ECE5881 Real Time System Design

Pendulum Design Exercise

A/Prof Lindsay Kleeman, Monash University.

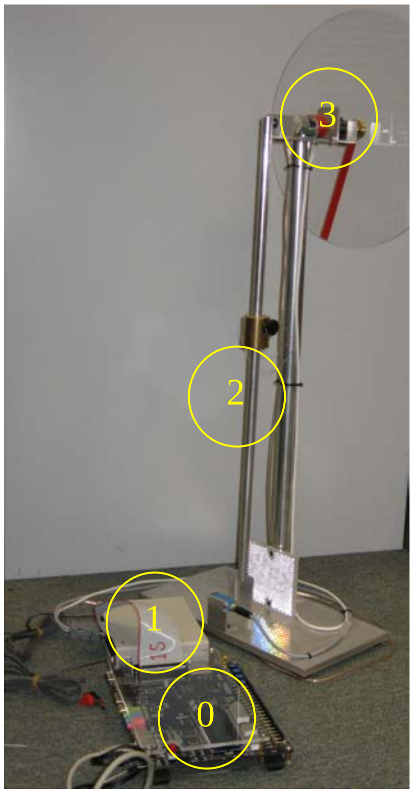
0 Aim

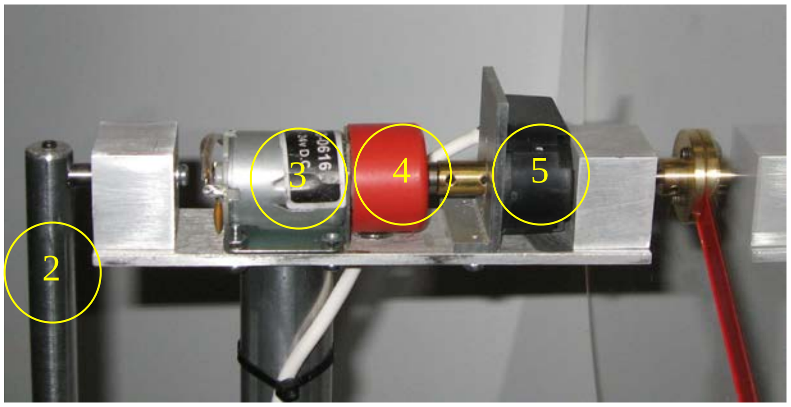
To develop a real time system to sense the motion of a pendulum and reproduce the same motion with a DC motor driving a pointer. The following learning objectives will be achieved by completing this design exercise:

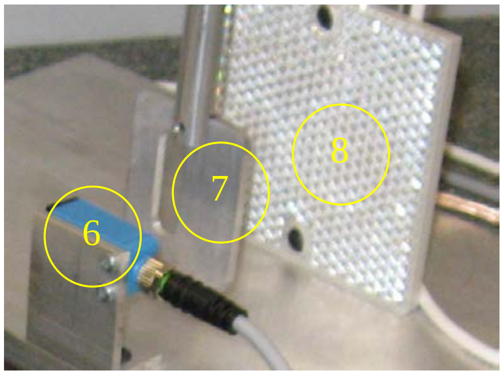
1. Develop and practise real time embedded system design, testing and debugging techniques.
2. Understand the trade-off between hardware and software implementations in a real time embedded system by studying a software hardware co-design problem.
3. Interpret timing information from a sensor and its relationship to a physical system.
4. Design modules that respond to sensor changes within hard deadlines.
5. Use Mathematical modelling of a physical system to guide real time design and implementation.
6. Design a PID motor controller with PWM output to produce desired motions in real time.

1 Equipment

- Altera Quartus/ NIOS II development software
- DE2 FPGA Development Board
- Pendulum mechanical system with:
  - Free swinging pendulum with variable position weight
  - Light beam retro reflective sensor
  - Slotted asymmetric mask mounted on the end of the pendulum
  - Co-axial DC motor with quadrature optical shaft encoder
- Interface Electronics with a 12 V plug pack power supply and sensor/motor connections.







