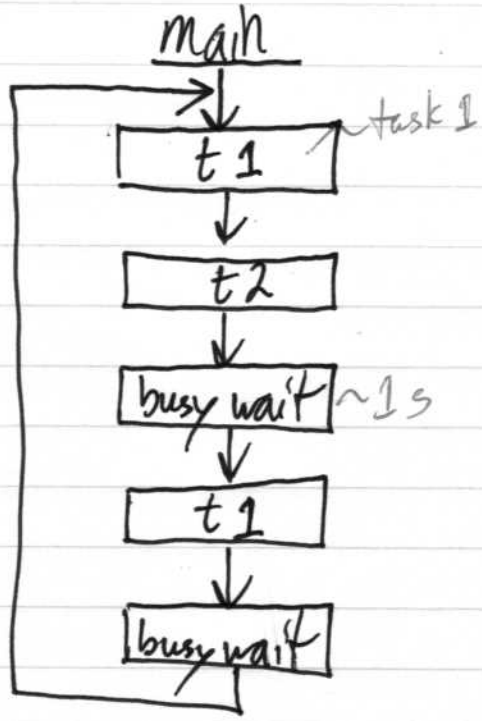
Tasks

1) Bare Metal (no OS)

- some real-time systems are constructed with no OS
- this can be done using a "superloop"
- e.g. task 1 executes at 1 Hz
  task 2 executes at 0.5 Hz



superloop

```
#include <stdbool.h>
#include <stdint.h>

void task1(void);
void task2(void);

int main(void){

    while(true){
        task1();
        task2();

        for(uint32t spin=0; spin<17000000; spin++)
            ;

        task1();

        for(uint32t spin=0; spin<17000000; spin++)
            ;
    }
}
```

