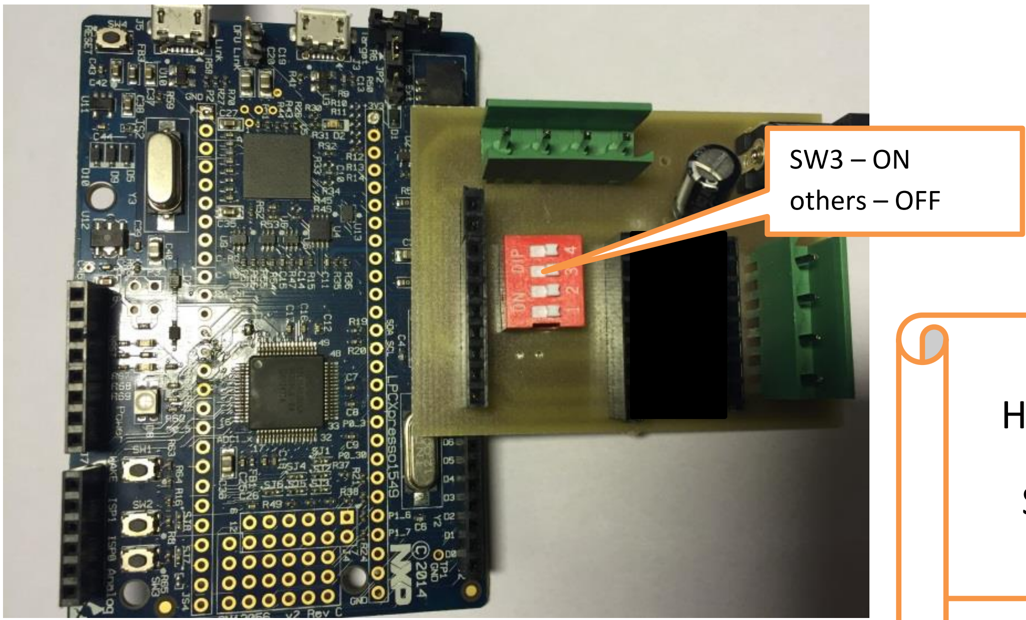
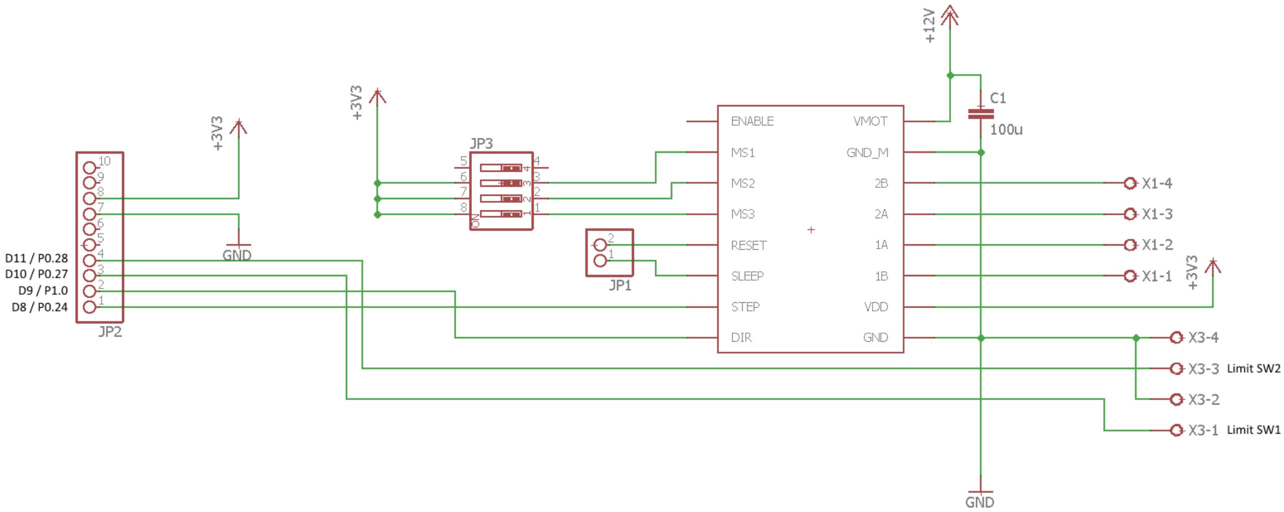
Stepper motors and interrupts

