Overview

This page describes the workings of C++ code isotwitch\_PIcontrol.

Additional resources

Need more help?

Check the resources, and then see Ken

Main content

**Part 1: Create DAPL Command**

The first part describes some necessary lines that are required to have the command work in DAPL. Important are the COMMAND and ENTRY lines, where the name if the command that is to be used in DAPL are specified. As for the other code here, it is probably best to go to Ken if you think it necessary to change these lines.

#define  COMMAND  "ISOTWITCHPICONT"

#define  ENTRY    ISOTWITCHPICONT\_entry

#include "C:\Program Files\Microstar Laboratories\DTD\Include\DTDMOD.H"

#include "C:\Program Files\Microstar Laboratories\DTD\Include\DTD.H"

int \_\_stdcall     ENTRY (PIB \*\*plib);

extern "C" \_\_declspec(dllexport) int \_\_stdcall

   ModuleInstall(void \*hModule)

{   return (CommandInstall(hModule, COMMAND, ENTRY, NULL)) ; }

**Part 2: Assign DAPL Parameters**

In the next part, the variables that will the specified in the DAPL code are assigned. If one or more variables are added to or removed from the code, the code will need to be adjusted at three different points here.  
First, the type of variable is specified (e.g. short int, long int, double).

// - - - - -  command implementation section  - - - - - - - - - -

int \_\_stdcall  ENTRY (PIB \*\*plib)

{

// Storage for parameters

void \*\*argv;

int argc;

PIPE \* in\_force\_pipe;

PIPE \* in\_fl\_pipe;

long int pre\_trigger\_samples;

short int arming\_level; //Does nothing anymore?

short int isotonic\_level;

short int isotonic\_increment;

long int max\_samples;

long int isometric\_points;

long int ramp\_points;

long int inter\_ramp\_points; //added

long int inter\_fl\_diff; //added

short int prop\_gain;

short int Ki; //added

short int fl\_polarity;

short int fl\_switch;

short int fl\_threshold;

short int smoothing\_points;

short int integral\_points; //added

short int gain\_asymmetry;

PIPE  \* out\_pipe;

After that the type of variable as used by DAPL is specified. The two numbers, in this case 19, 19, set the minimum and maximum number of variables that need to be assigned in the DAPL code. In this case all 19 variables need to be assigned.

// Access parameters

argv = param\_process (plib, &argc, 19, 19,

T\_PIPE\_W, T\_PIPE\_W,

T\_VAR\_L, T\_VAR\_W, T\_VAR\_W, T\_VAR\_L, T\_VAR\_L,

T\_VAR\_L, T\_VAR\_L, T\_VAR\_L, T\_VAR\_W, T\_VAR\_W,

T\_VAR\_W, T\_VAR\_W, T\_VAR\_W, T\_VAR\_W, T\_VAR\_W,

T\_VAR\_W, T\_PIPE\_W);

Lastly, the order in which the variables need to be assigned in the DAPL code are specified. Make sure to adjust the numbers if a new variable is added someplace between the existing variables.

in\_force\_pipe = (PIPE \*) argv[1];

in\_fl\_pipe = (PIPE \*) argv[2];

pre\_trigger\_samples = \*(long int volatile \*) argv[3];

isotonic\_level = \*(short int volatile \*) argv[4];

isotonic\_increment = \*(short int volatile \*) argv[5];

max\_samples = \*(long int volatile \*) argv[6];

isometric\_points = \*(long int volatile \*) argv[7];

ramp\_points = \*(long int volatile \*) argv[8];

inter\_ramp\_points = \*(long int volatile \*) argv[9];

inter\_fl\_diff = \*(long int volatile \*) argv[10];

prop\_gain = \*(short int volatile \*) argv[11];

Ki = \*(short int volatile \*) argv[12];

fl\_polarity = \*(short int volatile \*) argv[13];

fl\_switch = \*(short int volatile \*) argv[14];

fl\_threshold = \*(short int volatile \*) argv[15];

smoothing\_points =\*(short int volatile \*) argv[16];

integral\_points =\*(short int volatile \*) argv[17];

gain\_asymmetry =\*(short int volatile \*) argv[18];

out\_pipe= (PIPE \*) argv[19];

**Part 3: Additional Variables**

Additional variables are specified. These variables are mostly determined through calculations on the variables described above.

