# **EEE8147 Buck MSc PIC Controller Code Guide**

The controller is designed to operate using time triggered scheduling. After initial initialisation code is complete, the controller enters an infinite while loop. It therefore relies on the interrupt function complete all tasks. This document explains the general operation of all modules in the controller.

Contents

[**EEE8147 Buck MSc PIC Controller Code Guide** 1](#_Toc143014910)

[Initialisation 1](#_Toc143014911)

[GPIO 2](#_Toc143014912)

[ADC 4](#_Toc143014913)

[PWM Module 5](#_Toc143014914)

[Interrupt Routine 6](#_Toc143014915)

[State Machine 7](#_Toc143014916)

[Gain and Exponent Calculation 8](#_Toc143014917)

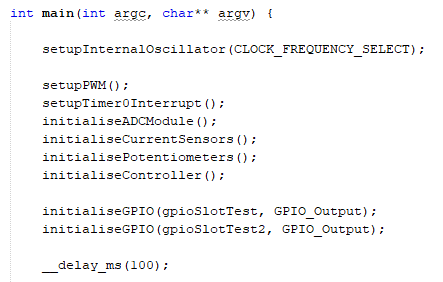
[Potentiometer and Scaling 9](#_Toc143014918)

[PI Controller 9](#_Toc143014919)

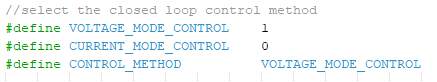
[Current Sensors 12](#_Toc143014920)

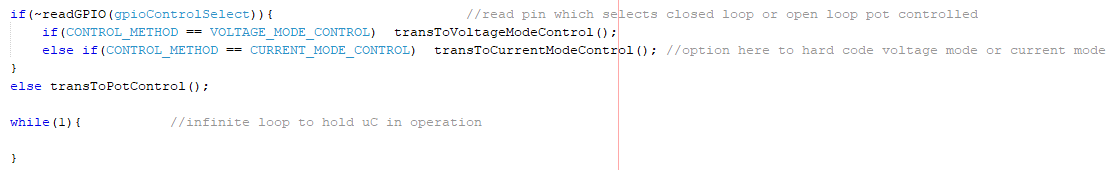
# Initialisation

The clock frequency is first selected. An enumerator named internalClockFreqSelect in contains a list of all possible clock frequencies and the function ‘setupInternalOscillator’ implements the selected frequency. The #define macro CLOCK\_FREQUENCY\_SELECT should be changed in order to change the frequency, using the internalClockFreqSelect enumerator. These are all in the Global.h header. Following this, the initialisation functions are ran to setup pins, and variables for all modules.



Following this, the Closed Loop enable pin is then polled to check for closed loop or open loop operation, and runs the state machine transition function accordingly. Once read, the PIC will remain in this operation mode permanently until being reset. In addition, the CONTROL\_METHOD define macro, which is set in the Controller.h header, must be set to the required control method. Note the code for current mode control has not been implemented. This define macro not only influences the state machine in the code below, but also prevents the code for the other control method from being compiled, reducing memory usage.



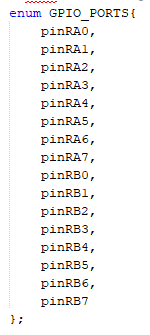


# GPIO

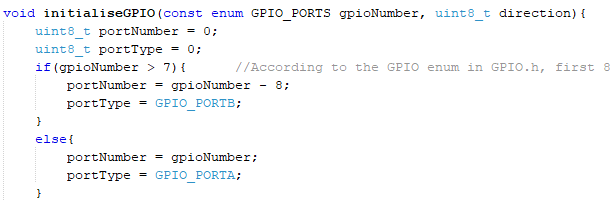
Each GPIO pin has two define macros. The pinX macro is unused, but is indicative to easily check which pin it is. The gpioNAME macro contains the A/B port number, in the format pinRAx or pinRBx, taken from the enumerator GPIO\_PORTS, in the GPIO.h header.



As enumerators effectively just order the contents numerically, pinRA0 translates directly to integer ‘0’, and pinRA7 to ‘7’. pinRB0 however translates to ‘8’.



