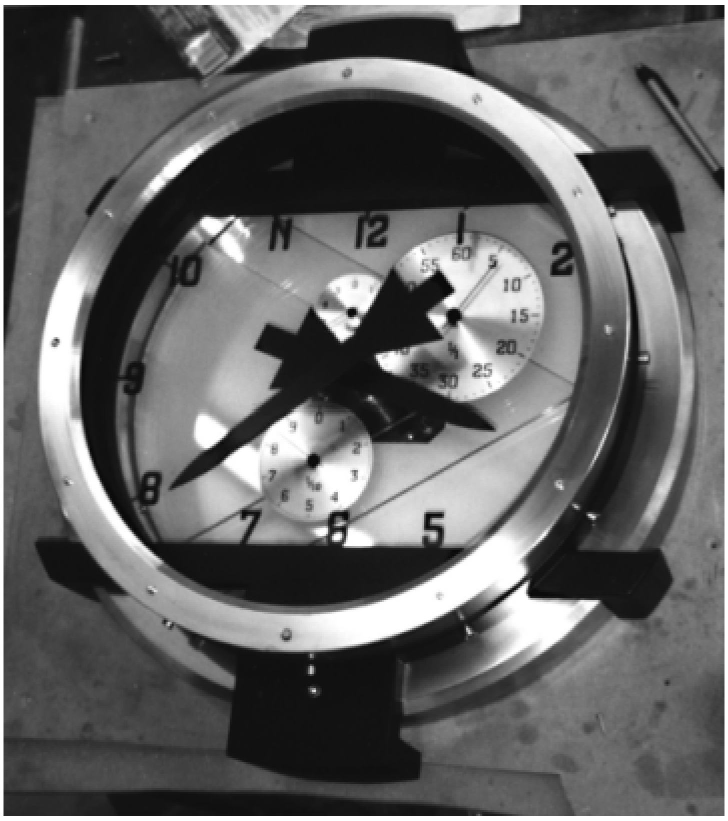
Basic DC Motor Speed PID Control With The Infineon C167 Family

We do get asked for some strange things sometimes: Vauxhall Vectra fans may recall the original launch TV advert which for a few seconds featured on-screen, a large multi-dialled clock, which was supposed to show time speeding forward to catch up with the leap into the future made by the new Vectra. Others might have suggested it was simply counting down the hours towards a cambelt failure.... click here to view the advert on youtube

The clock was a stage prop designed and built for the advert by a London model-making company. It now resides in a North London flat as a rather unusual coffee table. Unfortunately the expensive variable speed-regulated motors used to drive the three dials on the clock face were reclaimed by the production company and so despite the complex gearing system installed, it did not run.



A rather odd telephone call from the owner revealed that he was looking for some way to get it going again at minimum cost. He had seen an advert in a hobby magazine for C167 starter kits and wondered if there were some examples around of how to control the speed of a DC motor. Being helpful types and major fans of the Vectra (no chance), we came up with a solution based on a recycled Phytec miniMODULE167, a 12v DC motor, a fan, an infra-red LED, a photodiode and some simple C code.

The objective was to make the clock run in "real time", accurate enough to keep good time for the duration of the average dinner party but be able to run at high speed (as in the advert) on demand to impress the guests, just after the traditional serving of peppermint Rennies.

The result was a simple Proportional-Integral-Derivative (PID) controller for a 12v permanent magnet DC motor. PID is very widely used in industrial control systems and something we get asked for examples of very frequently. Strangely, a trawl of the Web revealed no C-coded examples of any sort so we decided to do it from scratch.

To make the clock run at a constant speed, here 600rpm, some form of accurate speed regulator mechanism was required. This would ensure that over time, the average motor speed would be constant. The nature of the clock mechanism was that the load on the motor varies. For example, as the various hands move, small load peaks occur which tend to disturb the running speed. The drag and motor efficiency were also subject to change, particularly as a result of temperature. A more appropriate motor drive mechanism to have used in this type of application would have been a stepper motor but most requests we get are for the control of conventional motors plus a reasonable DC motor just happened to be in the parts bin at the time....

