Will It Blend?:
Blend Switching User Guide

Ruslan Kurdyumov

March 7, 2012

Overview

We have implemented real-time switching between blend filters for ITMX. The switching relies on two identical filter banks for each DOF, the CURR and NEXT bank, each of which contains all the possible blend filters. We use a C code block in Simulink to turn on/off filter modules in the two banks and smoothly ramp from the output of one bank to another. A Perl script interfaces with MEDM and triggers the C code.

Contents

1 Operation

We have designed blend-switching with ease-of-use in mind. As a result, all the implementation details are hidden from the user. To complete a switch, the user interacts with the blend MEDM screen.

1.1 The Blend Screen

At the lowest level, the user can switch between blends for a given degree of freedom by clicking the desired blend. Below, we have the MEDM screen illustrating the state of the blend filters for the Stage 1 X DOF, which blends 3 sensors: CPS, T240, and L4C:

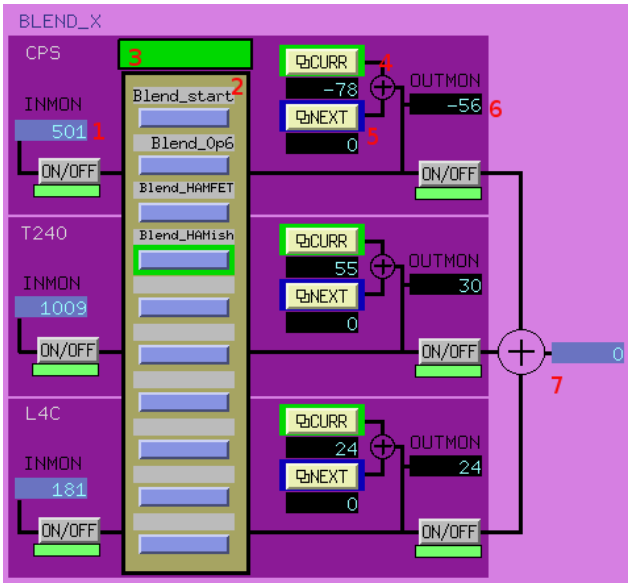


Figure 1: A typical blend MEDM screen for one DOF.

The important parts of the screen shown in Figure 1 are numbered in red and explained below:

1. The input signal for the sensor to be blended. In this case: the CPS X signal.
2. The blend bank, a bank which illustrates the possible blends we can choose from, and highlights the currently active blend in green. We can choose another blend by clicking on its button. In this case, the Blend_HAMISH blend is currently active, and we can click on another blend button to switch blends.
3. The progress bar, which visually indicates how far along a blend switch we are. In this case, the bar is solid green, indicating that the switch is complete and only one blend is active. During switching, the bar will progressively light up yellow indicator bars tracking how far along the switch we are.
4. The CURR blend output signal, which displays the output of the currently active blend (the one highlighted green in the blend bank). In this case, this is the Blend_HAMish blend.
5. The NEXT blend output signal, which displays the output of the blend we are switching to. In this case, we arent switching, so the next bank is off and the output signal is 0.
6. The mixed blend output, which is the weighted sum of the CURR and NEXT outputs, depends on the switching progress. In this case, we are not switching, so the CURR output is weighted 1 and the NEXT output is weighted 0. At this point, all the sensors are in comparable units.
7. The supersensor signal, which combines the mixed blend outputs to produce an overall signal for the given DOF. It happens to be 0 since we are under isolation.

1.2 Switching Between Blends

To switch to a different blend, you click the button of the desired blend. A screen like the one below will appear:

