**CalcsCt.C**

```
// File deals with the calculations acossiated with the design of the
controllers
#include <stdio.h>  // Including stdio.h header file
#include <math.h>   // Including math.h header file
#include <tgmath.h> // Including tgmath.h header file

#include "SystemOuts.C"    // Includes SystemOuts.C file
#include "SecondOrderIn.C" // Includes SecondOrderIn.C file
#include "FirstOrder.C"    // Includes FirstOrder.C file

void PIDLLcalcs1st(struct transfer_function_first_order *t1, struct
parameters *p2, struct controllers *c2, struct systemOuts *s2, int i);
// Decalers the calculation function for second order systems
float max(float x, float y);
// Declares the max function to find the max value
void PIDLLcalcs2nd(struct transfer_function_second_order *√•t2, struct
parameters *p2, struct controllers *c2, struct systemOuts *s2, int i); //
Declares the calculation function for first order systems

struct systemOuts // Deining the systemOuts structure
{
    float damping;     // member for the damping ratio
    float naturalFreq; // Natural frequency value
    struct Pid         // Pid structure nested within systemOuts
    {
        float kp;  // Proportional gain value
        float kin; // Integral's numerator (zero) value
        float kid; // Integral's denominator (pole) value
        float kd;  // Derivative's (zero) value
    } a;           // Variable for access
    struct PI      // PI structure nested within systemOuts
    {
        float kp;  // Proportional gain value
        float kin; // Integral's numerator (zero) value
        float kid; // Integral's denominator (pole) value
    } b;           // Variable name for acces
    struct PD      // PD structure nested within systemOuts
    {
        float kp; // Proportional gain value
        float Kd; // Derivative's (zero) value
    } c;          // Variable name for acces
    struct Lead   // Lead structure nested within systemOuts
    {
        float K;  // Proportional gain value
        float ra; // Zero value
        float r;  // Pole value
    } d;          // Variable name for access
    struct Lag    // Lag structure nested within systemOuts
    {
        float K;  // Proportional gain value
        float ra; // Pole value
        float r;  // Zero value
    } e;          // Variable name for access
    int order;    // Variabloe for the order of the system
};
```

```
void PIDLLcalcs2nd(struct transfer_function_second_order *t2, struct
parameters *p2, struct controllers *c2, struct systemOuts *s2, int i) //
defining the 2nnd order system calcs func
{

    float a = t2[i].a;                     // setting vaiable a to the
numerator
    float b = t2[i].b;                     // setting variable b to the
first s term
    float c = t2[i].c;                     // setting variable c to the
first pole
    float d = t2[i].d;                     // setting variable d to the
second s term
    float e = t2[i].e;                     // setting varable e to the
second pole
    float st = p2[i].settling_time;        // setting the settling time
to the variable st
    float po = p2[i].percentage_overshoot; // setting the percentage
overshoot to the variable po
    float gain = (p2[i]).ss_gain;          // setting the steady state
value to the variable gain

    float openP1 = (c / b) * -1;
// Calculates the open loop pole one value by multiplying the first pole
by -1
    float openP2 = (e / d) * -1;
// Calculates the open loop pole two value by multiplying the second pole
by -1
    printf("You have the following open loop poles: %f & %f", openP1,
openP2);           // tells the user what poles they have
    float damp = sqrt(1 / (1 + ((3.142 / log(po)) * (3.142 / log(po)))));
// Finds the damping ratio of the system using the parameter's percent
overshoot value
    s2[i].damping = damp;
// stores the damping ratio of the system into the stuctures value
    float wn = 4 / (damp * st);
// Finds the natural frequency of the system into the stuctures value
    s2[i].naturalFreq = wn;
// Stores the natural frequency in the controller structure
    float real = -wn;
// Calculates the real part of the desired poles
    float imag = ((wn)*sqrt(1 - damp * damp));
// Calculates the imaginary part of the desired poles
    printf("\nYour desired poles are: %f + %fi and %f - %fi\n", real,
imag, real, imag); // Displays the desired poles to the user
    float arg1 = 180 - atan(imag / (real + openP1));
// Calculates the first argument value using the desired pole one's real
and imaginary parts, and the first pole from the transfer function
    float arg2 = 180 - atan(imag / (real + openP2));
// Calculates the second argument value using the desired pole one's real
and imaginary parts, and the second pole from the transfer function
    float deriv = ((imag / (-180 - arg1 - arg2)) + real);
// Calculates the derivative's zero value using the desired pole one's
real and imaginary parts, and the first and second arguments
    float l1 = sqrt(((real + openP1) * (real + openP1)) + ((imag) *
(imag)));           // Calculates l1, which is the distance from pole
one to desired pole one
```

```
    float l2 = sqrt(((real + openP2) * (real + openP2)) + ((imag) *
(imag)));            // Calculates l2, which is the distance from pole
two to desired ploe two
    float l3 = sqrt(((deriv - real) * (deriv - real)) + ((imag) *
(imag)));              // Calculates l3, which is the distance from the
derivative's zero to desired pole one
    float prop = ((l1 * l2 * gain) / (l3)) / a;
// Calculates the total K value, whichis the value of the proportional
gain
    float lr = max(l1, l2) * -1;
// Calculates the length of r to the desired poles
    if ((c2[i]).PID == 1)
    {
        s2[i].a.kp = prop;  // sets k value
        s2[i].a.kin = 0.1;  // sets intgeral's numerator value
        s2[i].a.kid = 1;    // sets integral denominator value
        s2[i].a.kd = deriv; // sets derivative value
    };
    if ((c2[i]).PI == 1)
    {
        s2[i].b.kp = prop; // sets k value