The important elements of the speed controller are:

- A means of altering the voltage applied to the motor to control its speed
- A means of measuring the absolute speed of the motor shaft
- Some software running on a C167 to keep the speed constant, despite rapid variations in load caused by the clock gearing plus slow changes in drag due to ambient temperature

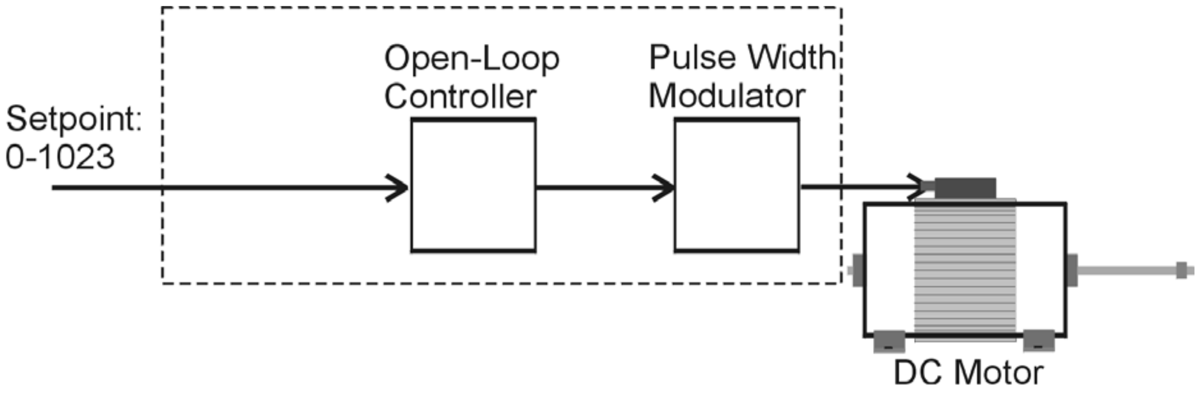
The PID controller software running on the C167 requires the following:

- An input, here a potentiometer, to set the absolute speed at the which the motor should run
- An interrupt to measure the motor speed
- A periodic interrupt to works out the difference between the actual measured speed and the required speed
- Some way of converting the speed error into a variable duty-ratio pulse width modulation (PWM) to give a controllable average voltage at the motor

A bit of control theory

If the motor runs at 600rpm when the PWM drive to the motor is at a 50% duty ratio, increasing this to 60% will make the motor try to run faster. Reducing the duty ratio to 40% will slow the motor down. In a very simple "open loop" speed controller, the program, the potentiometer on analog channel 0 is read to yield a value between 0-1023 (10-bits). This value is then fed into the PWM unit to allow the motor speed to be varied.

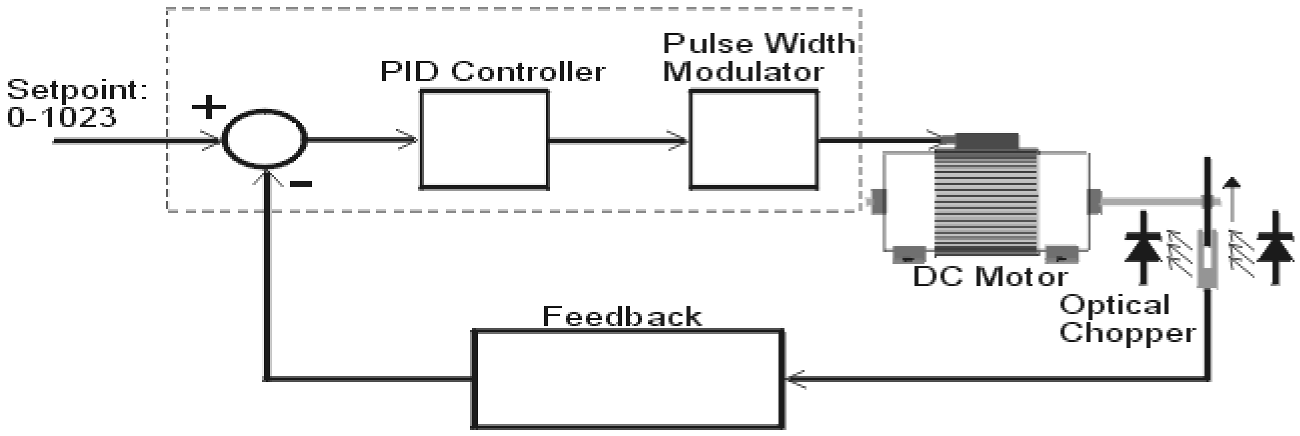
In the real world, this type of controller is not very useful. While it allows the motor speed to be set, it does not allow for changes in load and a basic flaw is that the absolute speed is not known unless an external tachometer is used.



Closing The Loop

The closed-loop controller is a very common means of keeping motor speed at the required "setpoint" under varying load conditions. It is also able to keep the speed at the setpoint value where for example, the setpoint is ramping up or down at a defined rate.
The essential addition to the previous system is a means for the current speed to be measured. In the example, a three bladed vane was attached to the motor shaft. An infra-red LED was obscured from the view of a photodiode by the vane blades so that a series of pulses with a frequency proportional to motor speed is now available.