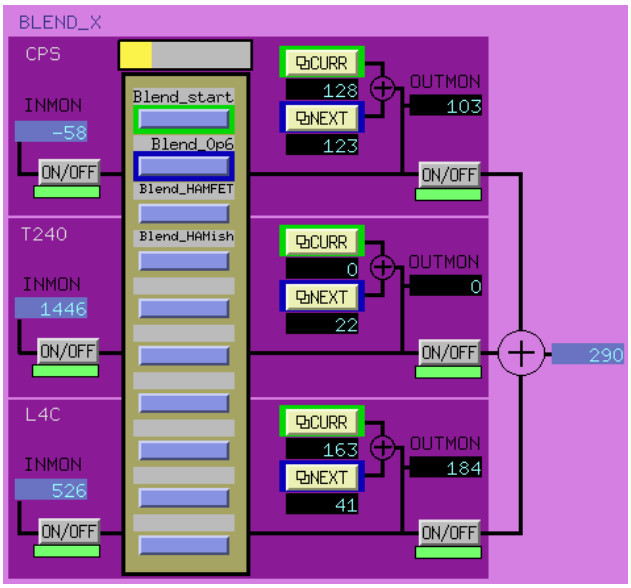


Figure 2: A typical blend MEDM screen switching from "Blend_start" to "Blend_0p6".

The progress bar will update as the switch progresses. Yellow bars will light up, indicating the following states:

1. Waiting for NEXT bank to settle
2. Ramping to NEXT bank
3. Waiting for CURR bank to settle
4. Ramping back to CURR bank.

The bar will go back to solid green when switching is complete. The mixed blend output (OUTMON) will also track the weighted sum of the CURR and NEXT blend outputs. In this case, we are switching from the Blend_start to the Blend_0p6 blend.

If you want to see the CURR and NEXT filter banks for a given sensor DOF for debugging purposes, you can click on the CURR and NEXT buttons to bring up the standard MEDM screens. In Figure 3, we are switching from Blend_HAMish (FM4) to Blend_0p6 (FM1).

You should avoid manually turning on/off filter modules in the CURR and NEXT screens except when debugging. During blend switching, the C code will have control of the filter modules, so you will not be able to turn them on or off.

1.3 Switching Multiple DOF

For convenience, we have included a SWITCH ALL MEDM screen, which can be accessed from the top of the overall blend filter MEDM screen. It is shown in Figure 4. This screen allows switching all the DOFs for a given stage to the selected blend simultaneously. Note that to use this functionality, matching blends must be loaded into the same filter modules for ALL the degrees of freedom for a given stage. In this case, FM1 has the Blend_start blend loaded in for all the stage 1 DOF, FM2 has Blend_0p6, etc.

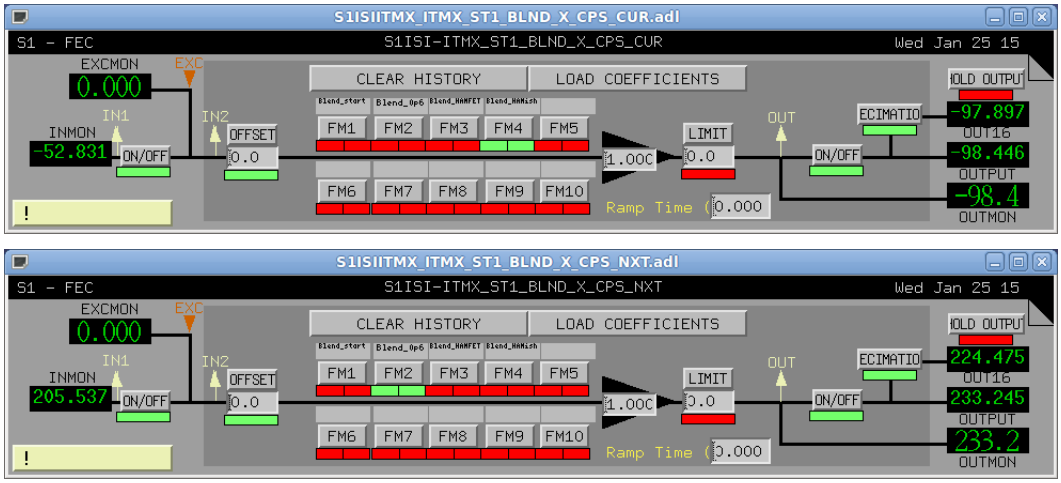


Figure 3: The CURR and NEXT blend screens for a given DOF.