The first step in both GPIO functions (initialise, read and write) is to determine the port type, and gpio number. According to the enum, if the passed gpio macro is above 7, it belongs to port B, and below is port A. If it is port b, 8 is subtracted from the passed macro to obtain the gpio number. Ie pin RB0 is 8. Therefore, subtracting 8 gives 0, which is the gpio number.



The passed direction argument, which takes the form of a macro, where GPIO\_Input = 1 and GPIO\_Output = 0, is used to set the TRISA or TRISB register accordingly. In the register, a 1 represents an input, and a 0 as output. To set a bit in the register, an 8 bit value is created with the bit value 1 shifted by the gpio number, creating a single bit set to 1 and all the others 0. For example, if we want to set bit 3, we create a byte containing 00000100. We then ‘or’ this with the register. This will cause the bit we want to modify in the TRISx register to definitely have a 1 afterwards, and all other bits will be unmodified (remain as a 0 or 1 as before). This code is shown below.



Alternatively, for an output, we want the corresponding bit in the TRISx register to be cleared (0). To do this, we create an 8 bit value, with the bit number we want cleared set to 0, and all other bits set to 1. This is created easily by shifting the bit value 1 by the GPIO number as before, and inverting the result. For example, to clear bit 3, we create a byte containing 11111011. We then ‘and’ this with the register. The bit we want to modify will definitely have a 0 in afterwards, and all other bits will be unmodified (if a 1 is present, 1&1 = 1, if a 0 is, 0&1=0 so no change). This code is shown below.



In addition to setting direction, we always clear the corresponding bit in the ANSELx register to set the pin as a digital IO.



For writing to a register we set the LATx bit accordingly.



For reading, we take the value directly from the PORTx register. We must bit shift and ‘and’ to obtain a byte with the resultant read bit in the first bit location. Ie 0000000X where X is the read bit.

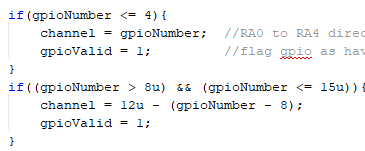


# ADC

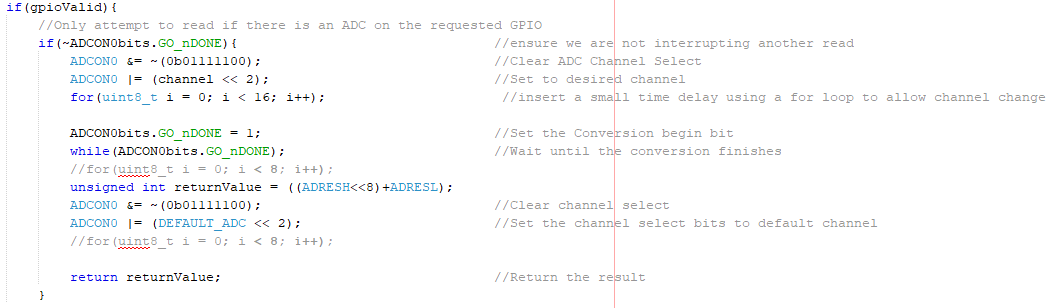
InitialiseADCModule function sets up the ADCON0 and ADCON1 registers. ADCON0 contains the channel select and conversion begin bits, only bit 0 needs to be set to enable the ADC module. ADCON1 contains settings for the format of the two byte result registers, conversion clock frequency, and supply voltage sources. Importantly, bit 6-4 containing the clock frequency is set to Fosc/8. Higher frequencies seem to cause malfunctions.

The initialiseADCPin function sets up the required pin as a gpio input, as in the intialiseGPIO function previously. However, the main difference is that only some GPIOs available as ADCs. Therefore this function only writes to the TRISx and ANSELx register if an ADC is available on the pin.