In this "closed loop" speed controller, a signal proportional to the motor speed is fed back into the input where it is subtracted from the setpoint to produce an error signal. This error signal is then used to work out what the magnitude of controller output should be to make the motor run at the required setpoint speed. For example, if the error speed is positive, the motor is running too fast so that the controller output should be reduced and vice-versa. The clever part is how the output drive is worked out....



At first sight it might be imagined that something simple like "if the error speed is negative, multiply it by some scale factor (usually known as "gain") and set the output drive to this level", i.e. the voltage applied to the motor is proportional to the error speed. In practice, this approach is only partially successful for the following reason: if the motor is at the setpoint speed under no load there is no error speed so the motor free runs. If a load is applied, the motor slows down so that a positive error speed is produced. The output increases by a proportional amount to try and restore the speed. However, as the motor speed recovers, the error reduces and so therefore does the drive level. The result is that the motor speed will stabilise at some speed below the setpoint at which the load is balanced by the error speed x the gain. If the gain is very high so that even the smallest change in motor speed causes a significant change in drive level, the motor speed may oscillate or "hunt" slightly . This basic strategy is known as "proportional control" and on its own has only limited use as it can never force the motor to run exactly at the setpoint speed.

The next improvement is to introduce a correction to the output which will keep adding or subtracting a small amount to the output until the motor reaches the setpoint, at which point no further changes are made. In fact a similar effect can be had by keeping a running total of the error speed speeds observed for instance, every 25ms and multiplying this by another gain before adding the result the proportional correction found above. This new term is based on what is effectively the integral of the error speed.

Thus far we have a scheme where there are two mechanisms trying to correct the motor speed which constitutes a PI (proportional-integral) controller. The proportional term is a fast-acting correction which will make a change in the output as quickly as the error arises. The integral takes a finite time to act but has the ability to remove all the steady-state speed error.

A further refinement uses the rate of change of error speed to apply an additional correction to the output drive. This means that a rapid motor deceleration would be counteracted by an increase in drive level for as long as the fall in speed continues. This final component is the "derivative" term and it is a useful means of increasing the short-term stability of the motor speed. A controller incorporating all three strategies is the well-known Proportional-Integral-Derivative, or "PID" controller.

For best performance, the proportional and integral gains need careful tuning. For example, too much integral gain and the control will tend to over-correct for any speed error resulting in oscillation about the setpoint speed. Several well-known mathematical techniques are available to calculate optimal gain values, given knowledge of the combined characteristics of the motor and load, i.e. the "transfer function". However, some simple rules of thumb and a little experimentation can yield satisfactory results in practical applications.

What The Gains Do

Integral Gain: Ensures that under steady state conditions that the motor speed (almost) exactly matches the setpoint speed. A low gain can make the controller slow to push the speed to the setpoint but excessive gain can cause hunting around the setpoint speed. In less extreme cases, it can cause overshoot whereby the speed passes through the setpoint and then approaches the required speed from the opposite direction. Unfortunately, sufficient gain to quickly achieve the setpoint speed can cause overshoot and even oscillation but the other terms can be used to damp this out.

Proportional Gain: Gives fast response to sudden load changes and can reduce instability caused by high integral gain. This gain is typically many times higher than the integral gain so that relatively small deviations in speed are corrected while the integral gain slowly moves the speed to the sepoint. Like integral gain, when set too high, proportional gain can cause a "hard" oscillation of a few Hertz in motor speed.

Derivative Gain: Can be used to give a very fast response to sudden changes in motor speed. Within simple PID controllers it can be difficult to generate a derivative term in the output that has any significant effect on motor speed. It can be deployed to reduce the rapid speed oscillation caused by high proportional gain. However, in many controllers, it is not used.