        s2[i].b.kin = 0.1; // sets intgeral's numerator value
        s2[i].b.kid = 1;   // sets integral denominator value
    };
    if ((c2[i]).PD == 1)
    {
        s2[i].c.kp = prop;  // sets k value
        s2[i].c.Kd = deriv; // sets derivative value
    };
    if ((c2[i]).Lead == 1) // Runs if a phase lead compensator is
selected
    {
        float ra = (imag / tan(180 - arg1)) + real;                    //
calculateds the ra value
        float lra = sqrt(((real - ra) * (real - ra)) + (imag * imag)); //
calculates the length from ra to the desired poles
        s2[i].d.ra = ra;                                               //
sets ra value
        s2[i].d.r = max(openP1, openP2);                               //
sets r value
        s2[i].d.K = ((l1 * l2 * lra) / (lr)) * -1;                     //
sets k value
    };
    if ((c2[i]).Lag == 1) // Runs if a phase lag compensator is selected
    {
        float ra = (imag / tan(-180 - arg1)) + real;                   //
calculateds the ra value
        float lra = sqrt(((real - ra) * (real - ra)) + (imag * imag)); //
calculates the length from ra to the desired poles
        s2[i].e.ra = ra;                                               //
sets ra value
        s2[i].e.r = max(openP1, openP2);                               //
sets r value
        s2[i].e.K = ((l1 * l2 * lra) / (lr)) * -1;                     //
sets k value
    };

}; // End of function definition for calculating 2nd order controllers
```

```
void PIDLLcalcs1st(struct transfer_function_first_order *t1, struct
parameters *p2, struct controllers *c2, struct systemOuts *s2, int i) //
defining the 1st order system calcs func
{

    float a = t1[i].a;                      // setting vaiable a to the
numerator
    float b = t1[i].b;                      // setting variable b to the s
term
    float c = t1[i].c;                      // setting variable c to the
pole
    float st = p2[i].settling_time;         // setting the settling time
to the variable st
    float po = p2[i].percentage_overshoot; // setting the percentage
overshoot to the variable po
    float gain = (p2[i]).ss_gain;           // setting the steady state
value to the variable gain

    float openPole = (c / b) * -1;
// Calculates the open loop pole one value by multiplying the first pole
by -1
    printf("You have the following open loop pole: %f\n", openPole);
// tells the user what pole they have
    float damp = sqrt(1 / (1 + ((3.142 / log(po)) * (3.142 / log(po)))));
// Finds the damping ratio of the system using the parameter's percent
overshoot value
    s2[i].damping = damp;
// sets the damping ratio of the system to the structure's value
    float wn = 4 / (damp * st);
// Finds the natural frequency of the system using the parameter's
settling time value
    s2[i].naturalFreq = wn;
// Stores the natural frequency in the controller structure
    float real = -wn;
// Calculates the real part of the desired poles
    float imag = ((wn)*sqrt(1 - damp * damp));
// Calculates the imaginary part of the desired poles
    float arg1 = 180 - atan(imag / (real - c));
// Calculates the first argument value using the desired pole one's real
and imaginary parts, and the first pole from the transfer function
    float deriv = ((imag / (-180 - arg1)) + real);
// Calculates the derivative's zero value using the desired pole one's
real and imaginary parts, and the first and second arguments
    float l1 = sqrt(((real - c) * (real - c)) + ((imag) * (imag)));
// Calculates l1, which is the distance from pole one to desired pole one
    float l3 = sqrt(((deriv - real) * (deriv - real)) + ((imag) *
(imag))); // Calculates l3, which is the distance from the derivative's
zero to desired pole one
    float prop = ((l1 * gain) / (l3)) / a;
// Calculates the total K value, which is the value of the proportional
gain
    if ((c2[i]).PID == 1)
// Runs if a PID controller is selected
    {
        s2[i].a.kp = prop;  // sets K value
        s2[i].a.kin = 0.1;  // sets intgeral's numerator value
        s2[i].a.kid = 1;    // sets integral denominator value
```

```
        s2[i].a.kd = deriv; // sets derivative value
    };
    if ((c2[i]).PI == 1) // Runs if a PI controller is selected
    {
        s2[i].b.kp = prop; // sets K value
        s2[i].b.kin = 0.1; // sets intgeral's numerator value
        s2[i].b.kid = 1;   // sets integral denominator value
    };
    if ((c2[i]).PD == 1) // Runs if a PD controller is selected
    {
        s2[i].c.kp = prop;  // sets K value
        s2[i].c.Kd = deriv; // sets derivative value
    };
    if ((c2[i]).Lead == 1) // Runs if a phase lead compensator is 
selected
    {
        float ra = (imag / tan(180 - arg1)) + real;                    // 
calculateds the ra value
        float lra = sqrt(((real - ra) * (real - ra)) + (imag * imag)); // 
calculates the length from ra to the desired poles
        s2[i].d.ra = ra;                                               // 
sets ra value
        s2[i].d.r = (c / b);                                           // 
sets r value
        s2[i].d.K = ((l1 * lra) / (l3 * l1)) * -1;                     // 
sets k value
    };
    if ((c2[i]).Lag == 1)
    {
        float ra = (imag / tan(-180 - arg1)) + real;                   // 
calculateds the ra value
        float lra = sqrt(((real - ra) * (real - ra)) + (imag * imag)); // 
calculates the length from ra to the desired poles
        s2[i].e.ra = ra;                                               // 
sets ra value
        s2[i].e.r = (c / b);                                           // 
sets r value
        s2[i].e.K = ((l1 * lra) / (l3 * l1)) * -1;                     // 
sets k value
    };

}; // End of function definition for calculating 2nd order controllers

float max(float x, float y) // defining the max function
{
    if (x >= y) // runs if x >= y
    {
        return x; // returns x
    }
    else // runs if y > x
    {
        return y; // returns y
    }
};
```