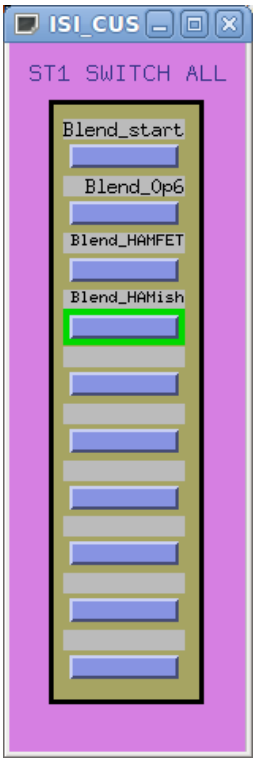


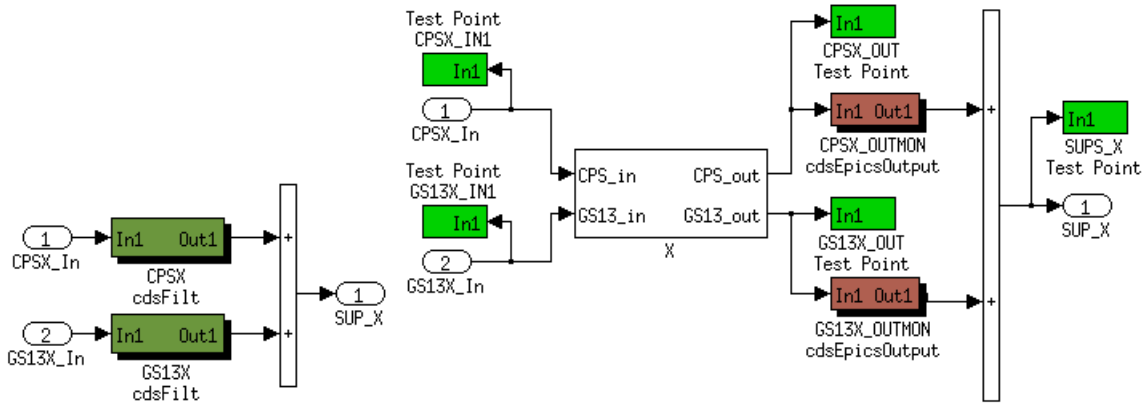
Figure 4: The SWITCH ALL blend screen, which simultaneously switches all the DOF to the selected blend.

2 Implementation

To implement real-time blend-switching, we rely on a modified Simulink diagram, C code to ramp between blends, and a Perl script to trigger the C code and perform housekeeping.

2.1 Simulink

The old and new blend filter diagrams for a single DOF are shown in Figure 5:



(a) Old (b) New

Figure 5: The old and new blend Simulink implementations.

From left to right, we read in the sensor inputs for a given DOF. These inputs are fed into the X block, where blend filters are applied to the sensors. The X block blend outputs are summed to create the supersensor signal. Switching between different blend filters is done inside the X block. We include the Test Points before and after the blending for backwards compatibility in naming conventions and writing the blend output to the framebuilder at the model rate, respectively. The Epics outputs are used in the blend MEDM screens.

All the actual work is done inside the X block, shown in Figure 6.

2.1.1 Blend filter modules

The key part of the diagram is the presence of two blend filter modules for each sensor which are used to smoothly switch between blends. Note that the CPS_In signal on the far right feeds into CPS_CUR and CPS_NXT cdsFiltCtrl modules in the middle. Each cdsFiltCtrl module takes three inputs: In1, Cin, and Mask, from top to bottom. In1 is the incoming data signal. Cin controls which filter modules are on. Mask determines whether the Cin input or the operator has control of the filter modules.

2.1.2 MASTERMIXBLENDS C function call

The MASTERMIXBLENDS cdsFunctionCall block contains the C code that governs blend switching. The block takes as input DESIRED_FM, a momentary variable set to 1-10 by Perl when requesting a switch, and 0 otherwise. The block has 6 outputs, from top to bottom: MIX, Cin_CUR, Mask_CUR, Cin_NXT, Mask_NXT, and MIXSTATE. MIX outputs the mixing variable [0-1], which gives weights to the CURR and NEXT blends. Cin and Mask switch on the proper FM when switching. MIXSTATE is used by MEDM to update the progress bar and Perl for housekeeping.