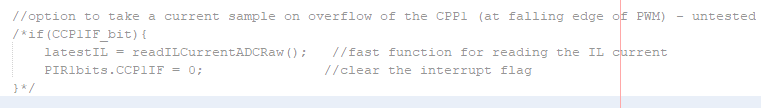
The readADCRaw function requires identification of the ADC channel from the GPIO pin. pinRA0 to pinRA4 directly translates to ADC channels 0 to 4. Pins between PinRB1 to pinRB7 are the only other pins with ADCs. However these ADC channels are numbered differently. The code below converts the enum numbering to standard numbering (i.e giving 1 for pinRB1 and 7 for pinRB7), then converts to the ADC channel numbering (I,e RB1 is channel 11, RB2 is channel 10).



If the passed gpio pin argument contains a valid ADC, a read is attempted. The ADCON0bits.GO\_nDONE bit is 1 if a read is undergo, and 0 if the module is free. This is first checked to be free. Then, the ADC channel select bits are cleared in ADCON0, then written to the desired channel. A small software delay is inserted using a for loop, counting to 16, allowing time for the bits to change. The conversion bit ADCON0bits.GO\_nDONE is then set to 1, activating a read. A while loop is entered which holds the controller while this bit is 1. When the read is finished, the processor automatically clears this bit. The registers are then returned, by shifting ADRESH up 8 bits for the high 8 bits, and taking ADRESL for the lower 8 bits in the resultant 16 bit read value. Note that the actual data is only 10 bits, and the highest 6 bits of ADRESH are blank. This format can be swapped by altering bit 7 in the ADCON1 register.

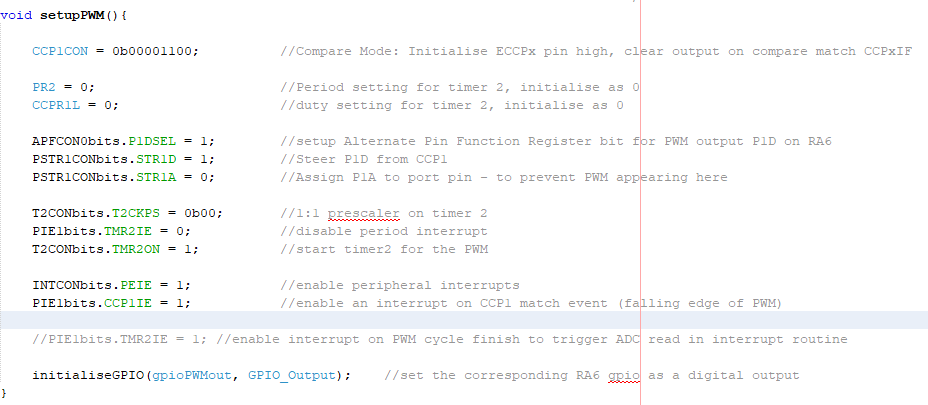


Note at the end of this read function, the channel select bits are set to a value given by DEFAULT\_ADC macro. This contains the iL ADC channel – so no channel change is needed for very fast reads of iL ADC in the function readILCurrentADCRaw. This would be used for peak current control method. Also see the commented out CCP1IF\_bit function in the interrupt function. This would run when the capture compare is detected on the PWM module, i.e the falling edge of the PWM. This would allow measurement of the peak current when the MOSFET switches off. This is untested.

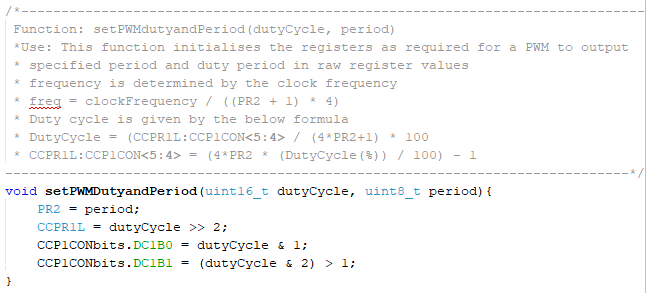


# PWM Module

The setupPWM function initialises the PWM module, using capture compare module 1, steering the P1D output from CCP1 to RA6 and deactivating P1A. Timer2 is the source for the pwm counting, this is set without a prescaler. The interrupt for a falling PWM edge (CCP1) is set as mentioned before which could be used to obtain peak IL current but is untested. Finally, the GPIO for pin RA6 is initialised as an output.