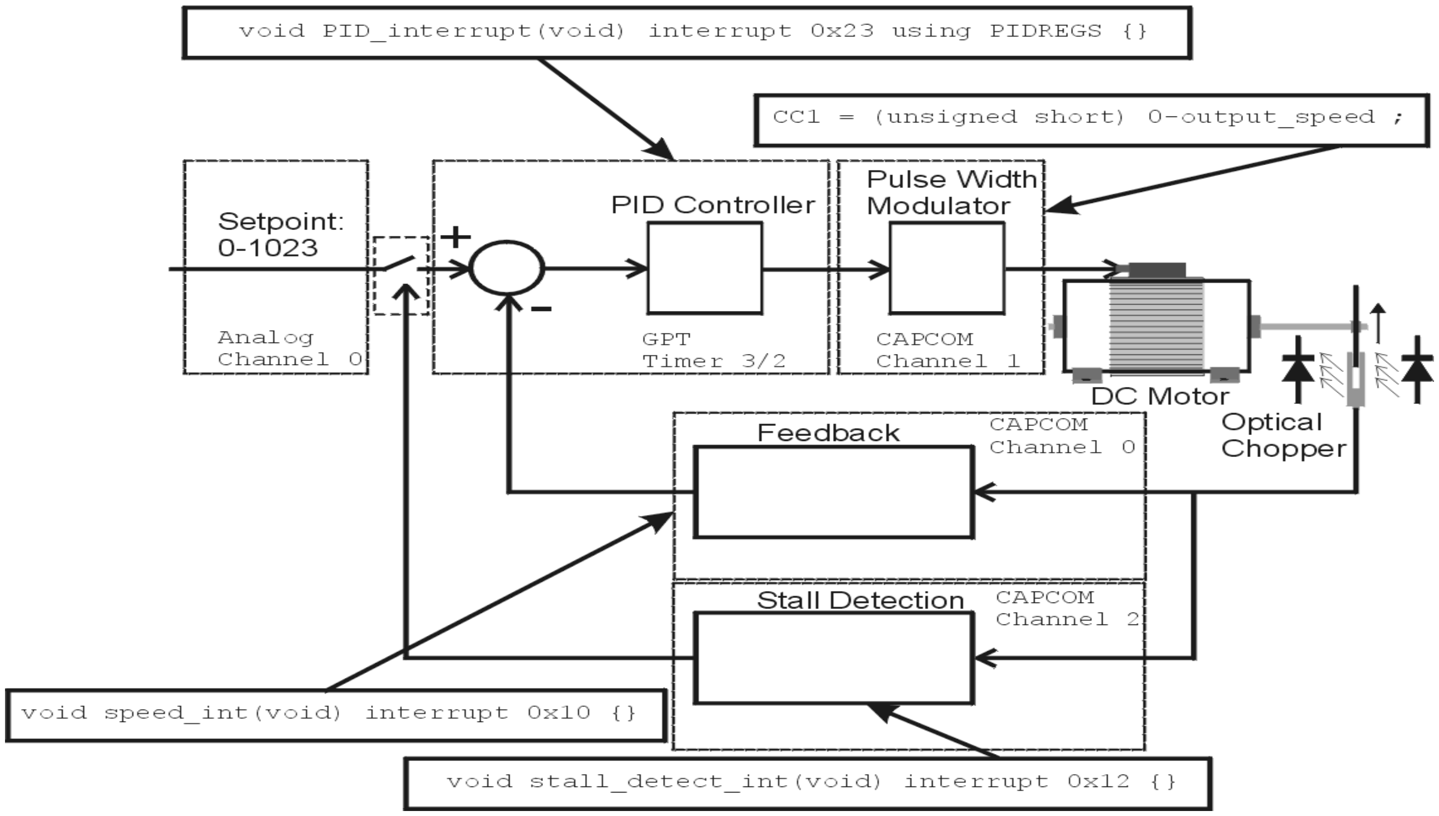
Practical Implementation Of A Simple PID DC Motor Speed Controller

In the original example, a SK167 C167 starter kit was chosen to implement the PID controller in software. The C167 is way too powerful for such a task but this excess of number crunching power makes the software easier to write and debug!

Basic Considerations

The main PID controller routine was designed to be fairly general purpose and hence modular. Whilst here it is used to control a DC motor, it could be re-deployed to other situations where some parameter has to be controlled to a set value under varying conditions. The actual control software is located in a single function and its major inputs and output are held in a structure. Although it was designed originally for a specific job it is really only intended as an example of the basic techniques involved and to allow those with no control system knowledge to experiment with a simple PID system.

The routines that gather the inputs and process the output are kept in separate functions in another module. The Keil C166 v4.01 compiler was used but it should not prove too difficult to port it to the rather less common Tasking compiler, or indeed the 8-bit C505 or C515.


void PID_interrupt(void) interrupt 0x23 using PIDREGS {}
CC1 = (unsigned short) 0-output_speed ;
Setpoint: 0-1023
Analog Channel 0
PID Controller
+
-
GPT Timer 3/2
Pulse Width Modulator
CAPCOM Channel 1
DC Motor
Optical Chopper
Feedback
CAPCOM Channel 0
Stall Detection
CAPCOM Channel 2
void speed_int(void) interrupt 0x10 {}
void stall_detect_int(void) interrupt 0x12 {}

For simplicity and to allow the easy modification of the important PID gain parameters, the C167's 10-bit analog to digital converter was used to derive 10-bit resolution inputs from simple trimmer potentiometers. The PWM used to drive the motor was chosen as 10-bits so that motor speed can be defined to approximately 0.1%, sufficient for most practical application. At this resolution, the C167 CAPCOM unit generates a 2.4kHz carrier which does produce some audiable motor winding ringing. The C167's high resolution dedicated PWM unit could have been used to increase this to a supersonic 19.5kHz but to allow easy porting to other C167 variants this route was not taken. Fortunately the use of a 10 bit resolution on the inputs and output makes some of the arithmetic easier!