This lab requires the stepper motor add on board. For the stepper motor exercises you need to set SW3 to ON and all other switches to OFF. The setting configures stepper controller to drive the motor half step per pulse at step input.



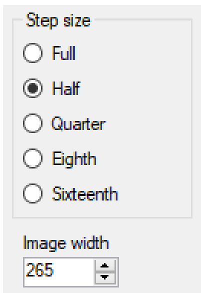
Simulator:
Half step mode

Set width to:
265



If you do this exercise with signal capture board and simulator then the pins to use are:

- Step P0.24
- Dir P1.0
- Limit SW1 P0.9
- Limit SW2 P0.29
- Button1 P0.8
- Button2 P1.6
- Button3 P1.8



In the following exercises you will learn how synchronize tasks and ISRs. Use FreeRTOS ISR based UART communication from SerialPort_FreeRTOS_ISR.zip. This implementation is thread safe and has built-in mutexes to protect reads and writes so no additional locking is needed. Note that the object can be constructed before scheduler is started but all other member function calls require task context. There is a simple example about setting up and using a UART in the zip file.

Note that when you use ISR based UART communication on the debug UART you MUST NOT call any Board_UART...() functions or use DEBUGOUT()

```
// transmit pin that goes to debugger's UART->USB converter
LpcPinMap txpin1 = { .port = 0, .pin = 18 };
// receive pin that goes to debugger's UART->USB converter
LpcPinMap rxpin1 = { .port = 0, .pin = 13 };


class LpcUart {
public:
LpcUart(const LpcUartConfig &cfg);
LpcUart(const LpcUart &) = delete;
virtual ~LpcUart();
int  free(); /* get amount of free space in transmit buffer */
int  peek(); /* get number of received characters in receive buffer */
```

Check buffer status.Do not use

```
int  write(char c);
int  write(const char *s);
int  write(const char *buffer, int len);
```

All writes return number of bytes written

```
int  read(char &c); /* get a single character. Returns number of characters read --> returns
0 if no character is available */
int  read(char *buffer, int len);
```

Reads without timeout are polling. Do not use these

ic_timeout is maximum time between two received characters. Useful only if your communication protocol requires it. For user interaction leave at default.

```
int  read(char *buffer, int len, TickType_t total_timeout, TickType_t ic_timeout =
portMAX_DELAY);
```

Always use read with timeout. Let OS handle the waiting! This timeout specifies how long read will for the requested number of characters. Read returns number of characters received which may be less than requested if timeout occurs. On timeout number of received characters can be [0 – len-1].

Exercise 1

In this exercise you will use RIT (Repetitive Interrupt Timer) create stepper motor driving pulses. Write a program that reads commands from serial port and executes them. The stepper motor must be driven using RIT ISR. The program must support the following commands:

- left <count> - runs <count> steps to left, <count> is an integer
- right <count> - runs <count> steps to right, <count> is an integer
- pps <count> - sets the number of driving pulses per second ie. sets the speed of the stepper motor. This setting must remain in effect until changed by another pps command.

The ISR must stop driving immediately if a limit switch is closed. The default pps value must be set to a value that makes stepper run slowly. Our motor's nominal step is 1.8 degrees (200 steps / revolution). With half stepping we need 400 pulses per revolution (0.9 degrees per (half) step).