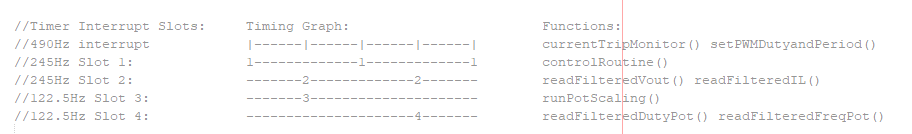
Setting the PWM simply requires changing the registers PR2 and CCPR1L and 2 bits in the CCP1CON register. The PR2 bits are proportional to the period, as shown by the formula in the description below, this is an 8 bit value. 100 percent duty cycle corresponds to a value 4 times that of PR2. Therefore, the duty cycle must be 10 bits where the period is 8 bits. These additional bits are located in the CCP1CON register, as CCPR1L is 8 bit.



# Interrupt Routine

With a maximum clock frequency (32MHz), the interrupt function has been chosen to operate at 490Hz, set in the setupTimer0Interrupt function, with a prescaler of 64. This is to allow enough time for the code to run without causing any overrun. Particularly, the controller code which needs to perform several calculations therefore is at risk of overrun if the interrupt occurs to frequently. It was found that 490Hz was optimal.

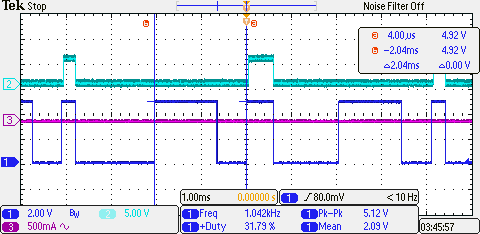
The interrupt itself is split into 4 slots, using if statements and Boolean flags. Slot 1 and slot 2 occur alternatively, at 245Hz. Slots 3 and 4 occur alternatively within slot 2, at 122.5Hz. The diagram below demonstrates operation, and functions within each slot.



The current execution times are summarised in the table below, with 2ms being the interrupt execution rate. Slot 2 is equal to slot 3 and slot 4 as no other functions occur within it.

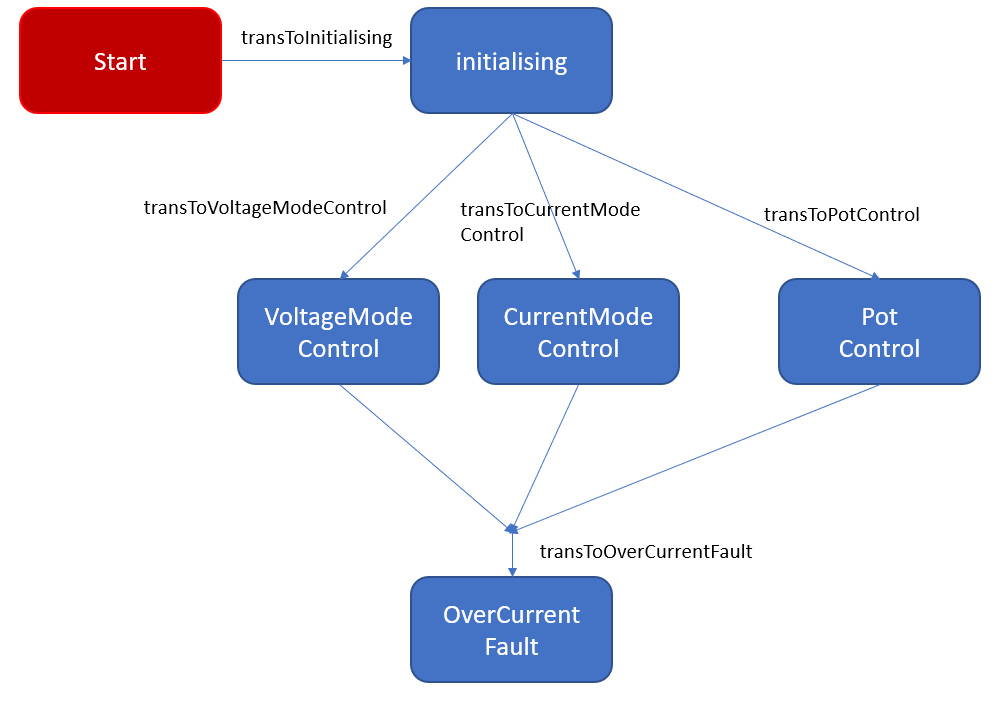
|  |  |  |
| --- | --- | --- |
|  | Timing (ms) |  |
|  | Closed Loop | Open Loop |
| Slot 1 | 1.38 | 0.044 |
| Slot 3 | 0.3 | 0.284 |
| Slot 4 | 0.6 | 0.604 |

