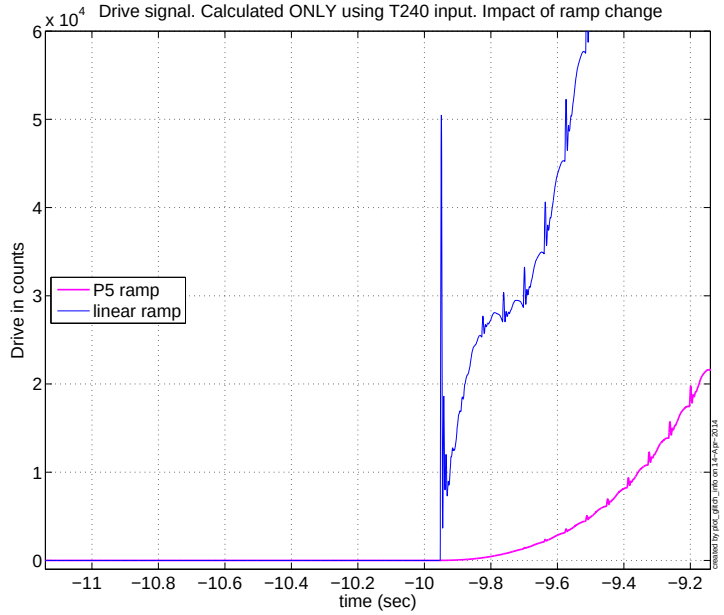

Figure 8: Simulated outputs of the control filter. The blue curve uses the linear ramp and makes the big glitch seen earlier. Using the P5 ramp eliminates the glitch.

5 Implementation

We have implemented the P5 ramp into the blend switching code, and run it in the HAM-ISI model at Stanford. The new code is called BLENDMASTER_P5.c and replaces BLENDMASTER.c in the {userapps}/trunk/isi/common/scr/ directory. There is no library part for the blend switching, so each of the 18 modules (6 for HAM-ISI, 12 for BSC-ISI) need to be updated by hand. Once the master models are updated, a simple rebuild of the chamber models will be adequate to effect the change. There is no change visible to the user. One can watch the ...BLND_X_MIX epics variable during a blend switch to confirm the ramp has been changed.

6 Implementation testing

The code was implemented in the ITMX model controlling the prototype at Stanford and used to control a blend switch under circumstances meant to generate the glitches. Stage 1 RX and RY loops were closed with level 3 style boosted controllers and X was controlled with level 3 style non-boosted controllers. These loops started with the 'blend-start' controllers. The other stage 1 DOFs were damped, and the stage 2 DOFs were all damped. The RY bias was changed by 1000 nanoradians, and about 10 sec after the tilt command, the stage 1 X blend was changed from 'Blend-start' to the 'HAMFET' blend. This blend has a pretty low blend frequency and uses the T240 signals. Note that the Stanford prototype uses STS-2 sensors rather the T-240s. Figure 9 shows the drive signal at the H2 DAC output (in counts). The top trace shows the drive with a linear ramp in the blend switch. The large glitches at 0, 5, 10, and 15 seconds correspond to the start and stop of the two ramps. The large, 10 sec period drive signal results from the attempt to compensate for the large signals present on the T240 channels. The second plot shows a repeat

of the blend switch when the ramp has been changed from the linear ramp to the P5 polynomial ramp. The glitches are not evident, but the 10 second motion is still quite clear.