do some work

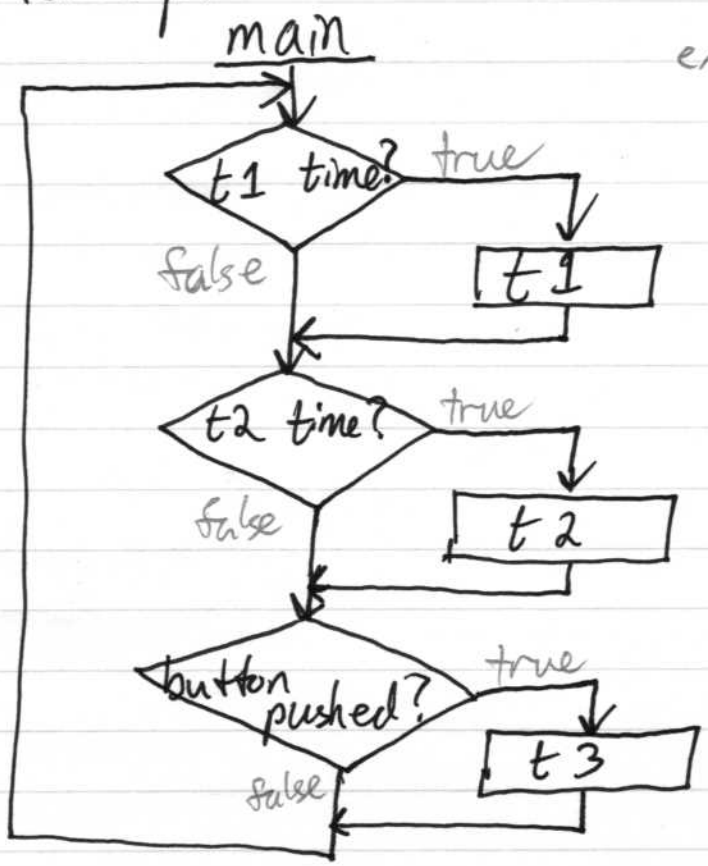
not actually tasks yet

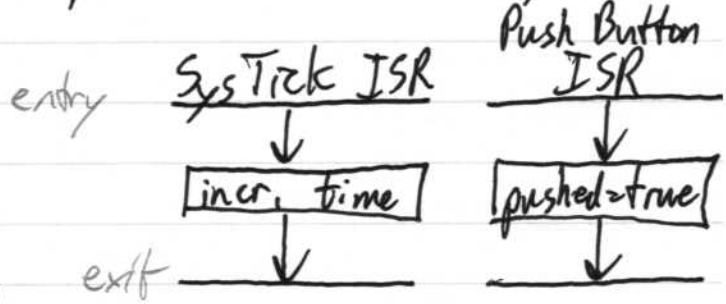
busy wait for ~1s

real-time programs don't terminate

- task1() and task2() are synchronous
  - they execute periodically
- shortcomings:
  - timing is approximate
  - cannot respond quickly to asynchronous events (IRQs)

- a better method is to keep time with the SysTick interrupt





this example expands on the previous one by adding an asynchronous Push Button input

- the push button is labelled INT0 in the lab and is on GPIO Port 2, Pin 10

```
#include <LPC17xx.h>
#include <stdbool.h>
#include <stdint.h>

uint32_t  msTick = 0;  millisecond tick count
bool      buttonPushed = false;

void  SysTick_Handler(void){
        msTick++;
}
```

this overrides the default SysTick ISR found in the system startup files

later

```
void setupINT0(void){
  LPC_GPIOINT->IO2IntEnF |= (1<<10);   enable push button interrupt on falling edge at the device
  NVIC_EnableIRQ(EINT3_IRQn);   allow IRQ at the processor
}

void  EINT3_IRQHandler(void){   overrides default INT0(push button) ISR
    buttonPushed = true;
    LPC_GPIOINT->IO2IntClr |= (1<<10);   clear current IRQ
}

void  task1(void);
void  task2(void);
void  task3(void);
```

```
int main(void) {
    uint32_t  t1Next = 0;  } when to run each task
    uint32_t  t2Next = 0;  }

                                    ticks per second
    setupINT0();
    SysTick_Config(SystemCoreClock / 1000 );
                   interrupt every 1ms

    while (true) {
        if(msTicks >= t1Next)  fails when msTicks
                                overflows

        if(msTicks - t1Next >= 1000)
            task1();
            t1Next += 1000;  add 1s
        }
        if(msTicks - t2Next >= 2000)
            task2();
            t2Next += 2000;  add 2s
        }
        if(buttonPushed) {
            buttonPushed = false;
            task3();
        }
    }
}
```

- this example is not a true superloop; it is an example of a "roll-your-own" scheduler

- aside: unsigned arithmetic wraps modulo 2^n in C
  e.g. 8-bit unsigned: range $\in [0, 255]$, $2^n = 256$
  $(255+1) \% 256 = 0$
  $(0 - 1) \% 256 = 255$ (add divisor if dividend is -ve)

e.g. uint8_t msTicks = 250, t1Next = 245;

```
if(msTicks - t1Next ≥ 10) ...
```

$(250 - 245) \% 256 = 5 \ngeq 10$

e.g. add 10 to msTicks: $(250 + 10) \% 256 = 4$

```
if(msTicks - t1Next ≥ 10) ...
```