Pins RB4 and RB5 can be used to check execution rate, with RB5/ gpioSlotTest2 only showing slot 2, and RB4/gpioSlotTest showing all slots. An example is below, for closed loop. As visible, slot 1 occupies a large portion of the available time. Slot 2 (3 and 4) execution time varies as one is slot 3, one is slot 4.



# State Machine

The state machine is very simple and can be summarised in the following flow chart. It is implemented using a single variable, currentState, which tracks the current state, and an enum stateMachine which holds all possible states. A function exists for transitioning to each state. This changes the currentState variable and performs any set up.



# Gain and Exponent Calculation

Before moving onto potentiometer scaling and the controller, an introduction to the method of multiplication and storing decimal point gains will be given. Since many of the gains required in the scaling and controller code require precision to several decimal places, the gain and exponent method is used. This performs an integer multiplication before performing a bit shift down to achieve the equivalent of multiplying by a decimal number. Generally, the larger the bit shift, the more precision achieved by the gain/exponent method. Due to rounding the gain value to an integer, some precision will always be lost.

The advantage of using this method however is that the calculation is fast, as bit shifting is very quick, and memory consumption is saved as floating point arithmetic is not required, which uses a large amount of memory.

For example, if we need a gain of 0.101. Firstly, if we test with a bit shift of 6. This is the equivalent of dividing by 2^6 = dividing by 64. 1/64 = 0.015625. To get a gain of 0.101, the multiplication must be 0.101 / 0.015625 = 6.464. This is rounded to 6. The real gain is therefore 6/64 = 0.09375, which is significantly smaller than 0.101. Therefore we need a larger bit shift to gain more precision.

Next, test the result with a bit shift of 8. This is the equivalent of dividing by 2^8 = dividing by 256. 1/256 = 0.00390625. To get a gain of 0.101, the multiplication must be 0.101 / 0.00390625 = 25.856. This is rounded up to 26. The real gain is therefore 26/256 = 0.1015625, which is slightly larger than 0.101 but accurate enough.

Below is an example of this method from the controller code.





# Potentiometer and Scaling

The potentiometer duty cycle and frequency ADC values are read in the functions readFilteredDutyPot and readFilteredFreqPot within the interrupt. These functions perform an ADC read, then insert the newest values into a FIFO, dutyPotFIFO and freqPotFIFO. They are set within the potentiometer.h header to have a size of 16. The mean of the 16 values is then calculated and returned from the function. The latest mean value is now stored in the variables filteredDutyPot and filteredFreqPot. Within the interrupt, the function runPotScaling then uses these values to perform the potentiometer scaling. To reduce jitter on the PWM signals, the execution rate of pot scaling is reduced down using a simple counter, and the define macro POT\_SET\_DIVIDER. Where the execution frequency is Slot 4 Freq / POT\_SET\_DIVIDER.