// Additional variables

int i;

short int in\_force\_value;

short int in\_fl\_value;

short int out\_value;

short int last\_out;

short int error\_value;

short int ramp\_increment;

short int out\_diff\_accumulator;  
    ...  
    ...

**Part 4: Perform Initializations**

In this part some initializations are performed, including reading in the two input pipes and writing out the output pipe.  
Double\_inter\_increment determines the slope for the interramp if that option is used.

// Perform initializations

pipe\_open(in\_force\_pipe, P\_READ);

pipe\_open(in\_fl\_pipe, P\_READ);

pipe\_open(out\_pipe, P\_WRITE);

armed=0;

was\_low=0;

isotonic\_hold=0;

isometric\_hold=0;

ramp\_back=0;

inter\_ramp=0; //added

run\_started=0;

sample\_counter=0;

inter\_ramp\_counter=0; //added

isometric\_counter=0;

ramp\_counter=0;

double\_inter\_increment = double(inter\_fl\_diff) / double(inter\_ramp\_points);

last\_out = 0;

out\_diff\_accumulator = 0;

double\_level = (double)isotonic\_level;

**Part 5: Create Buffers**

Since the PI-control makes use of values that have already been overwritten and have left the loop, these values need to saved in a buffer. This part allocates space for these values and sets the values within them to zero.

// Create space for the out and error buffers

last\_out\_buffer = (short int \*) malloc(smoothing\_points \* sizeof(short int));

error\_buffer = (short int\* ) malloc(integral\_points \* sizeof(short int));

// Zero buffers

for (i=0;i<smoothing\_points;i++)

{

last\_out\_buffer[i]=(short int)0;

}

for (i=0;i<integral\_points;i++)

{

error\_buffer[i]=(short int)0;

}

int temp\_counter=0;

long int error\_holder;

**Part 6: Start the Loop**

The while statement starts the complete loop within this manual command.

// Begin continuous processing

while (1)

{  
      
    ...  
  
    }

**Part 7: Find the Right Moment to Start Isotonic Hold**

Immediately when the command is used in DAPL, the sample\_counter starts running. After an amount of samples, specified by pre\_trigger\_samples, the system checks if the force has been below the isotonic hold level. If so, the moment the force rises above the isotonic\_level, isotonic\_hold is initiated.

   // Increment counter

sample\_counter++;

// Read input values

in\_force\_value = (short int) pipe\_get(in\_force\_pipe);

in\_fl\_value = (short int) pipe\_get(in\_fl\_pipe);

// Check if we are armed - that is, have collected enough samples

if ((armed==0)&&(sample\_counter>pre\_trigger\_samples))

armed=1;

// If we are armed, check we were below the isotonic level

if ((armed==1)&&(in\_force\_value<isotonic\_level))

was\_low=1;

// If we were low, and we are far enough into the record, have we got above the target?

// The run started variable prevents later re-triggering

if ((was\_low==1)&&(in\_force\_value>isotonic\_level)&&(run\_started==0))

{

run\_started=1;

isotonic\_hold=1;

}

// Now deduce behavior

out\_value=0;

**Part 8: Isotonic Hold**

During isotonic hold the system will try to keep the force exerted by the trabec on a steady level (i.e. on isotonic\_level).  
  
1. The error in force level is determined and written into the error buffer and error holder buffer (sum of errors) to be used later on in the loop)  
  
2. The integral gain Ki is corrected if necessary to counter the effect known as "droop". For more on this, see the *coding considerations* on the wiki or look up "pid control" and "voltage droop" on wikipedia.  
  
