CG2271 Real-time Operating Systems

Owen Leong
owenleong@u.nus.edu

October 7, 2023

1 GPIO

```
/** GPIO - Register Layout Typedef */
typedef struct {
__IO uint32_t PDOR; /**< Port Data Output Register, offset: 0x0 */
__O uint32_t PSOR; /**< Port Set Output Register, offset: 0x4 */
__O uint32_t PCOR; /**< Port Clear Output Register, offset: 0x8 */
__O uint32_t PTOR; /**< Port Toggle Output Register, offset: 0xC */
__I uint32_t PDIR; /**< Port Data Input Register, offset: 0x10 */
__IO uint32_t PDDR; /**< Port Data Direction Register, offset: 0x14 */
} GPIO_Type;
```

1.1 Bit Operations

```
n = MASK(foo); // overwrite n with mask
n |= MASK(foo); // set bit foo
~MASK(foo); // complement bit value of mask
n &= ~MASK(foo); // clear bit foo
```

1.2 PCR

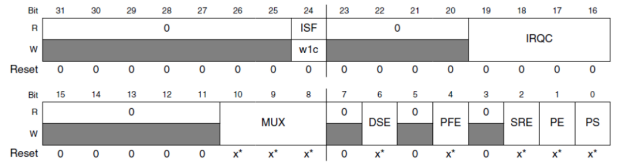


Figure 1: Pin Control Register

- PE: Pull enable. 1 to enable the pull resistor
- PS: Pull select. 1 for pull up, 0 for pull down
- DSE: 1 to drive more current
- MUX: To select configuration

1.3 Code

```
void InitGPIO(void)
{
  // Enable Clock to PORTB and PORTD
  SIM->SCGC5 |= SIM_SCGC5_PORTB_MASK);
  PORTB->PCR[LED] &= ~PORT_PCR_MUX_MASK;
  PORTB->PCR[LED] |= PORT_PCR_MUX(1);
  // Data Direction Registers for PortB
  PTB->PDDR |= MASK(LED); // 1 for output,
        ↪  0 for input
}
```

```
void led_control(color_t color) {
  if (color == RED) {
    PTB->PDOR &= ~MASK(LED);
  } else {
    PTB->PDOR |= MASK(LED);
  }
  // also PTOR (toggle), PCOR (clear),
        ↪ PSOR (set)
}
```

2 Interrupts

- `NVIC_SetPriority(IRQnum, priority)` priority 0, 1, 2 or 3
- Fixed priorities: Reset -3, NMI (non-maskable interrupt) -2, Hard fault -1
- New priority higher than current priority, then preempty current exception handler
- New priority lower or equal to current, then held in pending state
- Use volatile
- Entry in Vector table is a pointer to the Interrupt Handler function
- `NVIC_EnableIRQ(IRQnum)`, `NVIC_DisableIRQ(IRQnum)`, `NVIC_SetPendingIRQ(IRQnum)`, `NVIC_ClearPendingIRQ(IRQnum)`

2.1 Nested Vectored Interrupt Controller

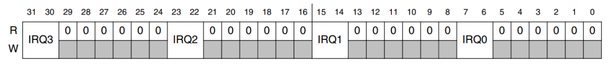


Figure 2: IPR0

Each IPR0-7 lets you configure 4 interrupt priorities, using 2-bits each, corresponding to the four priority levels.

2.2 IRQC in PCR

```
0000 // Interrupt disabled
1000 // Interrupt when logic zero
1001 // Interrupt on rising edge
```

```
1010 // Interrupt on falling edge
1011 // Interrupt on either edge
1100 // Interrupt when logic one
```