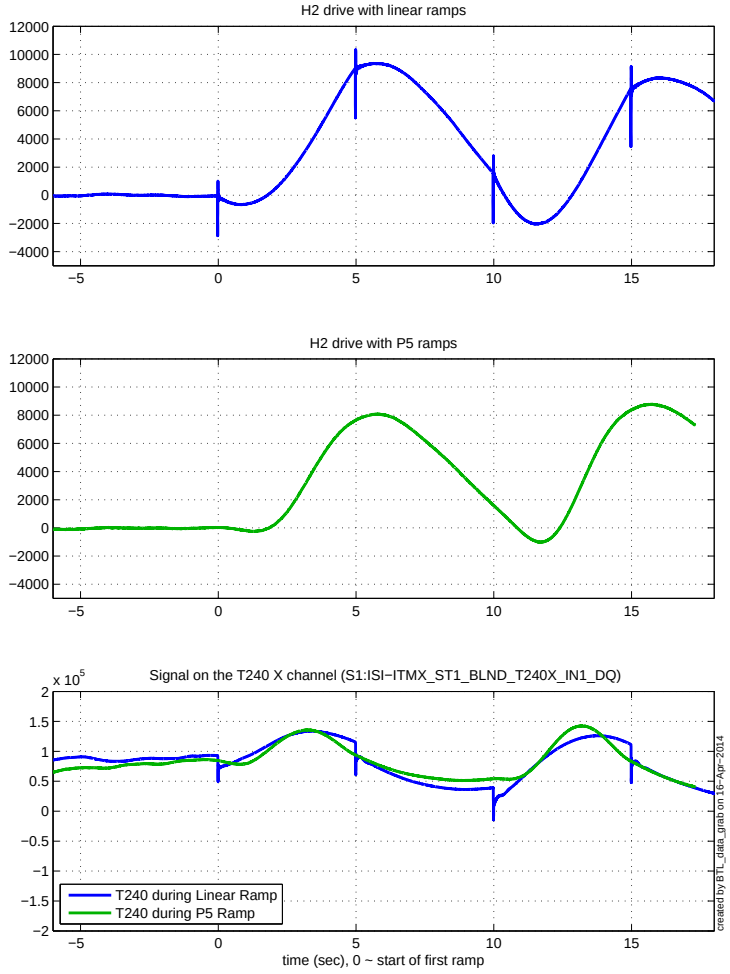

Figure 9: Measured drive to the H2 DAC during a blend switch for Stage 1 X after a large RY tilt. The top trace shows the glitches from the linear ramp. The middle trace is a repeat of the exercise using the new P5 ramp. The glitches are no longer evident, although the large drives with the 5 and 10 second time scales are about the same. The bottom trace shows the calibrated cartesian basis T240 (STS-2 at Stanford) signals, which are quite large. The sensors are (falsely) reporting average velocity of about 70,000 to 80,000 nm/sec at the start of the blend switch. This signal comes from the 1000 nanoradian tilt imposed about 10 sec before the blend switch.

The actual ramp as implemented on the front-end and retrieved from the framebuilder is shown in figure 10. It is smooth and asymptotes to 0.0 and 1.0 as it should. A detailed plot of the 4 corners is also shown in figure 11. The tops of the ramps are not as smooth as the bottoms. I suspect this is the result of some numerical issue in C. The matlab version of the same calculation is symmetric. This does not seem to cause any practical issue, but is noteworthy.

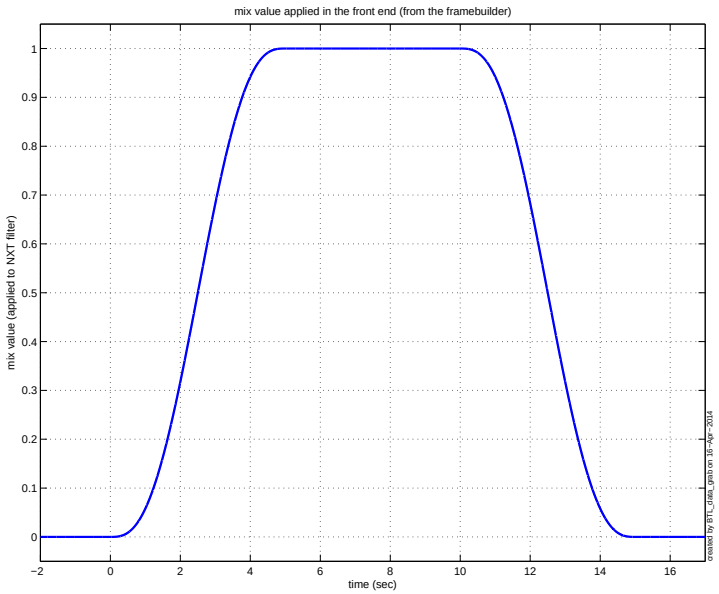

Figure 10: Measured P5 ramp set from S1:ISI-HAMX_X_MIX_TP during a blend switch. The shape is as expected.