The following code examples can be used as a starting point for this exercise. Modify the code to suit your needs. Use a binary semaphore to synchronize task and RIT ISR. Make sure that your binary semaphore is created in an "empty" state. See FreeRTOS API documentation for details.

The example code shows an example how we can set up an ISR to perform as series of operations and wait for ISR to notify the task when operation is complete. In the code the RIT is first set up and after the setup the timer is started and RIT interrupt is enabled. In the example the ISR simply decrements a counter on each interrupt and when the counter reaches zero the timer is stopped and a binary semaphore is given to notify task that the operation has completed.

The example code does not use critical sections since RIT interrupt is disabled during the setup which prevents ISR from modifying the shared variable. While RIT is running the task is blocked on the binary semaphore which prevents the task from reconfiguring RIT or modifying the variable. Note that in the example RIT_start is not protected by a mutex which means that only one task may call RIT_start. If the function needs to be called from multiple task then the function must be protected by a mutex (take at the beginning, give back at the end).

The following needs to be accessible by both ISR and RIT_start()

```
volatile uint32_t RIT_count;
xSemaphoreHandle sbRIT;
```

RIT needs to be initialized and interrupt priority needs to be set during the hardware setup.

```
// initialize RIT (= enable clocking etc.)
Chip_RIT_Init(LPC_RITIMER);

// set the priority level of the interrupt
// The level must be equal or lower than the maximum priority specified in FreeRTOS config
// Note that in a Cortex-M3 a higher number indicates lower interrupt priority
NVIC_SetPriority( RITIMER_IRQn, configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY + 1 );
```

ISR must always be wrapped inside 'extern "C" 'if they appear inside a C++ file. Startup files that place the ISR handlers in memory require C-style internal naming for ISRs to be placed in right place (and to work properly).

Only the FreeRTOS functions that end with ISR may be called from an interrupt handler or a function that is called by the interrupt handler. Keep the ISR as simple as possible. Usually all of the time consuming work can be done in a task.

```
extern "C" {
void RIT_IRQHandler(void)
{
  // This used to check if a context switch is required
  portBASE_TYPE xHigherPriorityWoken = pdFALSE;
  // Tell timer that we have processed the interrupt.
  // Timer then removes the IRQ until next match occurs
  Chip_RIT_ClearIntStatus(LPC_RITIMER); // clear IRQ flag

  if(RIT_count > 0) {
      RIT_count--;
      // do something useful here...
      }
      else {
      Chip_RIT_Disable(LPC_RITIMER); // disable timer
      // Give semaphore and set context switch flag if a higher priority task was woken up
      xSemaphoreGiveFromISR(sbRIT, &xHigherPriorityWoken);
  }
  // End the ISR and (possibly) do a context switch
  portEND_SWITCHING_ISR(xHigherPriorityWoken);
}
}
```

The following function sets up RIT interrupt at given interval and waits until count RIT interrupts have occurred. Note that the actual counting is performed by the ISR and this function just waits on the semaphore.

```
void RIT_start(int count, int us)
{
  uint64_t cmp_value;

  // Determine approximate compare value based on clock rate and passed interval
  cmp_value = (uint64_t) Chip_Clock_GetSystemClockRate() * (uint64_t) us / 1000000;

  // disable timer during configuration
  Chip_RIT_Disable(LPC_RITIMER);

  RIT_count = count;
  // enable automatic clear on when compare value==timer value
  // this makes interrupts trigger periodically
  Chip_RIT_EnableCompClear(LPC_RITIMER);
  // reset the counter
  Chip_RIT_SetCounter(LPC_RITIMER, 0);
  Chip_RIT_SetCompareValue(LPC_RITIMER, cmp_value);
  // start counting
  Chip_RIT_Enable(LPC_RITIMER);
  // Enable the interrupt signal in NVIC (the interrupt controller)
  NVIC_EnableIRQ(RITIMER_IRQn);

  // wait for ISR to tell that we're done
  if(xSemaphoreTake(sbRIT, portMAX_DELAY) == pdTRUE) {
      // Disable the interrupt signal in NVIC (the interrupt controller)
      NVIC_DisableIRQ(RITIMER_IRQn);
  }
  else {
      // unexpected error
  }
}
```