The C167 IO pins are allocated as per: P2.0 CAPCOM channel 0 - optical chopper encoder input P2.1 CAPCOM channel 2 - PWM drive output P5.0 analog channel 0 - setpoint input P5.1 analog channel 1 - derivative gain input P5.2 analog channel 1 - proportional gain input P5.3 analog channel 1 - integral gain input

Designing The PID Controller Routine

The PID control problem has to be converted from a theoretical continuous process into a real "discrete" system running on a microcontroller. What this means in practice is that the measuring of the setpoint and motor speed and the calculation of the output is only performed a regular interval. In the context of a microcontroller, this might correspond to some code run from a timer interrupt.

The PID controller can thus be expressed as: Output = Proportional Gain * (error_speed) + Integral_gain * S (previous_error_speeds) + Derivative_gain * (error_speed - last_error_speed)

This is quite easy to implement as a C interrupt function. The timer 3 interrupt is easy to configure to produce a regular interrupt event, with timer 2 acting as a reload register. The period of the interrupt determines the effective sampling rate of the PID controller.

The service routine introduces the small refinement to the basic control strategy in the form of a "deadband". This stops the controller trying to constantly correct very small speed errors which in fact can trigger instability.

/*** Work out current error speed ***/ this_error = PID.setpoint - PID.feedback_input ; /*** Check for error speed being less than deadband width ***/ temp = this_error ; // Is error speed negative if(N) { temp = -temp ; // Get absolute value } /* Is error speed within deadband? */ if(temp <= Deadband_Width) { this_error = 0 ; // Within deadband so zero error speed }

A trick is used to quickly find the modulus of the potentially signed error speed which involves checking the C167's own N (negative) flag directly and if set, negate the result. For strict 'C' portability, this sort of technique is a bit suspect but could easily be replaced by the proper ISO-C abs() function.

The proportional term is very easily calculated but the complication with this and in fact all terms, is catering for overflows. The PID controller was written to be fairly general purpose and so was based on signed 16 bit arithmetic. Here, the maximum value is 32767 (0x7FFF) and the minimum -32768 (0x8000). Unfortunately there is no quick way of checking for the term exceeding this condition so a long compare had to be made to see whether the output value had gone out of limits. Under these circumstances, the value is simply set equal to the appropriate positive or negative limit. This limiting of a parameter to a maximum value is known as "saturation".

```
/*** Calculate Proportional Term ***/

proportional_term_temp = ((long)this_error * (long)proportional_gain) ;

/*** Check For Proportional Term Out Of Range & Apply Saturation ***/

if(proportional_term_temp > (long)((long)32767 * Inputs_Scale_Factor))
{
proportional_term = 32767 ;
}
else
{
if(proportional_term_temp <= (long)((long)-32768 * Inputs_Scale_Factor))
{
proportional_term = -32768 ;
}
else
{
proportional_term = (short) ((long)proportional_term_temp/Inputs_Scale_Factor) ;
}
}
```

The derivative term is simply calculated thus:

```
/*** Calculate Derivative Term ***/
derivative_term = ((long)(this_error - PID.last_error) *
derivative_gain)/(long)Inputs_Scale_Factor ;
```

Given that the maximum error difference is 1023 and the derivative gain can never exceed 32767, there is no possibility of an overflow from 16 bits in the result and so no saturation checks are made.

The integral term is based on the sum of all previous observed error speeds. The complication is that not only must the accumulated error be checked for overflow but so must the resulting integral term. This highlights one of the most tricky aspects of this type of software - overflows must not be left unchecked as unlike in for example PC programs, an unexpected overflow can cause serious mechanical damage to a real motor!

```
/*** Find Accumulated Error ***/
acc_error_temp = ((long)PID.accumulated_error) + (long)this_error ;

/*** Check For Accumulated Error Out Of Range ***/

if(acc_error_temp > (long)32767)
{

// Is error > maximum value?

acc_error_temp = 32767 ;

// Limit to max +ve value

}

if(acc_error_temp < (long)-32768)
{
// Is error < minimum value?

acc_error_temp = -32768 ;

// Limit to max -ve value
}

PID.accumulated_error = (short) acc_error_temp ;

/*** Calculate Integral Term ***/

integral_term_temp = ((long)PID.accumulated_error * (long)integral_gain) ;

/*** Check For Integral Term Out Of Range & Apply Saturation ***/

if(integral_term_temp > (long)((long)32767 * Inputs_Scale_Factor))
{
integral_term = 32767 ;
}
else
{
if(integral_term_temp <= (long)((long)-32768 * Inputs_Scale_Factor))
{ integral_term = -32768 ;
}
else
{
integral_term = integral_term_temp/Inputs_Scale_Factor ;
}
}
```