Photos Above: Pendulum Apparatus showing DE2 board (0), Interface Electronics (1), Pendulum (2), Motor (3) driving Perspex disk with red position marker, gearbox (4) and co-axial encoder (5), Light Beam sensor (6) with asymmetric mask (7) and reflector (8).

2 Risk Assessment

There is the unlikely risk that the pendulum may strike a user or other people in the lab. For this reason the following precautions must be observed:

1. Ensure that the equipment is positioned away from people and in the centre of the bench with the swinging pendulum on the far side of the user.
2. Ensure that the fishing line constraint is present and working. It must prevent the pendulum swinging more than 45 degrees from the bottom rest position. The fishing line limits the energy of the swing and prevents the pendulum becoming unbalanced due to large swing excursions.
3. Keep hands and face clear of the bottom of the pendulum when it is operating.

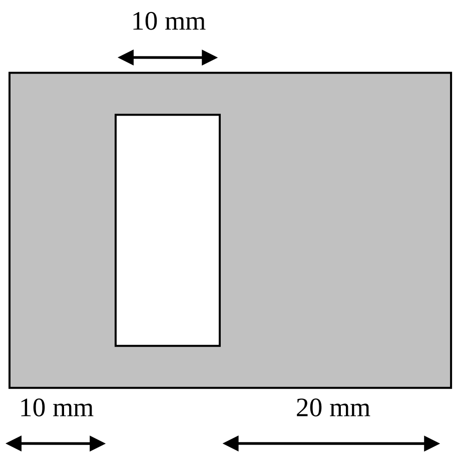
3 Background and Preliminary Work

Pendulum motion is a well understood behaviour that can be simply modelled Mathematically. For small angles of deviation of the pendulum from its rest position, the motion approximates simple harmonic motion. Simple harmonic motion is characterised by an acceleration being proportional to the displacement.

Derive the equations of motion for a pendulum of length R to a point mass m and small angle from rest of θ. Start by showing that the force component acting perpendicular to the pendulum (that is acting in the positive direction of angle) is $-m\,g\,\sin\theta \cong -m\,g\,\theta$ for small angles θ where g is the acceleration due to gravity. Use this result to find a 2nd order differential equation in θ and check that a sinusoidal solution exists with amplitude A, period T and phase angle ϕ. Find the period of this sinusoidal solution in terms of g and R. What is the maximum speed of the pendulum? Given the maximum speed and period, find the maximum amplitude A of the angle. These equations allow you to characterise the pendulum motion given measurements of its period, direction and speed.

In practice some energy is lost on each cycle due to friction in the pendulum bearing and air resistance. The amplitude of the swinging pendulum therefore decreases gradually. However sufficient accuracy can be obtained for our purposes if we assume there are no losses over each half period of motion and update the motion estimate at the end of each half cycle.

A light beam is mounted at the bottom of the pendulum path as shown in the photos above. The beam shines onto the corner cube reflector on the other side of the swinging pendulum and the corner cube returns the beam in the opposite direction to its arrival. The beam sensor output is a logic signal that is 1 when the beam is detected and 0 when the beam is blocked. A mask shape shown below is used to cut the beam in a particular pattern. What is the purpose of this mask shape?



4 Method

4.0 Setting up and Testing the Equipment.

1. Connect:
    - the ribbon cable to the inside JP1 connector
    - D9 connector (encoder and light beam)
    - motor cable
    - 12V power supply – ensure power is OFF when making connections.

2. Power the DE2 board before the interface board.
3. Power the Interface 12V supply. The red and orange interface box LEDs and the beam sensor LED should then be illuminated. The orange beam sensor LED should turn on when the red visible beam is unobstructed.