2.3 Code

ISFR: Interrupt status flag register (one per port), set to 1 if interrupt has been detected

```
void InitSwitch() {
  PORTD->PCR[SW_POS] &= ~PORT_PCR_MUX_MASK; // GPIO mode
  PORTD->PCR[SW_POS] |= PORT_PCR_MUX(1);
  PORTD->PCR[SW_POS] |= PORT_PCR_PE_MASK; // enable pullup/pulldown
  PORTD->PCR[SW_POS] |= PORT_PCR_PS(1); // use pullup
  PORTD->PCR[SW_POS] &= ~PORT_PCR_IRQC_MASK;
  PORTD->PCR[SW_POS] |= PORT_PCR_IRQC(10); // falling edge
  PTD->PDDR &= ~MASK(SW_POS); // set as input
  NVIC_EnableIRQ(PORTD_IRQn);
}
void PORTD_IRQHandler() {
  NVIC_ClearPendingIRQ(PORTD_IRQn); // clear pending interrupts
  if ((PORTD->ISFR & MASK(SW_POS))) {
    // process interrupt
    PORTD->ISFR |= MASK(SW_POS); // clear status flag
  }
}
```

3 Analog-to-Digital and Digital-to-Analog

4 Timers

- PIT: Periodic Interrupt Timer
- TPM: Timer/PWM Module
- LPTMR: Lower-Power Timer
- Real-Time clock
- SysTick

4.1 TPM: Timer/PWM Module

- Prescalar divides clock, possible values `1, 2, 4, ..., 128`
- 6 channels for each timer
- 3 Modes: Capture Mode, Output Compare, PWM

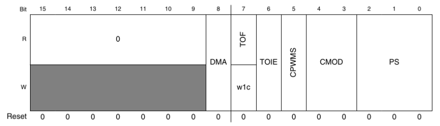


Figure 3: TPMx_SC

```
void InitPWM() {
  SIM_SCGC5 |= SIM_SCGC5_PORTB_MASK;
  PORTB->PCR[PTB0_Pin] &= ~PORT_PCR_MUX_MASK;
  PORTB->PCR[PTB0_Pin] |= PORT_PCR_MUX(3);
  PORTB->PCR[PTB1_Pin] &= ~PORT_PCR_MUX_MASK;
  PORTB->PCR[PTB1_Pin] |= PORT_PCR_MUX(3);
  SIM->SCGC6 |= SIM_SCGC6_TPM1_MASK; // enables clock to TPM1
  SIM->SOPT2 &= ~SIM_SOPT2_TPMSRC_MASK;
  SIM->SOPT2 |= SIM_SOPT2_TPMSRC(1); // use the MCGFLLCLK clock
  TPM1->CNT = 0;
  TPM1->MOD = 7500; // modulo value for LPTPM counter, write to CNT before writing to
        ↪ MOD
  TPM1_COV = 7500 / 2; // TPMx_CnV is channel (n) value
  TPM1->SC &= ~((TPM_SC_CMOD_MASK) | (TPM_SC_PS_MASK));
  TPM1->SC |= TPM_SC_CMOD(1); // CMOD is clock mode, 1 for increment counter on every
        ↪ counter clock
  TPM1->SC |= TPM_SC_PS(7); // PS is prescale factor selection, 7 for divide by 128
  TPM1->SC &= ~(TPM_SC_CPWMS_MASK); // CPWMS is center-aligned PWM select, 0 for up (
        ↪ edge aligned), 1 for up-down mode (center-aligned)
  // TPMx_CnSC is channel (n) status & control
  TPM1_COSC &= ~((TPM_CnSC_ELSB_MASK) | (TPM_CnSC_ELSA_MASK) | (TPM_CnSC_MSB_MASK) | (
        ↪ TPM_CnSC_MSA_MASK));
  TPM1_COSC |= (TPM_CnSC_ELSB(1) | TPM_CnSC_MSB(1));
}
```

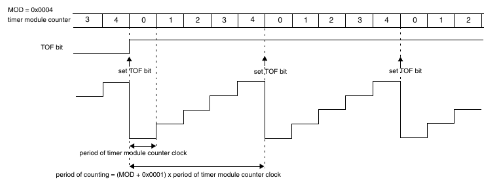

MOD = 0x0004
timer module counter
TOF bit
set TOF bit
period of timer module counter clock
period of counting = (MOD + 0x0001) x period of timer module counter clock

Figure 4: TPM Up-Counting

- EPWM (edge-aligned PWM mode), period is determined by `MOD + 1`, duty cycle determined by `CnV`. For `ELSnB:ELSnA = 1:0`, channel output is forced high at counter overflow, low at channel match (TPM counter = CnV).