The final job is sum up all the terms and again check for overflow. The casting to long at each stage makes sure that overflow will not occur during the summation. Although the controller can accommodate a negative output, in the context of the motor controller for simplicity it was decided to simply convert a negative value to zero. In applications where the controlled plant can accept both negative and positive values, this could be made for a check for < -32768.

```
/*** Sum Up Control Terms ***/


control_output_temp = (long) integral_term ;
control_output_temp  += (long)derivative_term ;
control_output_temp += (long) proportional_term ;


/*  Limit Value Of Control Term */


if(control_output_temp > 32767)
{
   control_output_temp  = 32767 ;
}
else
{
   if(control_output_temp < 0)
   {
      control_output_temp = 0 ;
   }
}
```

Generating the analog voltage to control the motor

The C167 CAPCOM unit can be configured to generate pulsewidth modulation with virtually no software overhead. To allow a very precise control of the motor speed, a 10-bit PWM was chosen, yielding a theoretical <0.1% resolution. In this application this is probably overkill but in other situations it might prove useful. The integration of the waveform by the L-R of the motor windings effectively creates a DC-level proportional to the duty ratio.

As configured, timer 0 is allowed to run between 0xFC00 (-1024) and 0xFFFF (65535) by setting the T0REL register to 0xFC00. A value of 0xFC01 to 0xFFFF placed in CAPCOM channel 1 will cause pin 2.1 to go high when the timer 0 values matches it. At each overflow of timer 0, the pin is automatically cleared. This mechanism requires zero software intervention once configured and is easy to use.

The PID controller's output is in the range of 0-32767 and so has to be scaled to cover the 0 to -1023 range required by the PWM generator.

```
/*** Scale PID Controller Output To Suit PWM ***/

output_speed = ((long)PID.control_output * 1023)/32767 ;

// Scale 0-1023

/*** Update PWM ***/

CC1 = (unsigned short) 0-output_speed ; // Negate value as timer 1 only counts up
```

Measuring The Motor Speed

The output from the optical chopper is fed straight into CAPCOM channel 0 via a 4k7 resistor, with no buffering. The C167 has back-to-back diodes on its port pins which protect against excessive voltage. By using a series resistor, any excess voltage is removed and provided it is large enough to keep the current into the port pin below 4mA, no damage will occur.

CAPCOM channel 0 is used in negative edge-triggered mode as this is the best defined edge output by the optical chopper. It captures the value of free running timer 1 and in a small interrupt routine, calculates the time since the last edge.

```
motor_period = CC0 - last_CC0 ; // Calculate time since last edge

last_CC0 = CC0 ; // Carry forward current time to next interrupt
```