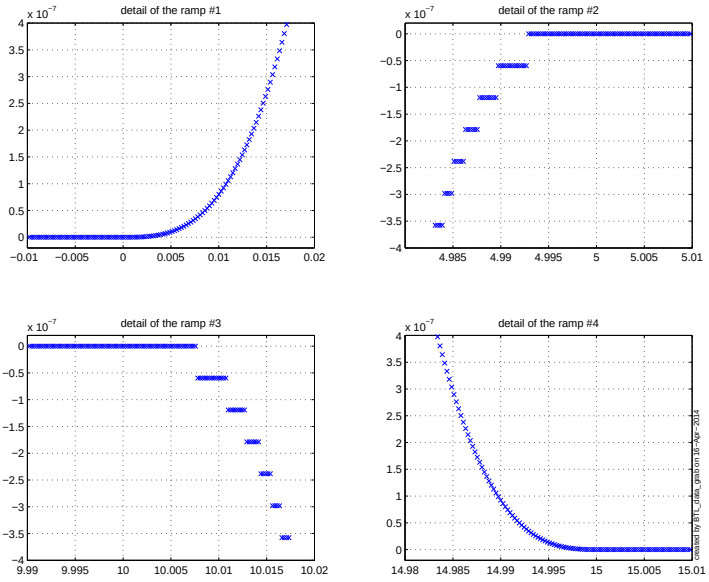

Figure 11: Details of the 4 corners of the ramp in figure 10. The top edges are pixelated, and I don't understand why. The top edges are plotted as RampValue - 1 for display reasons.