Test the equipment by using Quartus to download to the DE2 board the *motor.sof* file provided on the unit website. The following functions and displays are then available:

KEY[0] press for a reset
SW[9:0] is the signed 10 bit binary input to the PWM motor control

HEX3-HEX0 displays SW[9:0] in decimal.

LEDG[0] is the Hbridge_PWM
LEDG[1] is the Hbridge_InB
LEDG[2] is the Hbridge_InA

LEDG[5] = beam;
LEDG[6] = enA;
LEDG[7] = enB;

HEX7-HEX4 displays encoder angle count from reset
where the encoder has 2000 counts per revolution

Test that the encoder counts in both directions without losing steps – you can apply a reset with KEY[0] when the red disk line points downwards as a reference.

Test the motor drives in both directions using positive and negative binary SW values. SW[9] is the sign bit.

Check that the beam sensor is working as the pendulum moves slowly past.

4.1 Software Encoder Interface

The aim of this section is to accumulate and display the encoder shaft angle of the motor from the A and B outputs of the incremental encoder. The operation of an incremental shaft encoder is described in the lectures. Here are the detailed steps:

1. Create a new Quartus Verilog project with a NIOS-II processor and SOPC modules. Parallel port modules should be included that are configured as:
    - inputs from the encoder that generate interrupts on both rising and falling edges,
    - outputs for the HEX displays for the angle and
    - input from a KEY pushbutton to reset the encoder angle
    - inputs from the SW switches for selecting angle units for display –see 4 below.
    - Outputs for bit twiddling monitoring of the ISR latency on an oscilloscope.
    - JTAG UART
    - University Program SSRAM as described in the RTSchedLab.
    - Interval timer for a 1 msec real time interrupt.

    Ensure you ***allocate pins*** appropriately – see Appendix A at the end of these notes.

2. Create a new NIOS-II IDE uCOS-II Hello World project.

3. Write an interrupt service routine (ISR) to handle the changes in encoder A and B signals that updates an angle counter.

4. Display the angle with sign on HEX 7 segment that displays in native 2000 counts per revolution or degrees depending on a SW input.

By timing the interrupt latency using bit toggling techniques similar to the Interrupt Latency Lab in ECE3073, estimate the maximum speed that the shaft can rotate with your software interface. Compare this with the specification of maximum speed in the data sheet on the unit webpage for the shaft encoder.

Max speed of rotation achievable = …………………………………… (RPM)

4.2 HDL Encoder Interface

Achieve the same functionality as in the previous section except replace the ISR with a Verilog HDL design discussed in lectures and clocked at 50 MHz,. The output of the Verilog Encoder module should be a 32 bit signed angle counter that is interfaced to the NIOS processor with a synchronous PIO module. Estimate the maximum speed achievable for this hardware solution and again compare this with the encoder data sheet.

Max speed of rotation achievable = …………………………………… (RPM)

4.3 Motor Pulse Width Modulation (PWM) Interface

The aim in this section is to implement a hardware digital PWM module. This takes a binary input of N bits called *PWM_data_input* representing a desired average motor voltage and asserts the output *PWM_out* 1 to turn on an H-bridge so that it supplies full voltage to the motor or 0 for 0 Volts to the motor. The average motor voltage is arranged by the PWM module to be proportional to the binary input of the PWM module. The duty cycle of the output is equal to the N bit input/2^N in the case of an unsigned input of Part 1 below. The PWM output is synchronised to the PWM clock. There are 2^N clock cycles in a ***fundamental*** cycle of the PWM. The *PWM_out* is high for the first *PWM_input* number of clock cycles and low for the remainder. This can be implemented with a binary counter and comparator and described in Verilog with a single always block. Note that the fundamental cycle is the period of the fundamental frequency of the output digital signal coming out of the PWM module – that is if you looked at the Fourier series representing the output then is lowest frequency component would be the 1/(fundamental cycle). The fundamental frequency needs to be much higher than the motor response frequencies for the averaging to work well in practice. It is often possible to hear the fundamental frequency of the PWM drive coming from a motor when it is within audible frequencies of 20-20 kHz.