3. Now the new out\_value is calculated using the PI-control. Basically the formula would be  
Vo[n] = Vo[n-1] - ( Kp\*e[n-1] + Ki\*sum(e)\*dt )  
This has been adapted a bit to fit better with the system. Because the direction of the lengthening, a fl\_polarity multiplier has been included (this is either 1 or -1). The proportional gain Kp in the formula is written in the code as prop\_gain. Furthermore, not every previous errors are used for the sum of the errors. Finally, because of integer arithmetic, a correction factor of /10000 has been included.  
  
4. If either the out\_diff\_accumulator or the out\_value rises above (or below, depending on the polarity) a certain threshold, the system moves out of isotonic hold.  
  
5. Finally the isotonic\_level is updated. Again this is done to avoid the effect of "droop".

if (isotonic\_hold==1)

{

// Calculate force error

error\_value = (in\_force\_value - (short int)double\_level);

// Update error\_buffer

for (i=0;i<(integral\_points-1);i++)

{

error\_buffer[i] = error\_buffer[i+1];

}

error\_buffer[integral\_points-1] = error\_value;

// Sum the error\_buffer

error\_holder=0;

for (i=0;i<integral\_points;i++)

{

error\_holder = error\_holder + (long int)error\_buffer[i];

}

// Calculate changes in gain

if (out\_diff\_accumulator<0)

{

Ki2 = Ki + gain\_asymmetry;

}

else

{

Ki2 = Ki;

}

//Calculates out\_value with PIDcontrol, 100k is a constant that affects the gain params.

out\_value = last\_out -

(short int)

((long int)fl\_polarity \*

(((long int)prop\_gain \* (long int)error\_value) +

((long int)Ki2 \* (error\_holder / (long int)integral\_points)))/10000);

// Test for breaking out of isotonic hold and into isometric hold

if (fl\_switch==0)

{

// Code breaks out of control loop if out\_diff\_accumulator

// exceeds fl\_threshold. Use this to stop isotonic control

// when the motor reverses direction

// Need to check for both fl\_polarity options

if (((fl\_polarity>0)&&(out\_diff\_accumulator>fl\_threshold)) ||

((fl\_polarity<0)&&(out\_diff\_accumulator<-fl\_threshold)))

{

isotonic\_hold=0;

inter\_ramp=1;

}

}

else

{

// Code breaks out of control loop if motor returns to original value

if (((fl\_polarity>0)&&(out\_value>fl\_threshold)) ||

((fl\_polarity<0)&&(out\_value<-fl\_threshold)))

{

isotonic\_hold=0;

inter\_ramp=1;

}

}

// Update isotonic\_level for next iteration by adding increment

double\_level = double\_level + (double)((double)isotonic\_increment/1000.0);

double\_inter\_holder = last\_out;

}

**Part 9: Inter\_Ramp**

If this option is used, the system will have the trabec go into a ramp (either increase or decrease the held length linearly).

        if (inter\_ramp==1)

{

inter\_ramp\_counter++;

double\_inter\_holder = double\_inter\_holder + double\_inter\_increment;

out\_value = short int (double\_inter\_holder);

if (inter\_ramp\_counter>=inter\_ramp\_points)

{

inter\_ramp=0;

isometric\_hold=1;

// out\_value=0;

}

}

**Part 10: Isometric\_hold**

During isometric hold, the trabec will be held at a steady length for the duration of the number of isometric\_points. Furthermore the increment for the final ramp back is calculated.

        if (isometric\_hold==1)

{

isometric\_counter++;

// Test for breaking into ramp back

if (isometric\_counter>=isometric\_points)

{

isometric\_hold=0;

ramp\_back=1;

double\_increment = -double(in\_fl\_value) / double(ramp\_points);

double\_holder = last\_out;

}

out\_value = last\_out;

}

**Part 11: Ramp\_back**

During this part, the length of the trabec will ramp to a predefined level, for the duration of ramp\_points. After the ramp, the length will be set back to zero.

        if (ramp\_back==1)

{

ramp\_counter++;

double\_holder = double\_holder + double\_increment;

out\_value = short int (double\_holder);

if (ramp\_counter>=ramp\_points)

{

ramp\_back=0;

out\_value=0;

}

}

**Part 12: Emergency Break**

In case the system has not yet moved the out\_value back to zero by the end of the measurement, this statement moves it back to zero.