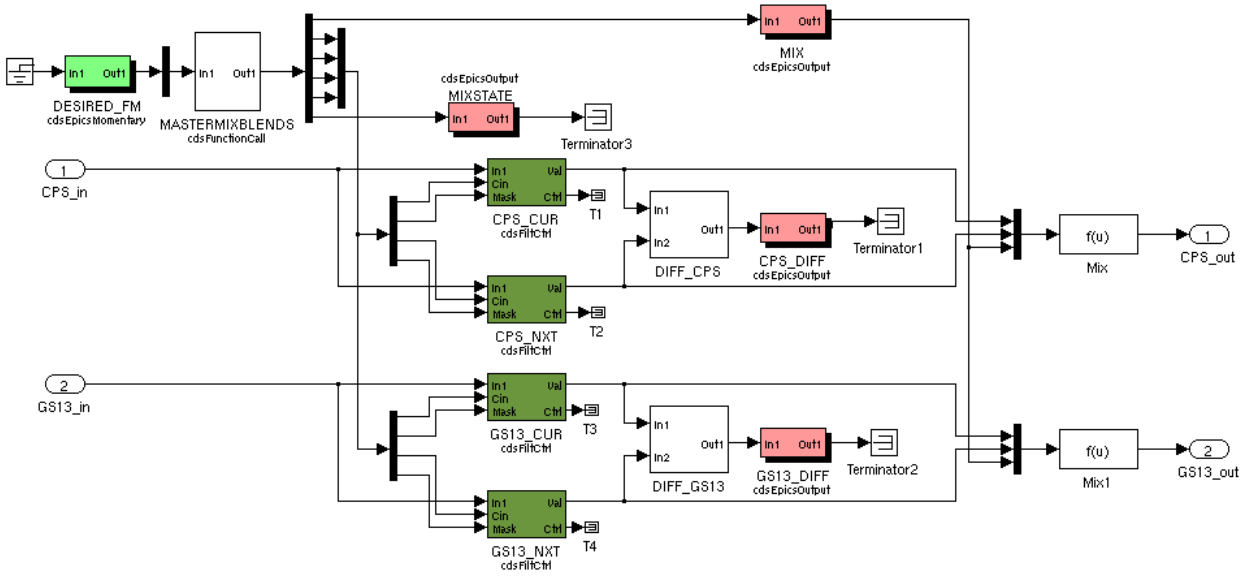


Figure 6: The low-level Simulink blend switching implementation for 1 DOF.

2.1.3 Mix function block

The combining of the CURR and NEXT blend outputs is done in the Mix function block using the following logic (CPS used as an example):

$$\text{CPS\_out} = (1 - \text{MIX}) * \text{CPS\_CUR} + \text{MIX} * \text{CPS\_NXT}$$

During normal operation, MIX = 0, so the output is simply the CURR blend. During switching MIX smoothly varies from $0 \rightarrow 1$ (and back from $1 \rightarrow 0$), so the output is a weighted sum of the CURR and NEXT blends.

2.1.4 DIFF block (unused)

Note that we have included a cdsEpicsOutput called DIFF, a low-pass filtered signal indicating the difference between the CURR and NEXT bank outputs. The motivation for this variable is the following: rather than switching for a fixed time, we switch and wait for the DIFF signal to settle before considering our switch complete. We have chosen not to use the DIFF signal for switching because testing showed our CURR and NEXT signals converged quickly in the fixed time approach. In addition, a fixed time switch guarantees that any requested switch is completed. However, the filter has been tested and could be used in future implementations.