For example, a 10 bit unsigned *PWM_data_input* we would need 2^{10} = 1024 clock cycles to complete one fundamental PWM cycle. If the *PWM_data_input* was 50 then the output is asserted for 50 out of the 1024 clock cycles or approximately 5% of full voltage. If a fundamental PWM frequency of around 10 kHz is used, then due to the motor dynamics being much slower than this, the motor "sees" only the average. What is the frequency of the PWM counter update for a 10 kHz fundamental and 10 bit *PWM_data_input*?

PWM counter update frequency = ……………………. Hz

The PWM module will be implemented in hardware here since a software solution is impractical. Why?

4.3.1 Unsigned PWM

Start with the simple unsigned PWM Verilog module template below. Complete the module so that the counter *count* increments on system clock rising edges whenever the clock enable *CE_in* is 1. *CE_in* is defined elsewhere so that *CE_in=1* for one clock period every *k* clock periods and 0 the rest of the time. This provides a mechanism to divide down the system clock rate whilst maintain the same system 50 MHz clock. A similar approach was taken with the "microsecond counter" in the Real Time Scheduling lab where a CE was asserted once every 50 clock cycles of the 50 MHz system clock.

When the counter *count* is less than the *PWM_data_input, PWM_out* should be 1 otherwise 0. It would be a good idea to only allow *PWM_data_input* to affect the

PWM_out at the end of each PWM fundamental period rather than allowing it to change mid cycle and produce pulses that are not synchronised correctly.

Simulate your design to check that the timing is correct – you may wish to reduce the PWM_IN_SIZE parameter to 4 to limit the duration of the simulations. If you have forgotten how to simulate a Verilog project, consult this tutorial document or see the resources section of the unit website:

ftp://ftp.altera.com/up/pub/Tutorials/DE2/Digital_Logic/tut_quartus_intro_verilog.pdf

You must **ALWAYS** check the hardware synthesized from your HDL description in Quartus by using the menu ***Tools->netlist viewers-> RTL view***.

```
module PWM(clk_in, CE_in, synch_reset_in, PWM_data_input, PWM_out)

parameter PWM_IN_SIZE = 10;  // this is a constant that can be overridden when
                               // Instantiating this module
input                        clk_in, CE_in, synch_reset_in;
                             // synch_reset_in must be synchronised to clk_in
input [PWM_IN_SIZE-1:0]      PWM_data_input;
output reg                   PWM_out;
                             // out=1 in proportion to[ecse1]  magnitude of
                             // PWM_data_input/2**[PWM_IN_SIZE]
reg [PWM_IN_SIZE-1:0]        count;
        // … other local signals declared here
always @(posedge clk_in)
        if (synch_reset_in)
                …
        else if (CE_in) begin

                …
        end
endmodule
```

Use a simple Quartus Verilog project to connect the SW[9:0] switches to your *PWM_data_input* and LEDR outputs and design a *CE_in* so that the fundamental PWM frequency is close to 10 kHz for the 50 MHz clock. Drive the H-bridge logic lines with an appropriate pin assignment – see Appendix A and the H-bridge data sheet on the unit webpage. Test your design with different PWM inputs. Show your demonstrator the *PWM_out* on a CRO by using free pins on the outer GPIO lines of the DE2 board. (See ECE3073 Interrupt Latency lab for details).

4.3.2 Signed PWM

Extend your design from Part 1 to a signed 2’s complement version, so that the motor can be driven in both directions. Need a quick refresher on 2’s complement notation? See

http://en.wikipedia.org/wiki/Two's_complement

Note that you will need to find (in hardware) the magnitude of a 2’s complement binary number for comparison with the PWM counter. Here is a simple way to achieve this:

```
wire [PWM_IN_SIZE-1:0] magn
        = (data_in[PWM_IN_SIZE-1] ? 1'b0-data_in : data_in);
```

Simulate and test your solution by driving the motor in both directions. Show your demonstrator the *PWM_out* on a CRO.