        // Emergency - force motor back to default for end of record

if (sample\_counter>max\_samples)

{

out\_value = 0;

}

**Part 13: Update the Buffers**

Update both buffers, last\_out\_buffer and out\_diff\_accumulator. Last\_out\_buffer holds the last "smoothing\_points" number of last\_out values. Out\_diff\_accumulator holds the sum of the last "smoothing\_points" number of last\_out values.

// Update last\_out\_buffer

for (i=0;i<(smoothing\_points-1);i++)

{

last\_out\_buffer[i] = last\_out\_buffer[i+1];

}

last\_out\_buffer[smoothing\_points-1] = last\_out;

// Calculate the out\_diff\_accumulator

out\_diff\_accumulator=0;

for (i=0;i<(smoothing\_points-1);i++)

{

out\_diff\_accumulator = out\_diff\_accumulator +

(last\_out\_buffer[i+1]-last\_out\_buffer[i]);

}

**Part 14: Update the last\_out value for the next loop**

// Hold value for next time

last\_out = out\_value;

**Part 15: Send out\_value to the output pipe**

// Send to output

pipe\_put(out\_pipe,out\_value);

**Part 16: Free Buffers**

If the buffers are not freed at the end of the loop, the space assigned for the buffers will clutter up and after a while the pc will get very slow.

// Tidy up

free(last\_out\_buffer);

free(error\_buffer);

**Part 17: Isotwitch \_Plcontrol C++ script**

// MODULE ISOTWITCHPICONT

//

// isotwitch (in\_pipe, iso\_level, n\_samples, iso\_points, iso\_gain, out\_pipe)

//  When in\_pipe rises above iso\_level, out\_pipe feeds back to keep in\_pipe

// at iso\_level, until out\_pipe starts to change direction. It then holds

// out\_pipe constant for iso\_points. The total number of samples is n\_samples

//

// Module name ISOTWITCH must be distinct from the DAPL command name

// assigned below.

#define  COMMAND  "ISOTWITCHPICONT"

#define  ENTRY    ISOTWITCHPICONT\_entry

#include "C:\Program Files\Microstar Laboratories\DTD\Include\DTDMOD.H"

#include "C:\Program Files\Microstar Laboratories\DTD\Include\DTD.H"

int \_\_stdcall     ENTRY (PIB \*\*plib);

extern "C" \_\_declspec(dllexport) int \_\_stdcall

   ModuleInstall(void \*hModule)

{   return (CommandInstall(hModule, COMMAND, ENTRY, NULL)) ; }

// - - - - -  command implementation section  - - - - - - - - - -

int \_\_stdcall  ENTRY (PIB \*\*plib)

{

// Storage for parameters

void \*\*argv;

int argc;

PIPE \* in\_force\_pipe;

PIPE \* in\_fl\_pipe;

long int pre\_trigger\_samples;

short int arming\_level; //Does nothing anymore?

short int isotonic\_level;

short int isotonic\_increment;

long int max\_samples;

long int isometric\_points;

long int ramp\_points;

long int inter\_ramp\_points; //added

long int inter\_fl\_diff; //added

short int prop\_gain;

short int Ki; //added

short int fl\_polarity;

short int fl\_switch;

short int fl\_threshold;

short int smoothing\_points;

short int integral\_points; //added

short int gain\_asymmetry;

PIPE  \* out\_pipe;