2.2 C code

The commented C code is attached below. To summarize, the C code waits for the DESIRED_FM input to change to a FM 1-10, turns on the requested FM in the NEXT bank, ramps to it, turns on the requested FM in the CURR bank, ramps to it, and goes back to waiting.

```
/* 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
#define TOTAL_MIX_TIME (5 * FE_RATE)
#define TOTAL_WAIT_TIME (5 * FE_RATE)

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 };

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


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

```
77    case WAIT_FOR_FM_SWITCH:
        if(wait_timer == 0 && desired_fm > 0 && desired_fm < 11){
79        next_fm = desired_fm;
          nxt_cin_bitmask = binary_fm_on[next_fm];
81        nxt_ctrl_bitmask = FULL_CONTROL;
          // Don't take control of the CURR bank since we don't know what FM
83        // it has loaded in
          current_state = WAIT_FOR_NEXT_SETTLE;
85        ++wait_timer;
        } else {
87        cur_ctrl_bitmask = NO_CONTROL; // back to waiting, give up control
          nxt_ctrl_bitmask = NO_CONTROL;
89      }
        break;
91    //STATE 1: Waiting for NEXT bank history to settle
      case WAIT_FOR_NEXT_SETTLE:
93      if(wait_timer < TOTAL_WAIT_TIME){
          ++wait_timer;
95      } else{
          current_state = MIX_TO_NEXT;
97      }
        break;
99    //STATE 2: Ramping to NEXT bank, then switch CURR bank to requested FM
      case MIX_TO_NEXT:
101     if(mix_timer < TOTAL_MIX_TIME){
          ++mix_timer;
103     } else{
          cur_ctrl_bitmask = FULL_CONTROL;
105       cur_cin_bitmask = binary_fm_on[next_fm];
          current_state = WAIT_FOR_CUR_SETTLE;
107     }
        break;
109   //STATE 3: Waiting for CURR bank history to settle
      case WAIT_FOR_CUR_SETTLE:
111     if(wait_timer > 0) {
          --wait_timer;
113     } else{
          current_state = MIX_TO_CUR;
115     }
        break;
117   //STATE 4: Ramping back to CURR bank, then switch all NEXT bank FMs off
      case MIX_TO_CUR:
119     if(mix_timer > 0){
          --mix_timer;
121     } else{
          nxt_cin_bitmask = binary_fm_on[0];
123       current_state = WAIT_FOR_FM_SWITCH;
        }
125     break;
    }
127 // The weighting given to the CURR and NEXT outputs ((1-x)*CURR + x*NEXT)
    double mix_value = (double) mix_timer / TOTAL_MIX_TIME;
129 argout[0] = mix_value;
    // The filter modules that the C code wants on in the CURR bank
131 argout[1] = cur_cin_bitmask;
    // The filter modules the C code has control of in the CURR bank
133 argout[2] = cur_ctrl_bitmask;
    argout[3] = nxt_cin_bitmask;
135 argout[4] = nxt_ctrl_bitmask;
    // Output the int value of the current state
137 argout[5] = current_state;
}
```

/opt/rtcds/userapps/release/isi/common/src/BLENDMASTER.c

2.3 Perl

The commented Perl code is attached below.

The script is rather self-explanatory, and much of the code is building the correct data structures and strings that enable easy switching later on. There is also some housekeeping that prevents the user from starting a switch while there is already a switch in progress or when attempting a switch to an empty filter module. The core of the code begins at line 110, when the script sets the DESIRED_FM Epics momentary to start the switching process. The script then waits until the C code has finished switching to the NEXT bank and sends a request for the REQUESTED_FM to the CURR bank. Once the C code has surrendered control, this request will be fulfilled, preventing the filter module from reverting to its original state.

```
#!/usr/bin/perl -I /ligo/cdscfg

# masterSwitchBlendFilters
# usage: ./masterSwitchBlendFilters system stage DOF FM [args]
#
# This script switches between two different blend filters in real time.
#
# Author: CJK 2011 Sep 7
#

use strict;
use warnings;
use Getopt::Std;
use stdenv;
INIT_ENV($ENV{IFO});
use CaTools;
use 5.010;

sub usage{
  print <<USAGE
usage: ./masterSwitchBlendFilters system stage FM [dofs]