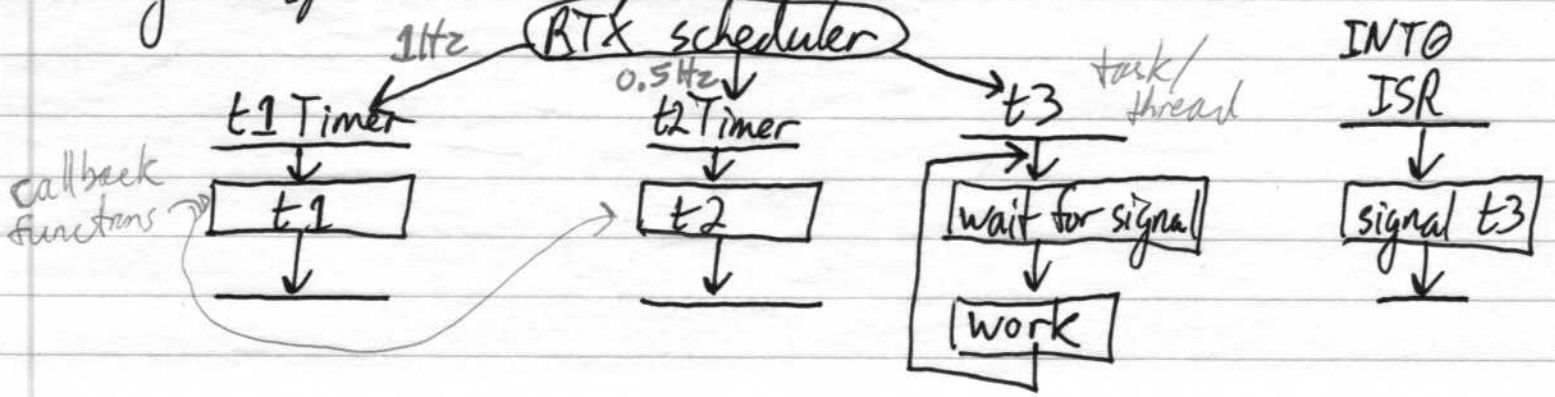
$(4 - 245) \% 256 = 15 \geq 10$

- problems with roll-your-own schedulers
  1. error prone (complicated code)
  2. no task priorities (and no preemption)

2) CMSIS-RTOS RTX

- Real-Time eXecutive
- it manages task scheduling, including task priorities

e.g. same example as above



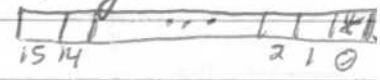
```
#include <stdbool.h>
#include <cmsis_os.h>          — RTOS header
#include <LPC17xx.h>           — board-specific info

void t1(void const *arg){...}
void t2(void const *arg){...}
osTimerDef(t1Timer, t1);       t1, t2 are call back functions
osTimerDef(t2Timer, t2);       run when the timers expire

void t3(void const *arg){
    while(true){
        osSignalWait(0x0001, osWaitForever);
        ...
    }
}
osThreadDef(t3, osPriorityNormal, 1, 0);
osThreadId t3Id;

void setupINT0(void){ same as before }
void EINT3_IRQHandler(void){
    osSignalSet(t3Id, 0x0001);
    LPC_GPIOINT->IO2IntClr |= (1<<10);
}
```

bit mask specifies 1 or more of a task's 16 event flags



don't time out

1 instance of t3

default stack size

signal t3's event flag 0

clear IRQ

```
int main(void) {
    osKernelInitialize();   set up kernel data structures, ready hardware including SysTick (defaults to 1ms)

    osTimerId t1Id = osTimerCreate(   timer repeats
        -->osTimer(t1Timer), osTimerPeriodic, NULL);   argument to callback function t1

    osTimerId t2Id = osTimerCreate(osTimer(t2Timer),
        -->osTimerPeriodic, NULL);
    t3Id = osThreadCreate(osThread(t3), NULL);   argument for thread function t3

    osKernelStart();   start scheduler
    osTimerStart(t1Id, 1000);   runs t1 at 1Hz
    osTimerStart(t2Id, 2000);   runs t2 at 0.5 Hz
    setupINT0();   allow push button interrupts
}
```

create OS objects

now main() is one of the threads scheduled

SysTicks

- thread priority enumeration:

~~osPriorityIdle = -3~~
~~" Low = -2~~
~~" BelowNormal = -1~~
~~" Normal = 0~~

```
enum osPriority {
    osPriorityIdle = -3,
    osPriorityLow = -2,
        "   BelowNormal = -1,
        "   Normal = 0,
        "   AboveNormal = +1,
        "   High = +2,
    osPriorityRealTime = +3,
    osPriorityError = 0x84
}
```

- the scheduler runs the ready threads that have highest priority