// Access parameters

argv = param\_process (plib, &argc, 19, 19,

T\_PIPE\_W, T\_PIPE\_W,

T\_VAR\_L, T\_VAR\_W, T\_VAR\_W, T\_VAR\_L, T\_VAR\_L,

T\_VAR\_L, T\_VAR\_L, T\_VAR\_L, T\_VAR\_W, T\_VAR\_W,

T\_VAR\_W, T\_VAR\_W, T\_VAR\_W, T\_VAR\_W, T\_VAR\_W,

T\_VAR\_W, T\_PIPE\_W);

in\_force\_pipe = (PIPE \*) argv[1];

in\_fl\_pipe = (PIPE \*) argv[2];

pre\_trigger\_samples = \*(long int volatile \*) argv[3];

isotonic\_level = \*(short int volatile \*) argv[4];

isotonic\_increment = \*(short int volatile \*) argv[5];

max\_samples = \*(long int volatile \*) argv[6];

isometric\_points = \*(long int volatile \*) argv[7];

ramp\_points = \*(long int volatile \*) argv[8];

inter\_ramp\_points = \*(long int volatile \*) argv[9];

inter\_fl\_diff = \*(long int volatile \*) argv[10];

prop\_gain = \*(short int volatile \*) argv[11];

Ki = \*(short int volatile \*) argv[12];

fl\_polarity = \*(short int volatile \*) argv[13];

fl\_switch = \*(short int volatile \*) argv[14];

fl\_threshold = \*(short int volatile \*) argv[15];

smoothing\_points =\*(short int volatile \*) argv[16];

integral\_points =\*(short int volatile \*) argv[17];

gain\_asymmetry =\*(short int volatile \*) argv[18];

out\_pipe= (PIPE \*) argv[19];

// Additional variables

int i;

short int in\_force\_value;

short int in\_fl\_value;

short int out\_value;

short int last\_out;

short int error\_value;

short int ramp\_increment;

short int out\_diff\_accumulator;

short int prop\_gain2;

short int Ki2;

int armed=0;

int was\_low;

int isotonic\_hold;

int isometric\_increment;

int isometric\_hold;

int ramp\_back;

int inter\_ramp; //added

int run\_started;

long int sample\_counter;

long int isometric\_counter;

long int inter\_ramp\_counter; //added

long int ramp\_counter;

long int ramp\_n;

short int \* force\_buffer;

short int \* last\_out\_buffer;

short int \* error\_buffer;

double double\_increment;

double double\_inter\_increment; //added

double double\_holder;

double double\_inter\_holder; //added

double double\_level;

// Perform initializations

pipe\_open(in\_force\_pipe, P\_READ);

pipe\_open(in\_fl\_pipe, P\_READ);

pipe\_open(out\_pipe, P\_WRITE);

armed=0;

was\_low=0;

isotonic\_hold=0;

isometric\_hold=0;

ramp\_back=0;

inter\_ramp=0; //added

run\_started=0;

sample\_counter=0;

inter\_ramp\_counter=0; //added

isometric\_counter=0;

ramp\_counter=0;

double\_inter\_increment = double(inter\_fl\_diff) / double(inter\_ramp\_points);

last\_out = 0;

out\_diff\_accumulator = 0;

double\_level = (double)isotonic\_level;

// Create space for the out and error buffers

last\_out\_buffer = (short int \*) malloc(smoothing\_points \* sizeof(short int));

error\_buffer = (short int\* ) malloc(integral\_points \* sizeof(short int));

// Zero buffers

for (i=0;i<smoothing\_points;i++)

{

last\_out\_buffer[i]=(short int)0;

}

for (i=0;i<integral\_points;i++)

{

error\_buffer[i]=(short int)0;

}

int temp\_counter=0;

long int error\_holder;

// Begin continuous processing

while (1)

{

// Increment counter

sample\_counter++;

// Read input values

in\_force\_value = (short int) pipe\_get(in\_force\_pipe);

in\_fl\_value = (short int) pipe\_get(in\_fl\_pipe);

// Check if we are armed - that is, have collected enough samples

if ((armed==0)&&(sample\_counter>pre\_trigger\_samples))

armed=1;

// If we are armed, check we were below the isotonic level

if ((armed==1)&&(in\_force\_value<isotonic\_level))

was\_low=1;

// If we were low, and we are far enough into the record, have we got above the target?