  <system> is the name of the system, formatted as ifo:sys-chamber, e.g. s1:isi-itmx
  <stage> is the stage, e.g. st1 or st2
  <FM>    is the filter module to switch to, e.g. FM2
  <dof>   is an optional list of the degrees of freedom to
     switch, e.g. X or RZ

USAGE
}

unless (@ARGV >= 3){
  &usage; die "Incorrect number of arguments.";
}

#verify we got a good subsys
my $SubSys = uc(shift);
my $state;
eval{ ($state) = caGet("${SubSys}_MASTERSWITCH") };
die "Error: bad subsys $SubSys." if ($state eq '');

# Read in all of the inputs
my $STAGE = uc (shift);
my $fm = uc(shift);
my @DOFS = @ARGV;
die "Error: improper FM argument." unless ($fm =~ m/FM[1-9]0?/);
my $fm_num = substr($fm, 2);
# Create arrays filled with the names of the switch readback channels for the correct stage
my @ST1_SENSORS = qw ( CPS T240 L4C );
my @ST2_SENSORS = qw ( CPS GS13 );
my @HAM_SENSORS = qw ( CPS GS13 );
if(!@DOFS) { @DOFS = qw ( X Y Z RX RY RZ ); }

# Create sensor names
my @SENSORS;
if( $STAGE =~ m/ST[12]/) {
```

```
  # Use stage 1 or stage 2 sensors for the BSC-ISI
  @SENSORS = ($STAGE eq 'ST1') ? @ST1_SENSORS : @ST2_SENSORS;
  $STAGE = $STAGE . '_';
} elsif( $STAGE eq 'HAM' ){
  # Use the HAM sensors for HAM-ISI
  @SENSORS = @HAM_SENSORS;
  $STAGE = "";
} else {
  die "Error: improper stage argument. Use ST1, ST2, or HAM.";
}

my $blend_prefix = "${SubSys}_${STAGE}BLND";
my $num_sensors = @SENSORS;
my $num_dofs = @DOFS;
my @DESIRED_FM_CHANS;
my @MIX_STATES;
my @CUR_FILTER_BANKS;
my @NXT_FILTER_BANKS;
foreach my $dof (@DOFS){
  foreach my $sensor (@SENSORS){
    push @CUR_FILTER_BANKS, "${blend_prefix}_${dof}_${sensor}_CUR"; # names for cur and next
        filter banks
    push @NXT_FILTER_BANKS, "${blend_prefix}_${dof}_${sensor}_NXT";
  }
  push @DESIRED_FM_CHANS, "${blend_prefix}_${dof}_DESIRED_FM"; # names for 'desired fm'
      channels
  push @MIX_STATES, "${blend_prefix}_${dof}_MIXSTATE"; # names for 'running' channels
}

# make sure we're not switching to an empty filter module
my ($name_num) = ($fm =~ /FM([1-9]0?)/);
$name_num = "0" . ($name_num-1);
my @NAME_CHANNELS = map { $_ . "_Name$name_num" } (@CUR_FILTER_BANKS, @NXT_FILTER_BANKS);
my @names = caGet(@NAME_CHANNELS);
foreach my $name (@names){
  if ($name eq ""){
    &usage;
    die "Error: Trying to switch to empty filter module.";
  }
}
# make sure we're not already running
my @runnings = caGet(@MIX_STATES);
foreach my $running (@runnings){
  if($running){
    die "Error: blend switching already running.";
  }
}
#Store the offsets in the appropriate place
my @OFFSET_CHANS_CUR = map{ $_ . "_OFFSET" } @CUR_FILTER_BANKS;
my @OFFSET_CHANS_NXT = map{ $_ . "_OFFSET" } @NXT_FILTER_BANKS;
my @offsets = caGet(@OFFSET_CHANS_CUR);
caPut(@OFFSET_CHANS_NXT, @offsets);

# Turn on next filter bank & set the DESIRED_FM Epics variable for the C code
my $command = "ALL OFF INPUT OUTPUT OFFSET DECIMATE $fm ON";
caPut(@DESIRED_FM_CHANS, ($fm_num) x @DESIRED_FM_CHANS);
sleep(1);
caSwitch(@NXT_FILTER_BANKS, ($command) x ($num_sensors * $num_dofs));