Does your design ever output the full range from 0 to full scale – that is have the H-bridge always driving 0V to the case where the H-bridge is always full scale voltage in both positive and negative directions?

…………………………………………………………………………………….

4.4 PID Implementation with a uCOS-II Task

A PID (Proportional Integral Differential) controller is commonly used to control DC motors. The input is an *error* defined as the difference between the desired and encoder positions for the motor shaft angle. The aim of the PID controller is to reduce the error as much as possible subject to the dynamics of the DC motor and its mechanical load via appropriate feedback of a voltage to the motor. The implementation equation suitable for our application is

$$MotorVolt_n = \left(K_p E_n + K_d (E_n - E_{n-1}) + (128 + K_i \sum_{j=0}^{n} E_j)/256 + 128 \right)/256$$

- K_p, K_d and K_i are the proportional, differential and integral gains respectively and E_n=*desired angle – encoder angle* represents the error at time step *n*.

- Since we are using integer representations for the gains and errors, we have incorporated **rounding** of integer division into the above expression with the two 128 additions.

- The divisions by 256 indicate that we are storing 16 fractional bits for K_i and 8 fractional bits in K_p and K_d.

- Integer multiplication, addition and subtraction can cause **overflow**. Choose appropriate numbers of bits in your representations to avoid multiplier overflows. With addition and subtraction, check for overflow and apply saturation to the results of each addition or subtraction. Overflow in addition and subtraction can be detected by looking at the sign bits of the operands and results alone. For example adding two positive integers with a negative result indicates overflow. If overflow occurs, the result should be set to the maximum/minimum allowable number if the result should have been positive/negative. Also the final conversion to a limited (say 10) number of motor voltage bits for the PWM module should be a saturating conversion rather than a simple truncation – the sign must be preserved correctly in the conversion.

Implement and test the PID controller in a single uCOS-II task that runs every millisecond. Here are some of the steps that need to be completed:

1. Check that **correct numerical results** are obtained for corner test cases using *printf* statements or the debugger.

2. Verify that **numerical overflow** is handled correctly by varying gains and injecting synthetic errors and checking that the output is well behaved. Check largest gains and maximum/minimum positive/negative errors.

3. Measure the **execution time** in microseconds of your PID calculation to ensure that the code does not overrun a millisecond (assuming a 1 kHz servo rate). This can be done by using bit twiddling and a CRO or using the microsecond timer that was used in the RTSched Lab.

   Your measured PID execution time = …………….. usec

4. **Tune the gains** of the PID controller by observing the step response of the system. That is servo about a constant set point and physical move the disk by 45 degrees and let it go. You should think about an interface that uses the SW as inputs to change the gains without re-compiling your program. Start with increasing K_p with K_i and K_d both zero until oscillation occurs. Decrease K_p by around 20-50%. Adjust K_i to remove steady state error. Adjust K_d to reduce overshoot. There is a good discussion in Wikipedia that is helpful:
   http://en.wikipedia.org/wiki/PID_controller

5. By using *printf* statements, **record the step response** of your tuned PID controller. Be careful not to cause delay in the PID loop when collecting this data. Plot the step response in Matlab for inclusion in your final report. Show this to your demonstrator.

4.5 Beam Sensor Interface

Connect the light beam sensor to a PIO that generates interrupts on both edges. The beam ISR should update a record of the times in milliseconds and the values of the last *n* sensor readings.

What is the minimum choice for *n*? ……………………………………………….