Within the scaling function, the period of the PWM is calculated using the formula:

setPeriod = ( (filteredFreqPot – POT\_OFFSET )\* (MAX\_PERIOD\_FROM\_POT – MIN\_PERIOD\_FROM\_POT) / 1024 ) + MIN\_PERIOD\_FROM\_POT

The min and max duty cycle values are then calculated based on the period. 100% duty equates to 4\* the period, and the MIN\_DUTY macro is in percent, therefore:

minDuty = MIN\_DUTY \* setPeriod / 25

maxDuty = MAX\_DUTY \* setPeriod / 25

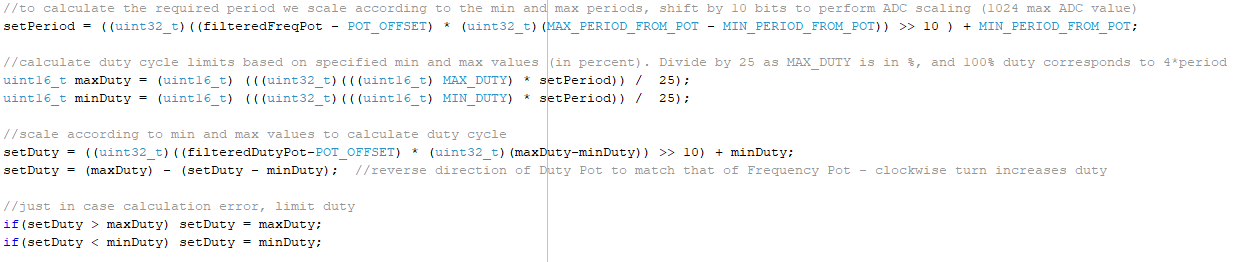
Now we know this, we can calculate the duty cycle, and scale based on the pot value:

setDuty = ((filteredDutyPot – POT\_OFFSET) \*(maxDuty-minDuty) / 1024 ) + minDuty

Finally, we need to reverse the direction of the duty pot, as per hardware requirements.

setDuty = maxDuty – (setDuty – minDuty);

All of this code is shown below.

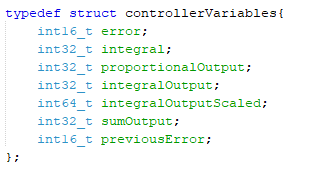


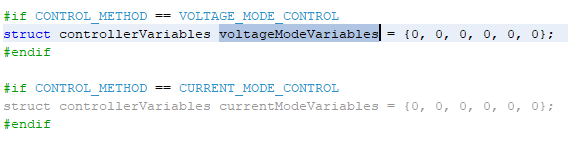
# PI Controller

The initialiseController function which is ran in the main, sets up the output voltage measurement ADC, then also calculates the maximum integrator limit, but scaled up according to the number of bit shifts performed in the PI controller (the Ki bit shift and the Dt bit shift). These shifts occur due to using a gain/exponent calculation method explained earlier, which avoids floating point numbers and is faster. However, due to the very small increments in integrator which is required (fractional numbers <1), the integrator value is stored as a scaled up value, and windup limiting must be performed before scaling down. Therefore the initialise function calculates this scaled up limit based on the scaled down integrator windup limit, and the bit shifts performed. The scaled down limit has been set with INTEGRAL\_LIMIT in the controller.h file. 50% maximum possible duty has been chosen, since the controller can output either +50% or -50% duty. This is 1024/2 = 512. The scaled up value is calculated as below.



All PI calculated values are stored in the struct variable voltageModeVariables or currentModeVariables. The struct is shown below, followed by the define.





The controller is executed within the interrupt, controlRoutine determines which controller to run based on the state machine. The PI controller for the method chosen is then ran. Voltage mode is the only method which has been developed, and will be explained below. First, the filteredVout value, which is calculated using a FIFO in the interrupt as with the filteredDutyPot and filteredFreqPot values, is converted to milli volts using parameters for the voltage divider, in convertRawToMilliVolts. The PI calculations are standard, as summarised below. The control select header pin is checked as this can be used to change the voltage reference from two possible values, TARGET\_VOLTAGE\_MV\_1 or TARGET\_VOLTAGE\_MV\_2. I.e if the header is present on boot of the PIC, it will go into closed loop mode. The initial voltage reference is TARGET\_VOLTAGE\_MV\_1. If this header is removed, the reference will change to TARGET\_VOLTAGE\_MV\_2.

Error = TARGET\_VOLTAGE\_MV\_1 – newVoltage

We then perform the integral for the latest time step, with the gain part of DT, and multiply the gain part of Ki.