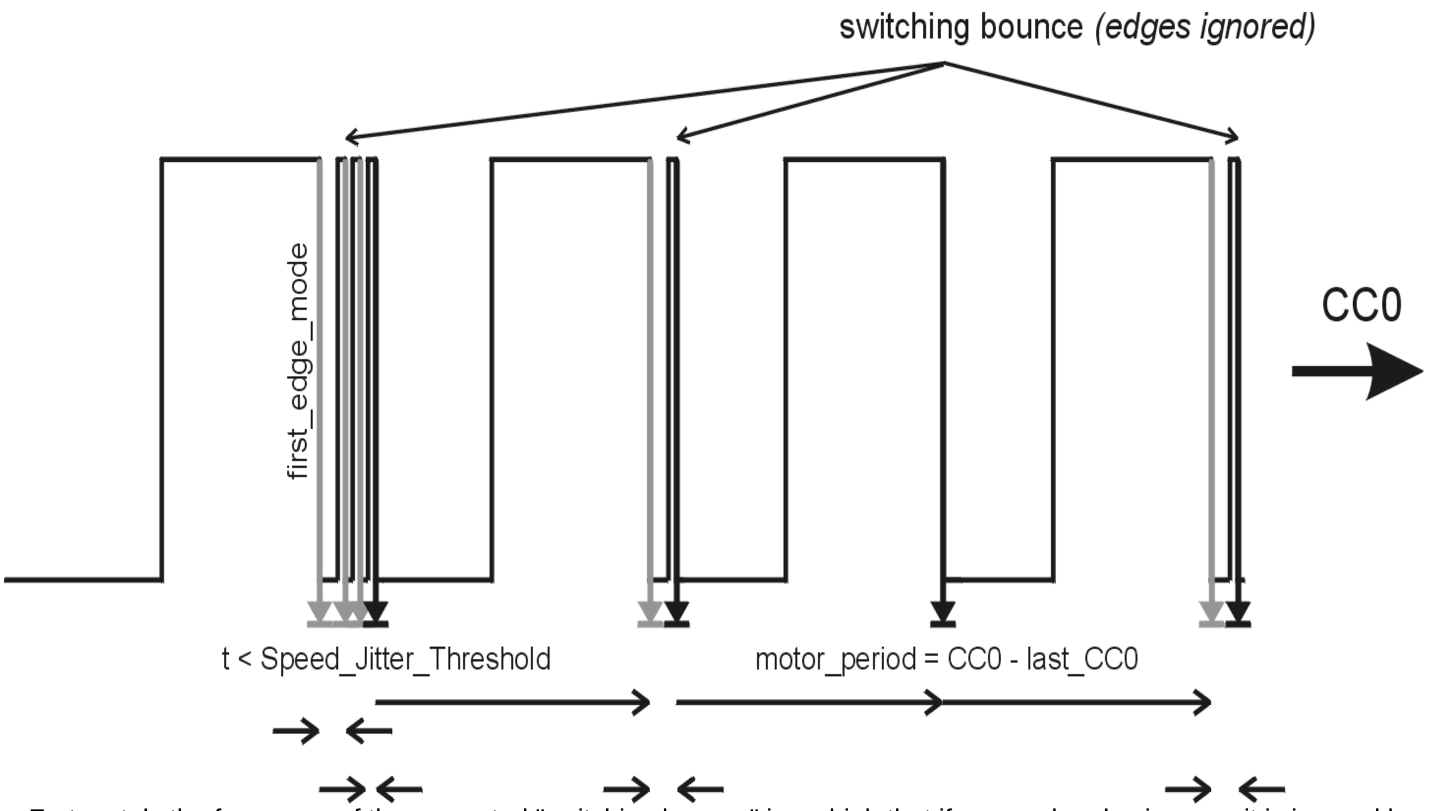
The first edge seen has to ignored as until at least one valid edge has to have been detected before an accurate period measurement can be made.

```
if(first_edge_mode)
{ first_edge_mode = Off ; // Period calculation next time...
}
else
{
motor_period_temp = CC0 - last_CC0 ;

// Calculate time since last edge

if(motor_period_temp > Speed_Jitter_Threshold) {
```

A very awkward real-world complication is that on the first edge as the motor starts to move, there is a random high frequency switching with pulses of around 30us width. This is due to the poor quality of the signal conditioning hardware in the photodiode circuit. In addition, it also produces an unwanted extra 30us pulse after the real falling edge.


switching bounce (edges ignored)
first_edge_mode
CC0
t < Speed_Jitter_Threshold
motor_period = CC0 - last_CC0

Fortunately the frequency of the unwanted "switching bounce" is so high that if any such pulse is seen, it is ignored by checking that the motor_period is greater than a threshold corresponding to around 100us. Some might suggest that the effort required to solve this problem might have been better spent designing proper hardware!

Stall detection

In any speed measurement like this, the problem always arises as to what to do when the motor stops. The longest period (i.e. lowest speed) that can be measured by the above means corresponds to 0xFFFF counts. Fortunately, the DC motor used was not able to run stably below about 2 revs/sec so this limitation was not a problem. However, some means of actually forcing the period to 0xFFFF is required as the last measured period is not defined as there is no "last edge" in cases where the motor stops dead. Therefore a special "stall detect" mechanism is required to cope with this.

Another CAPCOM channel (CC2) is used in a special mode where it simply generates an interrupt. On each optical chopper edge, CC2 is set equal to the current timer 1 count + 0xFFF0. If no further edges occur (i.e. the motor has stopped) the CC2 interrupt occurs and the motor period is forced to 0xFFFF. The first edge flag is also set so that the speed measurement will restart correctly.

Getting The Motor Speed From The Period Measurement

The PID controller expects a speed feedback value in the range 0-1023. The motor frequency is likely to be in a completely different range so some scaling is required. The scale factor is the motor period at the maximum possible motor speed divided by the 1023 scale factor.

```
#define Max_Speed_Count 670 /* Timer 1 counts between edges at max speed */
#define Motor_Speed_Scale_Factor (Max_Speed_Count*1023L)
```

The motor speed is found by dividing the scale factor by the measured motor period.

```
motor_speed = (unsigned long)Motor_Speed_Scale_Factor/motor_period_temp ;
PID.feedback_input = motor_speed ;
```

This results in the speed feedback being in the same units and dynamic range as the setpoint speed simplifying the PID routine's error _speed calculation.

Setting Up The Controller

This requires a bit of experimentation but a few simple rules can be applied.

PID interrupt period

This perhaps the most basic setting. The optimum period is a trade-off between speed of response and CPU loading. Bearing in mind that this job of this routine is to decide what corrective action to take and then apply it, there is no point running the interrupt too fast: if the PID controller makes corrections faster than the motor can react to them, instability is bound to result. In the example, it was found by experimentation that it took around 40ms for any change in PWM duty ratio to be reflected in a change in speed. Therefore the interrupt was set to run at this rate.