The ISR should signal a semaphore. Wait on this semaphore in a task that process new data whenever the light beam sensor changes. The processing of the time/value information from the beam sensor needs to reliably and robustly determine when the pendulum crosses the beam sensor, and determine the speed and direction of that crossing. Beware that using a debugger may extend interrupt latency and corrupt the timing measurements. Short *printf* statements can be used judiciously. Test your sensor interpretation software with corner cases of pendulum behaviour: namely small and large angle pendulum swings at different swing periods obtained by moving the mass to each end of the pendulum.

4.6 Motion Generation – Mimicking the Pendulum

In this section integrate all the previous modules and generate a matched sinusoidal motion that mimics the swinging pendulum. One approach worth considering is to use a velocity ***trajectory generator*** module driven by a higher level ***motion generator*** – see Appendix B.

The purpose of the trajectory generator is to take a desired velocity from the motion generator and produce desired position outputs every PID servo loop cycle. The velocity can then be updated at a slower rate than the servo loop time. You need to choose how this is implemented: Verilog or a uCOS-II task. Document the factors that have determined your choice in your notes.

The motion generator is then responsible for producing velocity commands that approximate the perceived state of the pendulum. In practice a quarter cycle of a sinusoidal motion is all that need be stored and around 10 waypoints can be used to approximate the *sine* function. Associated with each waypoint can be a target time of arrival of the motor. As time reaches each waypoint time, a new velocity command can be issued based on the current desired position and the target desired position at the end of the next waypoint. Note that the PID controller provides the only feedback to the motor from the encoder – this results in a clean, simple controller design. Should you choose to use encoder readings at the motion generator level, you run the risk of creating an interacting complicated and potentially poorly controlled system.

Clearly the waypoints need to be updated when either the current times have expired or a new pendulum motion estimate arrives from the light beam sensor. In your design extend the previous pendulum cycle so that smooth motion is maintained, even if the light beam data is late in arriving.

Document the testing and results that you have performed. The demonstrators will have a few tricks up their sleeves to see how well your implementation performs. Try to outsmart the demonstrators with your clever design!

5. Conclusion

This pendulum project has involved many levels of modelling and design of a real time embedded system. Hopefully you will have a much deeper understanding of the concepts that have been presented in lectures now!

Appendix A: Interface Electronics

The following circuitry is contained in the interface electronics

- H-bridge electronics for driving the DC motor.
- Opto isolation and level translation of the H-bridge logic inputs and outputs to be compatible with the DE2 board 3.3 V DE2 inputs and outputs.
- Level translation of the Light Beam sensor and opto isolation.
- Connections to the optical encoder.
- Voltage regulators for the DC motor and optical encoder.
- Protection circuits.

See the unit webpage for data sheets for the different components.

Two independent 12V power supplies should be connected to the Red (positive) and Black (negative) banana plugs. One supply powers the motor and the other the encoder and light beam sensor. The motor supply is kept electrically isolated to reduce noise coupling into the DE2 board.