Integral(scaled up) = VOLTAGE\_MODE\_KI \* error \* DT\_GAIN

At this point, the calculated value has not been scaled down according to the VOLTAGE\_MODE\_EXPONENT and DT\_EXPONENT which are required as part of the VOLTAGE\_MODE\_KI and DT\_GAIN multiplications.

The calculated increment is added onto the moving integral. The increment is very small, whereas the moving integral has built up over time and is large. If the increment had been scaled down first, we would lose all precision, and the moving integral would stay as 0.

integralOutputScaled = integralOutputScaled+ integral

The windup limit is then performed

If(integralOutputScaled > integratorScaledLimit)

integralOutputScaled = integratorScaledLimit

And same for the negative limit.

Finally, the scale down is performed

integralOutput = integralOutputScaled >> (DT\_EXPONENT + VOLTAGE\_MODE\_KI\_EXPONENT)

Next the proportional output is calculated, and is scaled down since no windup limit is required.

proportionalOutput = (VOLTAGE\_MODE\_KP \* error) >> VOLTAGE\_MODE\_KP\_EXPONENT

Finally, the integral and proportional outputs are added to get the PI output.

sumOutput = integralOutput + proportionalOutput

This can be all shown below in the real code.

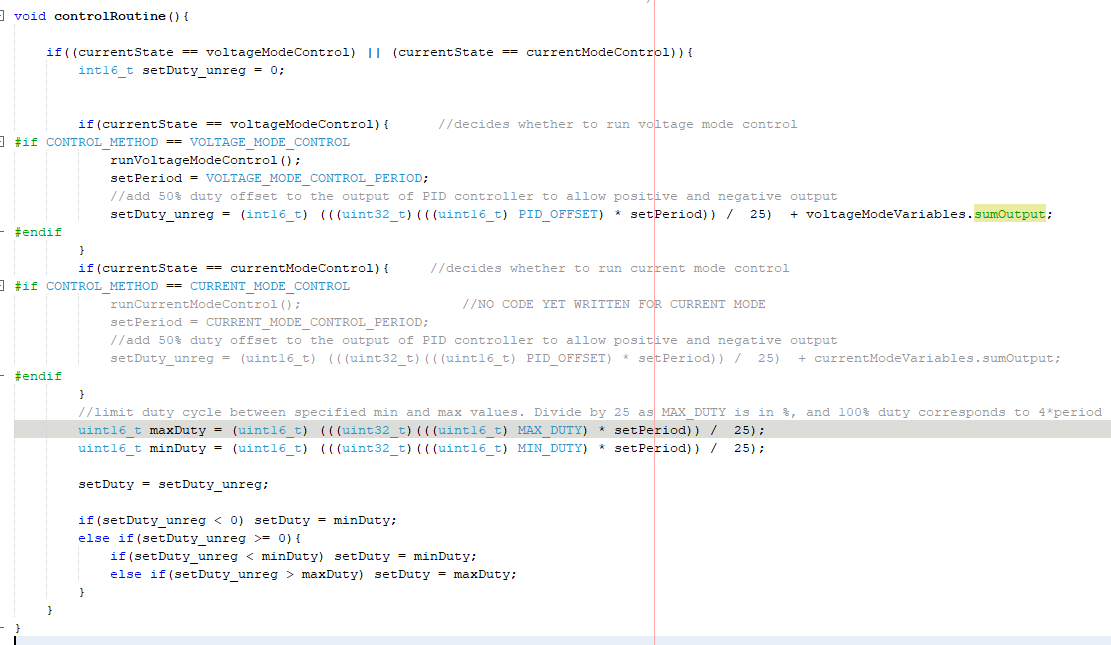


The PI output is added onto the PI offset to get the duty cycle unregulated. The PID\_OFFSET is set to 50%, allowing a positive or negative PI output rather than purely positive, allowing the P component to work. The PID\_OFFSET is in percentage, therefore period must be used to calculate the value first before adding.

setDuty\_unreg = (PID\_OFFSET) \* setPeriod / 25) + sumOutput

Finally, the min and max duty cycles are calculated based on the current period. setDuty\_unreg is checked against the limits, and setDuty is set to setDuty\_unreg, or the limits.