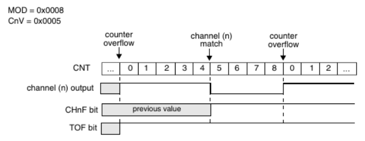


Figure 5: PWM Edge Aligned

- CPWM (center-aligned PWM mode), TPM counter counts up to MOD then counts down to 0. Period determined by `2 x MOD`, duty cycle determined by `2 x CnV`

CPWMS	MSnB:MSnA	ELSnB:ELSnA	Mode	Configuration
X	00	00	None	Channel disabled
X	01/10/11	00	Software compare	Pin not used for LPTPM
0	00	01	Input capture	Capture on Rising Edge Only
		10		Capture on Falling Edge Only
		11		Capture on Rising or Falling Edge
	01	01	Output compare	Toggle Output on match
		10		Clear Output on match
		11		Set Output on match
	10	10	Edge-aligned PWM	High-true pulses (clear Output on match, set Output on reload)
		X1		Low-true pulses (set Output on match, clear Output on reload)
	11	10	Output compare	Pulse Output low on match
		X1		Pulse Output high on match
1	10	10	Center-aligned PWM	High-true pulses (clear Output on match-up, set Output on match-down)
		X1		Low-true pulses (set Output on match-up, clear Output on match-down)

Figure 6: Timer Modes

5 UART

- Universal Asynchronous Receiver/Transmitter
- Transmits low order bits first

```
void InitUART2 (uint32_t baud_rate)
{
  SIM->SCGC4 |= SIM_SCGC4_UART2_MASK;
  SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;
  PORTE->PCR[UART_TX] &= ~PORT_PCR_MUX_MASK;
  PORTE->PCR[UART_TX] |= PORT_PCR_MUX(4);
  PORTE->PCR[UART_RX] &= ~PORT_PCR_MUX_MASK;
  PORTE->PCR[UART_RX] = PORT_PCR_MUX(4);
  NVIC_SetPriority(UART2_IRQn, 2);
  NVIC_ClearPendingIRQ(UART2_IRQn);
  NVIC_EnableIRQ(UART2_IRQn);
  UART2->C2 &= ~((UART_C2_TE_MASK) | (UART_C2_RE_MASK)); // disable transmitting and
      ↪ receiving while configuring
  UART2->C2 |= UART_C2_TIE_MASK | UART_C2_RIE_MASK; // transmit interrupt enable,
      ↪ receiver interrupt enable
  uint32_t bus_clock = (DEFAULT_SYSTEM_CLOCK) / 2;
  uint32_t divisor = bus_clock / (baud_rate * 16); // 16 because 16x oversampling
  UART2->BDH = UART_BDH_SBR(divisor >> 8);
  UART2->BDL = UART_BDL_SBR(divisor);
  UART2->C1 = 0; // can set parity mode with this
  UART2->C2 |= ((UART_C2_TE_MASK) | (UART_C2_RE_MASK));
  UART2->C3 = 0;
  UART2->S2 = 0;
}
```

5.1 Interrupt Handler

```
void UART2_IRQHandler() {
  NVIC_ClearPendingIRQ(UART2_IRQn);
  if (UART2->S1 & UART_S1_TDRE_MASK) { // transmit data register empty
    UART2->D = ...; // UART data register
  }
  if (UART2->S1 & UART_S1_RDRF_MASK) { // receive data register full
    data = UART2->D;
  }
  if (UART2->S1 & (UART_S1_OR_MASK | UART_S1_NF_MASK | UART_S1_FE_MASK | UART_S1_PF_MASK
      ↪ )) { // handle the error
  }
}
```

5.2 Polling

```
void UART2_Transmit_Poll(uint8_t data) {
    // wait until transmit data register is empty
    while (!(UART2->S1 & UART_S1_TDRE_MASK));
    UART2->D = data;
}

uint8_t UART2_Receive_Poll() {
  while (!(UART2->S1 & UART_S1_RDRF_MASK));
  return UART2->D;
}
```