// The run started variable prevents later re-triggering

if ((was\_low==1)&&(in\_force\_value>isotonic\_level)&&(run\_started==0))

{

run\_started=1;

isotonic\_hold=1;

}

// Now deduce behavior

out\_value=0;

if (isotonic\_hold==1)

{

// Calculate force error

error\_value = (in\_force\_value - (short int)double\_level);

// Update error\_buffer

for (i=0;i<(integral\_points-1);i++)

{

error\_buffer[i] = error\_buffer[i+1];

}

error\_buffer[integral\_points-1] = error\_value;

// Sum the error\_buffer

error\_holder=0;

for (i=0;i<integral\_points;i++)

{

error\_holder = error\_holder + (long int)error\_buffer[i];

}

// Calculate changes in gain

if (out\_diff\_accumulator<0)

{

Ki2 = Ki + gain\_asymmetry;

}

else

{

Ki2 = Ki;

}

//Calculates out\_value with PIDcontrol, 100k is a constant that affrcts the gain params.

out\_value = last\_out -

(short int)

((long int)fl\_polarity \*

(((long int)prop\_gain \* (long int)error\_value) +

((long int)Ki2 \* (error\_holder / (long int)integral\_points)))/10000);

// Test for breaking out of isotonic hold and into isometric hold

if (fl\_switch==0)

{

// Code breaks out of control loop if out\_diff\_accumulator

// exceeds fl\_threshold. Use this to stop isotonic control

// when the motor reverses direction

// Need to check for both fl\_polarity options

if (((fl\_polarity>0)&&(out\_diff\_accumulator>fl\_threshold)) ||

((fl\_polarity<0)&&(out\_diff\_accumulator<-fl\_threshold)))

{

isotonic\_hold=0;

inter\_ramp=1;

}

}

else

{

// Code breaks out of control loop if motor returns to original value

if (((fl\_polarity>0)&&(out\_value>fl\_threshold)) ||

((fl\_polarity<0)&&(out\_value<-fl\_threshold)))

{

isotonic\_hold=0;

inter\_ramp=1;

}

}

// Update isotonic\_level for next iteration by adding increment

double\_level = double\_level + (double)((double)isotonic\_increment/1000.0);

double\_inter\_holder = last\_out;

}

if (inter\_ramp==1)

{

inter\_ramp\_counter++;

double\_inter\_holder = double\_inter\_holder + double\_inter\_increment;

out\_value = short int (double\_inter\_holder);

if (inter\_ramp\_counter>=inter\_ramp\_points)

{

inter\_ramp=0;

isometric\_hold=1;

// out\_value=0;

}

}

if (isometric\_hold==1)

{

isometric\_counter++;

// Test for breaking into ramp back

if (isometric\_counter>=isometric\_points)

{

isometric\_hold=0;

ramp\_back=1;

double\_increment = -double(in\_fl\_value) / double(ramp\_points);

double\_holder = last\_out;

}

out\_value = last\_out;

}

if (ramp\_back==1)

{

ramp\_counter++;

double\_holder = double\_holder + double\_increment;

out\_value = short int (double\_holder);

if (ramp\_counter>=ramp\_points)

{

ramp\_back=0;

out\_value=0;

}

}

// Emergency - force motor back to default for end of record

if (sample\_counter>max\_samples)

{

out\_value = 0;

}

// Update last\_out\_buffer

for (i=0;i<(smoothing\_points-1);i++)

{

last\_out\_buffer[i] = last\_out\_buffer[i+1];

}

last\_out\_buffer[smoothing\_points-1] = last\_out;

// Calculate the out\_diff\_accumulator

out\_diff\_accumulator=0;

for (i=0;i<(smoothing\_points-1);i++)

{

out\_diff\_accumulator = out\_diff\_accumulator +

(last\_out\_buffer[i+1]-last\_out\_buffer[i]);

}

// Hold value for next time

last\_out = out\_value;

// Send to output

pipe\_put(out\_pipe,out\_value);

}

// Tidy up

free(last\_out\_buffer);

free(error\_buffer);

return 0;

}

// End of TRIGMOVEM module