This can be all seen in the code below.

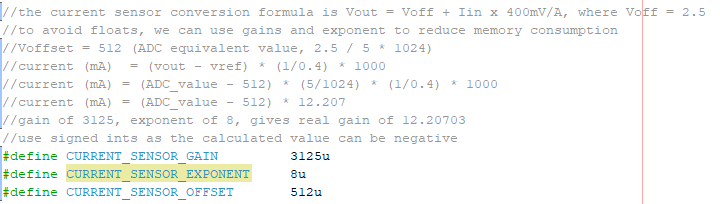


# Current Sensors

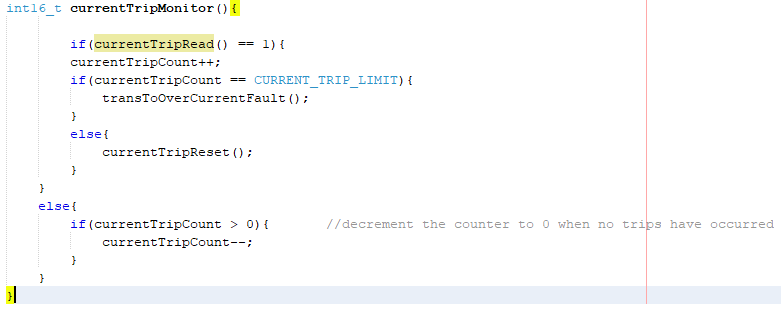
The current sensors are fed into ADC inputs, and can be read using the ADC. In addition, a GPIO input from each current sensor is pulled low if a current trip has occurred on the current sensor. If this has occurred, the current sensor breaks the circuit and must be reset. One single GPIO out then acts to reset both chips by turning off an NMOS, which allows the Voc pin on the current sensor to be pulled to Vcc, resetting the chip.

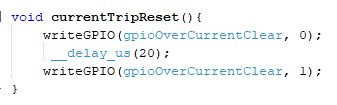
Firstly reading from the ADCs will be described. The raw values are obtained and filtered just as with the previous ADC values using FIFOs. Currently, the filteredIDS read is commented out of the main interrupt. Neither current sensor values are actually used, but the filteredIL value is in the template for the current mode control PI loop, so is obtained in the interrupt.

To convert the raw ADC value to a signed current, the offset is removed, and the reading is scaled up to give a value in milliamps, in the convertRawToMilliAmps function. The parameters are calculated as below, again using the gain and exponent method.



The current trip signals are both polled as GPIOs within the currentTripMonitor function, called every interrupt, which calls the currentTripRead function. If this function returns a 1, then one of the signals have been pulled low, meaning the sensor has tripped.

An issue was found in that simply switching the inductor toggle switch causes a spike in current, which trips the current sensors. This is not ideal when performing experiments as we want to be able to change inductors on the go. Therefore, a currentTripCount is used, which counts consecutive trips. This is incremented each time a trip is detected in the currentTripMonitor function, or is decremented if the function has been polled and no trip has occurred. If the trip count is below the limit, CURRENT\_TRIP\_LIMIT, the reset GPIO is set to low for 20us, in the currentTripReset function, before pulled set again. This turns off the NMOS which resets the chip. If the trip count has reached the limit, the state machine transitions to a hard fault. The functions are shown below. 



The result of this is that there will be some trips on change of the inductor, but these are reset quickly and will not affect steady state operation. However, if a genuine hardware problem has occurred, pulling a large amount of current from the supply, the controller should detect consecutive trips, and go into a hard fault. This should tell the operator that the board has a fault and needs investigation.

The current trip threshold, which is set via a voltage divider on the Voc pin (to be 0.091Vdd, giving 1.2xIP which is 6A) may be above the current draw capability from the bench supply at 24V, therefore an overcurrent may not be detected in the case of a short. However the current flowing will be at a safe level and the supply will be seen to be current limiting, and the voltage will drop, indicating a fault.

Below are the schematics for the current sensor, for reference.