Exercise 2

As specified in the exercise 1 but with the following additions:

- When the program starts the belt connector block is driven in the middle of the aluminium rail
- Program stores commands in a queue and starts executing them (left/right/pps) when command "go" is given.
- While commands in the queue are being executed new movement commands may be added to the queue. If the queue is full user is notified with "command queue full"
- If "stop" is entered while commands are being executed the currently executing command is finished and then executing stops until "go" is entered again. When "go" is entered again the execution continues with the next command(s) in the queue.

Hints:

You will need at least two tasks: one for reading commands and other for executing commands from the queue.

Put new movement/speed commands at the end of the queue and "stop" at the front of the queue.

Do not put "go" in the queue at all. Instead make the processing task to wait on a binary semaphore that you give when "go" is entered.

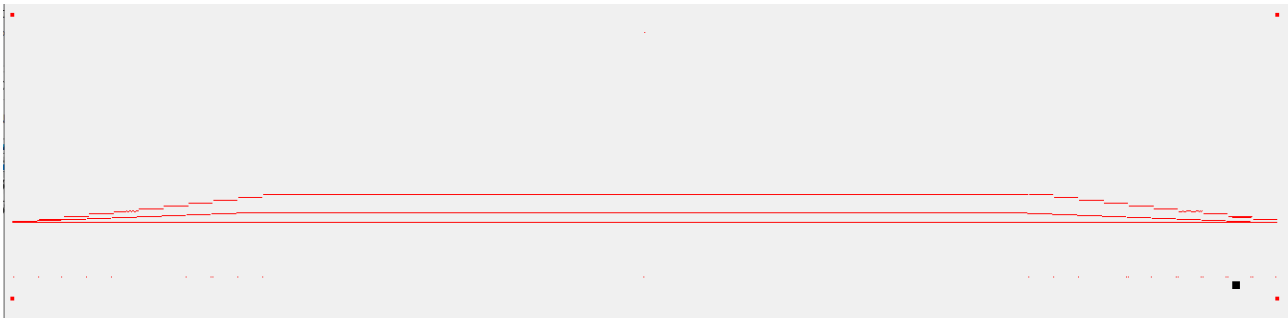
Exercise 3

Write a program that runs the stepper motor between the limit switches as fast as possible. Make your program first do a couple of test runs at slow speed to determine the number of steps between the switches. Then the program starts running the stepper motor between the switches so that it starts from a position where a limit switch is closed and ends the run in a position where the other limit switch is closed. The belt connector block may not hit the body of the switch. The motor must stop so that the switch is just closed. The switches are used to measure the run time of the motor. Running must be based on counting the steps between the switches.

On each high speed round the motor maximum speed is increased until the runtime no longer improves. If the limit switch is not hit after the calculated number of pulses has been given the test ends. Not hitting the switch means that motor is missing steps which means that the stepper can't keep up with the pace.

Important note: at high speeds you can't start/stop directly at maximum speed but you need to start with slower stepping and accelerate gradually to the required speed and then decelerate to low speed before stopping. Use 10-20% of total steps for ramping at both ends. A safe starting speed is 500 pps (2 ms period).

Implement two tasks that switch red/green led on for 500 ms when a limit switch is closed. Red led for the left limit switch and green led for the right.



Simulator (v1.2 and above) plots the pulse intervals above the black dot as it moves. The example above shows example of static speed run (lowest line) and two ramping runs. The higher the line is the shorter the interval between pulses. Note that simulator can reach unrealistically high speeds so be prepared to see lower values when you test on the real hardware.

At the end of the test the program reports the following based on the fastest successful run between the limit switches:

- maximum pps value
- maximum rpm value
- fastest time between two limits