# wait for the switching to stop
my @outputs = caGet(@MIX_STATES);
my $requested = 0;
while(1){
  sleep(1);
  (@outputs) = caGet(@MIX_STATES);
  if(!$requested && $outputs[0] == 3) { # If we've taken control of the cur module
    caSwitch(@CUR_FILTER_BANKS, ($command) x ($num_sensors * $num_dofs));
    $requested = 1;
```

```
    } elsif($outputs[0] == 0){
124     last;
    }
126 }

128 # now that we're all done, turn the next filter bank off
caSwitch(@NXT_FILTER_BANKS, ("ALL OFF") x ($num_sensors * $num_dofs));
130 exit 0;
```

/opt/rtcds/userapps/release/isi/s1/scripts/masterSwitchBlendFilters

2.4 MEDM

Implementing the MEDM screens that display the blend switching progress is mostly straightforward. The trickiest part is displaying the blend bank. To highlight the current and next blend, we take advantage of the bit information in the SW1R and SW2R readback channels:

SW1R		SW2R	
Bit	State	Bit	State
2	Input switch	1	FM7
5	FM1	3	FM8
7	FM2	5	FM9
9	FM3	7	FM10
11	FM4	10	Output switch
13	FM5		
15	FM6		

Note that the odd bits represent whether a FM is on, while the even bits represent whether a FM request is has been made.

So, for example, let's check whether the NEXT bank is switching to a blend loaded into FM2. We'll check whether the Output switch of the next filter bank is on and whether FM2 is on. The visibility calculation is therefore: (SW1R & 2^7) && (SW2R & 2^{10}). If we wanted to check if both the Input and Output switches were on, it would be: (SW1R & ($2^7 + 2^2$)) && (SW2R & 2^{10}).

3 FAQ

3.1 Why use two filter banks instead of switching between filter modules in one bank?

There is no way to smoothly switch between filter modules in one bank.

3.2 Why switch back to the CURR bank? Can't you just alternate which bank is the CURR bank?

You can and it would make the switching faster. The basic answer - it's easier to display in MEDM. The highlighting of the current blend and the display of the current blend output rely on the fact that the CURR bank is fixed - it's always bank 1. You can alternate CURR banks and design other displays, but we didn't think they would be as intuitive.

3.3 What happens if I try to switch blends while switching is active?

The Perl script called by MEDM will ignore any switch requests that include an actively switching blend. Therefore, if CPSX is switching and you choose to SWITCH_ALL to a different blend, the SWITCH_ALL request will be ignored even though 5 of the 6 DOFs are free to switch.

3.4 What happens if I accidentally switch to my current blend?

The C code will load your request (the current blend) and switch to it. In practice, the only downstream effect will be ramping to an identical filter whose history was recently reset.

3.5 Why did you allow switching to the same blend?

We chose to allow this so that the C code did not have to keep track of which filter module it last switched to. If it had to do so, we would be forced to block the user from manually turning on/off filter modules to guarantee that the C code history was accurate.

3.6 What happens if I switch to an empty blend?

The Perl script will not allow you to switch to an empty blend. It checks whether the requested FM name field is empty.

3.7 Are the blend ramping and settling times fixed?

Yes, we have hardcoded the ramping and settling times to 5 seconds to keep the Simulink model simple. The times can be changed in the BLENDMASTER.c file.

3.8 In the Simulink model of the blend filters, why do you have a test point and an EPICS output for each blend output?

You need the test point to sample the blend output at the model rate and you need the EPICS output to display the blend output in MEDM.

3.9 Why can't I manually turn on/off filter modules during switching?

The C code takes control of the filter modules when switching to ensure that the switch can successfully complete. The only exception is that the CURR bank remains under user control for the first half of the switch (CURR→NEXT). Otherwise, the C code would need to know which CURR bank FM was on when the switch was requested, and turn it on when taking control.

3.10 How do I abort a blend switch in progress?

You can't. Since the C code has control of the filter bank, you will have to wait for the switch to complete.