7 New Source Code

Below is the listing of the source code found in {userapps}/trunk/isi/common/src/BLENDMASTER_P5.c

```
/* BLENDMASTER.c Function: MASTERMIXBLENDS
 *
 * This function switches smoothly between two different blend filters. It
 * uses the NEXT bank of filters to switch to temporarily, then switches
 * back to the original filter. During this switch, we use the cdsFilt
 * with ctrl to control which filter modules are turned on. The input
 * DESIRED_FM is tied to an EPICS momentary. It is used to determine
 * whether the we should switch - a switch is initiated if the current
 * value for DESIRED_FM > 0. Once switching begins, the code ignores any
 * changes to DESIRED_FM.
 *
 * We use a state machine architecture to perform the switching. We turn
 * on the requested FM in the second bank, wait for it to settle, ramp
 * from the first to the second bank, switch the first bank to the
 * requested FM, wait for it to settle, and ramp back to the first bank.
 *
 * Inputs:
 *
 * int desired_fm: the filter module the user wants to switch to
 *
 * Outputs:
 *
 * double mix_value: the ramping variable [0->1] used to combine the
 *      output from two blends
 * int cur_cin_bitmask: the filter modules that the C code wants on in the
 *      CURR bank
 * int cur_ctrl_bitmask: the CURR filter modules the C code controls
 * nxt_cin_bitmask: the filter modules that the C code wants on in the
 *      NEXT bank
 * int nxt_ctrl_bitmask: the NEXT filter modules the C code controls
 * BlendingState current_state: typedef'd enum variable keeping track of
 *      our mixing state
 *
 * Authors: CJK RK
 * January 25th 2012
 *
 *
 */

#define FULL_CONTROL 0b1111111111
#define NO_CONTROL 0b0000000000



typedef enum { WAIT_FOR_FM_SWITCH, WAIT_FOR_NEXT_SETTLE, MIX_TO_NEXT,
    WAIT_FOR_CUR_SETTLE, MIX_TO_CUR } BlendingState;

const static int binary_fm_on[11] = {
0b0000000000,
0b0000000001,
0b0000000010,
0b0000000100,
0b0000001000,
0b0000010000,
0b0000100000,
```

```
0b0001000000,
0b0010000000,
0b0100000000,
0b1000000000 };


// BTL mods on April 14, 2014 to use a P5 ramp.

const static int TOTAL_WAIT_TIME = (5 * FE_RATE);

const static int    TRAMP                       = 5;
const static int    TOTAL_MIX_TIME      = TRAMP * FE_RATE;
const static double StartRampValue      = 0.0;
const static double FinalRampValue      = 1.0;
const static double WaitingOutput       = 0.0; // mix value when holding - no signal
    from NXT filter.
const static double Xdiff = FinalRampValue - StartRampValue;  // Total change for
    the ramp
const static double Vmax = (1.875) * Xdiff/TRAMP;   // max velocity, computed from
    dX and dT



void MASTERMIXBLENDS(double *argin, int nargin, double *argout, int nargout){

        static int wait_timer = 0; // Waiting for filter history to catch up
        static int mix_timer = 0;  // Switch between different filters
        static int next_fm = 1;    // The filter module we are switching to
        static BlendingState current_state = WAIT_FOR_FM_SWITCH;
        static int cur_cin_bitmask = 0b0;
        static int cur_ctrl_bitmask = NO_CONTROL;
        static int nxt_cin_bitmask = 0b0;
        static int nxt_ctrl_bitmask = NO_CONTROL;
        static double RpC[6];  // these are the polynomial Ramp Coefs.

    // The filter module we want to switch to (0 when no switch requested)
        int desired_fm = argin[0];

        double tt;      // time from ramp start, but scaled as -T/2 -> T/2.
        double mix_value; // start at 0;

        // RpC are the Ramp Coefficients
        static int starting = 1;

        // precalculate the ramp coefs.
        if (starting == 1) {
                RpC[0] = StartRampValue + (0.5 * Xdiff);
                RpC[1] = Vmax;
                RpC[2] = 0.0;
                RpC[3] = (-2.6666666666667/(TRAMP* TRAMP)) * Vmax;
                RpC[4] = 0.0;
                RpC[5] = (3.20/(TRAMP*TRAMP*TRAMP*TRAMP)) * Vmax;
                starting = 0;
        }

        // STATE SWITCH
        switch(current_state){
        // STATE 0: Waiting for command, then turn on NEXT bank filter module
        case WAIT_FOR_FM_SWITCH:
```

```
        if(wait_timer == 0 && desired_fm > 0 && desired_fm < 11){
                next_fm = desired_fm;
                nxt_cin_bitmask = binary_fm_on[next_fm];
                nxt_ctrl_bitmask = FULL_CONTROL;
                // Don't take control of the CURR bank since we don't know
                    what FM
                // it has loaded in
                current_state = WAIT_FOR_NEXT_SETTLE;
                ++wait_timer;
        } else {
                cur_ctrl_bitmask = NO_CONTROL; // back to waiting, give up
                    control
                nxt_ctrl_bitmask = NO_CONTROL;
        }
        break;
//STATE 1: Waiting for NEXT bank history to settle
case WAIT_FOR_NEXT_SETTLE:
        if(wait_timer < TOTAL_WAIT_TIME){
                ++wait_timer;
        } else{
                current_state = MIX_TO_NEXT;
        }
        break;
//STATE 2: Ramping to NEXT bank, then switch CURR bank to requested FM
case MIX_TO_NEXT:
        if(mix_timer < TOTAL_MIX_TIME){
                ++mix_timer;
        } else{
                cur_ctrl_bitmask = FULL_CONTROL;
                cur_cin_bitmask = binary_fm_on[next_fm];
                current_state = WAIT_FOR_CUR_SETTLE;
        }
        break;
//STATE 3: Waiting for CURR bank history to settle
case WAIT_FOR_CUR_SETTLE:
        if(wait_timer > 0) {
                --wait_timer;
        } else{
                current_state = MIX_TO_CUR;
        }
        break;
//STATE 4: Ramping back to CURR bank, then switch all NEXT bank FMs off
case MIX_TO_CUR:
        if(mix_timer > 0){
                --mix_timer;
        } else{
                nxt_cin_bitmask = binary_fm_on[0];
                current_state = WAIT_FOR_FM_SWITCH;
        }
        break;
}
// The weighting given to the CURR and NEXT outputs ((1-x)*CURR + x*NEXT)
// so the ramp will start at 0 and go to 1, hold at 1, then ramp back down
    to 0.
// the RampCoef are pre-calculated, as described in T1300128

if (current_state == WAIT_FOR_FM_SWITCH){
        mix_value = WaitingOutput;
}
```

```
        else if (current_state == WAIT_FOR_NEXT_SETTLE) {
                mix_value = StartRampValue;
        }
        else if ((current_state == MIX_TO_NEXT) || (current_state == MIX_TO_CUR)) {
                tt = (2.0 * mix_timer) - TOTAL_MIX_TIME;
                tt = (0.5 * tt)/(1.0 * FE_RATE);  // cast to double before the
                    divide
                // RC[5]*tt^5 + RC[4]*tt^4 + ... RC[0]
                mix_value = ((((RpC[5]*tt + RpC[4])*tt + RpC[3])*tt + RpC[2])*tt +
                    RpC[1])*tt + RpC[0];
        }
        else if (current_state == WAIT_FOR_CUR_SETTLE) {
                mix_value = FinalRampValue;
        }
        else {
                // this should never happen
                mix_value = WaitingOutput;
        }

        argout[0] = mix_value;
        // The filter modules that the C code wants on in the CURR bank
        argout[1] = cur_cin_bitmask;
        // The filter modules the C code has control of in the CURR bank
        argout[2] = cur_ctrl_bitmask;
        argout[3] = nxt_cin_bitmask;
        argout[4] = nxt_ctrl_bitmask;
        // Output the int value of the current state
        argout[5] = current_state;
}
```