6 RTOS

6.1 Process State Transitions

```
    New -----Admit------> Ready
  Ready ----Dispatch----> Running
Running ----Time-out----> Ready
```

```
Running -----Release----> Exit
Running ---Event-wait---> Blocked
Blocked --Event-occurs--> Ready
```

7 Tutorial 1

7.1 Event-triggered scheduling with Interrupts

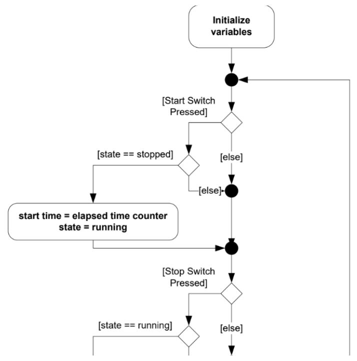
```
function main() {
    state = stopped;
    display_delay = 100;
    elapsed_time = 0;
}
function timerISR() {
    if (state == running) {
        elapsed_time += 1 ms;
        display_delay -= 1;
        if (display_delay == 0) {
            display_delay = 100;
            display(elapsed_time);
        }
    }
}
```

```
function startISR() {
    state = running;
}
function stopISR() {
    state = stopped;
}
function clearISR() {
    if (state == stopped) {
        elapsed_time = 0;
    }
}
```

7.2 Static Scheduler without Interrupts

```
state = stopped;
display(elapsed_time_counter);
next_display_update = elapsed_time_counter + 100;
while (1) {
    if (start_switch_pressed()) {
        if (state == stopped) {
            start_time = elapsed_time_counter;
            state = running;
        }
    }
    if (stop_switch_pressed()) {
        if (state == running) {
            stop_time = elapsed_time_counter;
            state = stopped;
        }
    }
    if (clear_switch_pressed()) {
        if (state == stopped) {
            start_time = stop_time;
        }
    }
    if (elapsed_time_counter > next_display_update) {
        if (state == running) {
            display(elapsed_time_counter - start_time);
        }
        else {
            display(stop_time - start_time);
        }
        next_display_update = next_display_update + 100;
    }
}
```

7.3 Flowchart



7.4 Scheduling

7.4.1 Static, non-preemptive scheduler

Base case: Task C starts immediately. $T_r = 0 + 2 = 2$
Worst case: Task A and Task B run first. $T_r = 0+3+1+2 = 6$

7.4.2 Dynamic, non-preemptive scheduler

Best case: Task C starts immediately.
Worst case: Longest task A just started running ϵ time units ago, $T_r = 0 + 3 - \epsilon + 2 = 5 - \epsilon$

7.4.3 Dynamic, preemptive scheduler

Best case: Task C starts immediately.
Worst case: Longest task just started running ϵ time units ago, but is preempted by C. $T_r = 0 + 2 = 2$

8 Tutorial 2

8.1 IRQ0, IRQ10, IRQ31

`DMA0_IRQn, SPI0_IRQn, PORTD_IRQn`

8.2 Priority levels to order IRQs

8.2.1 Interrupts in the order IRQ10, IRQ0, IRQ31

```
NVIC_SetPriority(SPI0_IRQn, 1); // set NVIC_IPR2[23:22] = 01
NVIC_SetPriority(DMA0_IRQn, 2); // set NVIC_IPR0[7:6] = 10
NVIC_SetPriority(PORTD_IRQn, 3); // set NVIC_IPR7[31:30] = 11
```

8.2.2 Enable IRQ13 but disable IRQ24

```
NVIC_EnableIRQ(13); // Set NVIC_ISER[13] to 1
NVIC_DisableIRQ(24); // Set NVIC_ICER[24] to 1
```

8.3 Clock 48 MHz, interrupt at 10 kHz, ISR takes 14.9 us, 1 us overhead

8.3.1 Percentage of processor time spent servicing interrupts incl. overhead

$$\frac{14.9\mu s+1\mu s}{1/10kHz} = \frac{15.9\times10^{-6}}{10^{-4}} = 15.9\times10^{-2} = 15.9\%$$

8.3.2 Percentage left for main loop

$100\% - 15.9\% = 84.1\%$

8.3.3 Min main loop rate if main loop requires 37 ms

Time for one loop including interrupts $= \frac{37ms}{84.1}\times 100 = 44ms$
Update rate $= \frac{1}{44ms} = 22.7Hz$