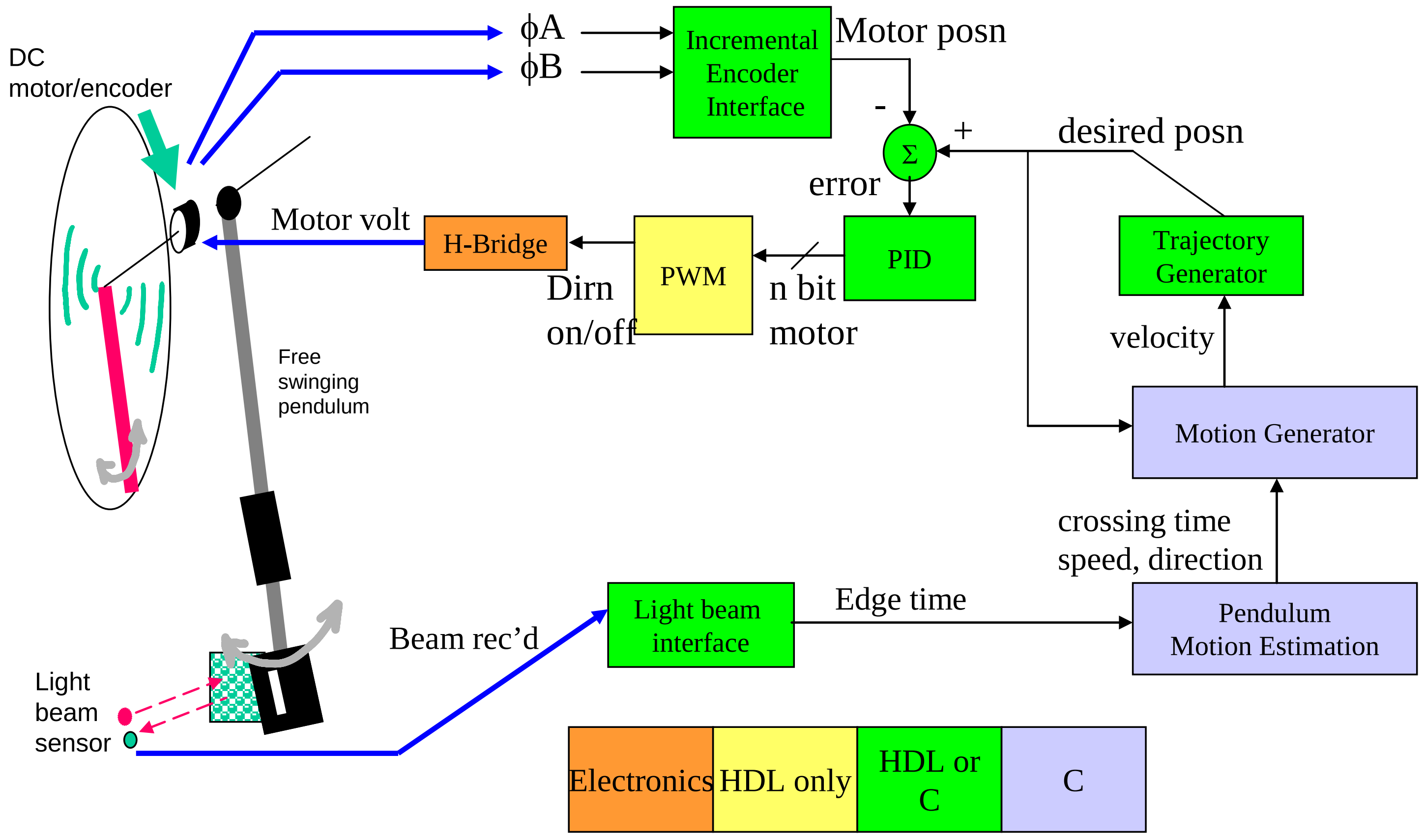
The ribbon cable should be connected to the inner GPIO header (marked JP1 on the DE2 board).

Pin Assignments:

```
(* chip_pin = "D25" *) output Hbridge_InA;  //GPIO_0[0]
(* chip_pin = "J22" *) output  Hbridge_InB;   //GPIO_0[1]
(* chip_pin = "E26" *) output Hbridge_PWM; //GPIO_0[2]

(* chip_pin = "E25" *) input encoderAin; //GPIO_0[3]
(* chip_pin = "F24" *) input encoderBin; //GPIO_0[4]
(* chip_pin = "F23" *) input beam;          //GPIO_0[5]
```

Appendix B –System Level Overview.


DC motor/encoder
ϕA
ϕB
Incremental Encoder Interface
Motor posn
-
+
Σ
desired posn
error
Motor volt
H-Bridge
PWM
PID
Trajectory Generator
Dirn on/off
n bit motor
velocity
Free swinging pendulum
Motion Generator
crossing time speed, direction
Light beam interface
Edge time
Pendulum Motion Estimation
Beam rec'd
Light beam sensor
Electronics
HDL only
HDL or C
C