The PID_Sample_Period constant is set according to
(required interrupt period)/(timer 3 count rate).

In the example this is 0.025ms/0.8us = 31250.

Speed Constants

Max_Speed_Count is the number of timer 1 counts between optical chopper edges at the maximum possible motor speed. In the example this was 670 counts and timer 1 runs at 3.2us per bit, giving 670 * 3.2us = 2.144ms per edge.

Speed_Jitter_Threshold is the number of timer 1 counts which the erroneous jitter pulses correspond to. In practice this is simply Max_Speed_Count less a 20% safety margin that prevents motor overspeed being confused with jitter.

Output_Offset - unfortunately the motor used in the example did not start to move until at least 30% of the 100% PWM duty ratio is applied. To stop the integral term having to wind up from zero to 30%, which takes a significant time, a fixed amount is simply added to the output. The value of the offset was chosen as being around 20% of the maximum controller output figure.

Setting Up The Gains

These two controls need to set in tandem as one very much influences the other.

There are many ways to arrive at an initial setting of the gains but here is one we found worked reasonably well; Ideally there needs to be some means of easily applying a zero to full speed step change to the setpoint input or at least a single transition from the minimum to maximum expected setpoint. This is so that the response of the controller to step changes can be observed. Set the setpoint to maximum speed and with the integral and derivative gains at zero, increase the proportional gain so that the speed reaches the maximum possible before a speed oscillation sets in. Reduce the setpoint to zero. Repeatedly apply a step change in setpoint to 75% of full speed and increase the integral gain gradually until the speed starts to overshoot.

The speed should now rise quickly with the step change and settle at the setpoint without significant overshoot. The integral gain setting will be particularly influenced by the moment of inertia of the load and some experimentation will be required. The controller is now configure as a proportional-integral controller which should quickly correct speed errors without oscillation.

The derivative term can be introduced but in many applications it is not necessary. However some of the fun of PID controllers is finding the right combination of values to give fast response to setpoint and/or load changes. The derivative term is good at removing sudden speed changes or the rapid "hunting" of speed that can result from the proportional gain being too high. It can also be useful in eliminating overshoot when the integral gain is very high. The deadband constant can also be useful in preventing the controller from continually tweaking the speed around the exact setpoint when the error is insignificant.

Software Performance

In such a system, the runtime of the main PID interrupt is critical. In the example, at 20MHz with zero waitstates, 16-bit non-mux bus, the interrupt ran in a worse case time of 6us.

Mission Accomplished

Going back to the original mission of getting the clock to run and keep reasonable time over a few hours was achieved. The output speed needed to be 600 rpm so that the clock's 100:1 gearing would yield 1 rpm for the second hand drive. The controller's speed scaling constants were set accordingly. Thus a setpoint of 256 bits produced a motor speed of 600rpm. After tweaking the deadband width and integral gain, the speed had a steady-state error of +/-0.8% with a peak transient error of +1.5%, -1%. This translates into the clock gaining or loosing 4 minutes over an evening, sufficient to give the appearance that it was a real timepiece, especially if the guests were the worse for drink.

Future Improvements

In many systems, the gains required for good performance at high speeds are different from what is best at lower speeds. A mechanism for coping with this will be added.

A fundamental problem of simple PID controllers is "wind-up". This occurs when a heavy load is suddenly removed. The accumulated error will be very high and the output at maximum. The speed will increase rapidly until this error can be "burnt" off. The derivative gain can be useful in suppressing this effect but a proper mechanism for handling it is really needed.

Finally, the controller can easily be adapted to receive its parameters via the CAN module and by the time this Newsletter hits the streets, this should have been done.

Getting Hold Of The PID Controller Example

The source code for the PID controller is available on an "Open Source" basis for you to modify for your own purposes. It was written using the Keil C166 v4 compiler and uVISION2 workbench. It is included in the Infineon (Siemens) C166 starter kits, although the motor and optical chopper will have to be added by the user!
You can download the program from here for just £1 (worth it if it means your project can at least get started!). A suitable C167 starter kit on which to run the code can be found here or you can buy one on-line here.

The program has been written in such a way that it could be ported to an 8-bit CPU quite easily. The obvious choice might be the C505C as this has both a CAPCOM unit and 10-bit AD convertor. We hope to have such a version soon.

Further Material On PID Controllers

There is some good and more academic material on PID control theory and practice at several university sites on the web. At http://www.eng.uwyo.edu/tutorials/matlab/ctm/ examples/motor/PID2.html you will find some specific examples of DC motor controller tuning.