8.4 Interrupt at 25 kHz, ISR takes 34 us, 1 us overhead

8.4.1 Percentage of time on interrupt and overhead

$$\frac{34\mu s+1\mu s}{1/25kHz} = 35\mu s\times 25kHz = 87.5\%$$

8.4.2 Percentage for main loop

$100\% - 87.5\% = 12.5\%$

8.4.3 Min main loop rate if main loop requires 37 ms

Time for one loop $= \frac{37ms}{12.5}\times 100 = 296ms$
Update rate $= \frac{1}{296ms} = 3.38Hz$

9 Tutorial 3

9.1 Generate a 50% duty PWM using Periodic Interrupt Timer clocked by 20 MHz clock

9.1.1 Pseudocode

```
function main() {
    set PWM_GPIO pin to 0
    set Timer start value to 0x7FFF
    enable interrupts for Timer
    start Timer
    while (1) {}
}
function Timer_ISR() {
    toggle PWM_GPIO pin
}
```

9.1.2 Period of the PWM waveform

Period of clock $= \frac{1}{20MHz} = 50\times10^{-9} = 50ns$
Interval between ISR $= 50ns\times(0x7FFF+1) = 50ns\times 32768 = 1.64ms$
Period of PWM $= 2\times 1.64ms = 3.28ms$

9.1.3 No timer module but generate PWM

```
function main() {
    set PWM_GPIO to 0
    int counter = 0x7FFF
    while (1) {
        counter--;
        if (counter == 0) {
            toggle PWM_GPIO
            counter = 0x7FFF
        }
    }
}
```

9.2 Serial Interrupts to capture data through bluetooth

```
volatile char rx_data; // Global Variable
Serial_ISR {
    rx_data = Serial_Read_Buffer();
    rx_new_data = 1;
}
Main() {
    if(rx_new_data == 1) {
        rx_new_data = 0;
        if(rx_data == 0x00)
            move_robot_forward();
        if(rx_data == 0x01)
            move_robot_right();
        if(rx_data == 0x02)
            move_robot_left();
        else
            stop_robot();
    }
    else
        Do_Other_Things();
}
```

9.2.1 Describe some issues with the implementation above

- rx_data is not processed immediately when it is received, so it may be overwritten if new data comes in before it is processed
- Needs to finish Do_Other_Things before it can process rx_data

9.2.2 rx_data is declared as volatile. How does it affect the behaviour?

- Everytime rx_data is accessed, its value is loaded from memory, so if the interrupt is triggered while Main is in the if statement, the latest rx_data will be used

9.3 Use Circular Queue

```
//Queue declared with a size of 10 (characters)
Serial_ISR {
    rx_data = Serial_Read_Buffer();
    if(!Queue_Full())
        Q_Enqueue(rx_data);
}
Main() {
    if(!Queue_Empty()) {
        my_data = Queue_Dequeue();
        if(my_data == 0x00)
            move_robot_forward();
        if(my_data == 0x01)
            move_robot_right();
        if(my_data == 0x02)
            move_robot_left();
        else
            stop_robot();
    }
    else
        Do_Other_Things();
}
```

9.3.1 What does the new implementation resolve? Are there still things to be concerned about?

- Solves the issue of overriding the same variable and losing information.
- Response is still not immediate.
- Potential issue of data being lost if interrupts occur too frequently and buffer gets full before data is done being processed.

9.3.2 How can we resolve this?

- Can use ability in ARM to generate an INT request once new data has been updated to the circular queue, so another ISR can process it immediately.

10 Tutorial 4

10.1 State Transitions

10.1.1 Task A is running and chooses to give up CPU voluntarily.

It goes into Ready state.

10.1.2 Task A is running and a higher priority task B becomes ready.

Task A goes into ready state, task B goes into running state.

10.1.3 After some time, task B requests for some resource and is unable to acquire it.

Task B goes into blocked. Task A goes into running.

10.1.4 After 5 ms, the resource acquired by task B is available.

Task B goes from blocked state to ready state, if there is no higher priority task running it transitions into running state and task A goes into ready state.

10.2 Creating a task in RTX

10.2.1 What are the three parameters than osThreadNew takes?

- First argument is a function for the thread
- Second argument is a point passed as an argument for the function
- Third argument is thread attributes

10.2.2 When will app_main() be called?

Once the osKernelStart() function is called, the multi-threaded environment has started. At that time, there can be many tasks that are in the Ready state, the one with the highest priority will run.

10.2.3 Why is there a need for for(;;) loop in the app_main?

In embedded operating systems, that is how most tasks are run.

10.3 Blinky

10.3.1 When we call osDelay() what happesn to the app_main() task?

Transitions to Blocked state for the duration of 1000 ticks, corresponding to a delay of 1s by default.

10.3.2 What will the CPU execute during that delay time?

The OS has its own idle thread that is not shown to the developer. This thread will be run if there are no other threads that can make use of the CPU.

10.3.3 If we use a normal delay() routine, will we see the same effect?

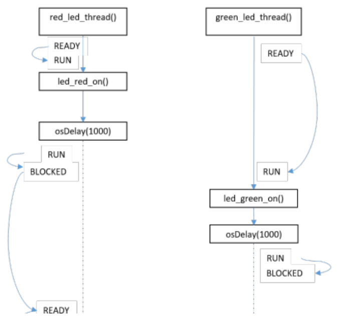
With osDelay() CPU is freed up to do other aactivities, with normal delay() the CPU is still used to execute code in order to generate the required delay.

10.4 Double Blinky

10.4.1 What is the expected behaviour?

Both red and green LEDs will light up together, because of the context switching that happens when we call osDelay().

10.4.2 Draw timeline



11 Midterms AY19/20 Sem 1

11.1 Objective of LINE 2

0x0A | 0x01 corresponds to bit 0, 1 and 3. Referring to the datasheet, pins 16-19 correspond to interrupt configuration. The bit pattern is 1011, corresponding to interrupt on either edge. Hence answer is (d).

11.2 Objective of LINE 3

0xFE is equal to 0x01. Pins 8-10 correspond to MUX control. The bit pattern 001 is equal to 1, for Port D pin 6, corresponds to GPIO mode. Hence answer is (d).

11.3 How many interrupts generated?

Signal contains both a falling edge and a rising edge, so answer is (b) for 2 interrupt requests.

11.4 What is the value in IPR7 given Port D is mapped to IRQ 31?

IPR[31:30] would be set to 0:1. Leftmost hex character are for bits 28-31, which has bit pattern 0100 which is 0x4. So answer is (d) 0x40000000.

11.5 Which of the following correctly configures IRQ priority level?

(c). Leftshift by 30.

11.6 Resistor configuration for pull-up switch.

(b). When pressed, connected to ground so goes to zero, when unpressed, pulled up to V_{CC} by the pull-up resistor.

11.7 PIT module

S1 is true. S2 is true. S3 is false because PIT always counts down. Answer is (b).

11.8 PIT module generating interrupts

S1 is possible. S2 is possible if push button IRQ handler has higher priority. S3 is possible. Answer is (e).

11.9 Which is true for volatile global variable?

S1 is false, will re-fetch data. S2 is true. S3 is true. So answer is (c).

11.10 UART2 module configured for both Transmit and Receive data

S1 is false, only one IRQ handler. S2 is true. S3 is true. Answer is (b).

11.11 T1 is running and gets switched to ready.

(e). None of the statements valid. (c) and (d) will move T1 to blocked state.

11.12 T1 is running and gets switched to blocked.

(b). Attempted to acquire a resource but was unable to get it.

11.13 Task is in running with all required resources. Scheduler will perform context switch for equal-priority tasks

(a). True.

11.14 T1, T2, T3, T4 in order of priority. T1 is running most of the time, T2 and T3 only go to running once. T4 running occasionally.

S1 is false since T1 may not always be in running state. S2 cannot be true since otherwise T4 will never be run. S3 is possible. S4 is possible. Answer is (d).

11.15 Two threads in main()

S1 is false, GPIO not used by scheduler. S2 is false, red thread will run first. S3 false, multi-threaded environment will only start in osKernelStart. Answer is (e).