PGT302 – Embedded Software Technology

PART 3

Bare-metal Programming

Objectives for Part 3

- Need to DISCUSS and ANALYZE the following topics:
  - Bare-metal programming
  - Infinite loop (control loop)
  - Event triggered (interrupt controlled)
  - Time-slice @ round robin (cooperative multitask)
- Need to ANALYZE some examples (implementation) in assembly and C
- Need to DEVELOP some simple C codes

Embedded Systems Characteristics

- Require minimal user input
  - Simple user interface
  - Not programmable by end users
- Do specific task (known input patterns)
  - Respond to real-time @ predicted/expected events
  - Known set of inputs
- Provide only expected output
  - Error tolerant (must output something)

Embedded System Implementations

- Does not need the fastest processing unit
  - Real-time is knowing when to react!
- Most popular processing unit is microcontroller
  - Programmable, self-sufficient microprocessor
  - Multiple reprogramming with flash
- Programmable device like FPGA offers dynamic hardware reconfiguration (reconfigurable computing)
- Deciding factor: more features at lower cost

Embedded Systems Implementation

- Specifications-based design decisions
  - Performance or cost?
  - Power requirements
- Contemporary design decisions
  - Microprocessor/microcontroller selection
    - Legacy processing or reconfigurable computing
  - OS kernel or ‘bare-metal’ code
  - Assembly or high-level programming

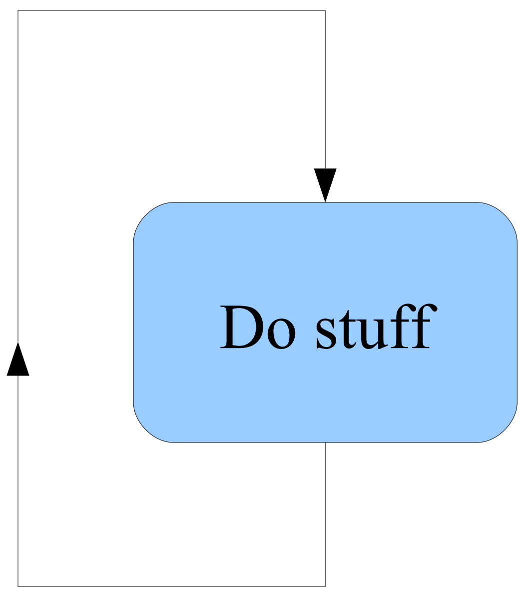
Bare-metal Programming

- Creating bare-metal codes
  - code that runs on hardware without any OS
  - low-level hardware access
- Simple/common application
  - single task, single threaded
  - using (100%) assembly is still possible
  - using C is sometimes an overkill (still.. it works)
- May implement multi-tasking
  - static scheduling

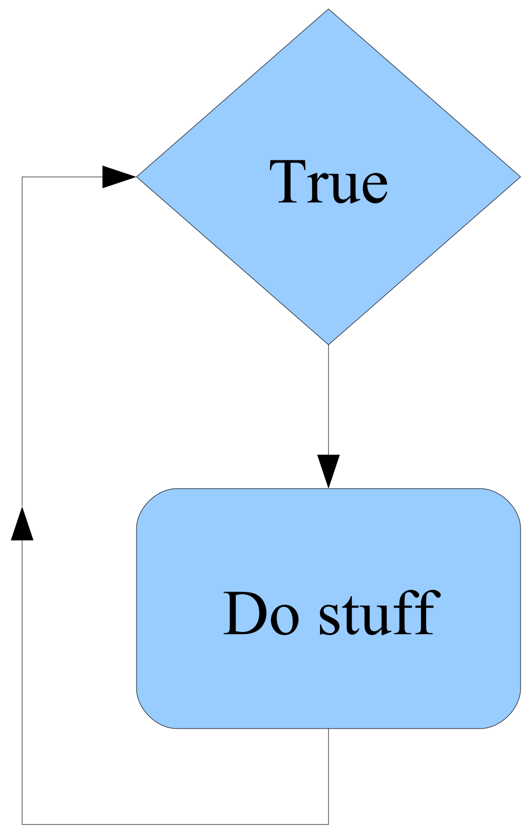
Bare-metal Programming (cont.)

- Basic code structure
  - Simple control (do only one task... for now)
    - No complex algorithm
    - Most probably single input, single output
  - Do it indefinitely (loop!)
- Examples for discussion
  - Lighthouse control
  - Traffic light control
  - Pedestrian light control

Infinite Loop



Non-structured flow
(not in C unless using *goto*)



Structured flow

Simple Implementation (8051)

Infinite Loop

Assembly

```
; gtuc51x001 i/o board
led0     bit 0B4h ; P3.4

         cseg at 0000h
         jmp init
; skip interrupt vector table!
         cseg at 0040h
init:
here:
         cpl led0
         mov r1,#255
loop:
         mov r0,#255
         djnz r0,$
         djnz r1,loop
         jmp here

         end
```

C

```
__sbit __at (0xB5) led0;  /* P3.5 */

void main()
{
  unsigned char r0, r1;
  while(1)
  {
    led0 = !led0;
    for(r1=0;r1<255;r1++) {
      for(r0=0;r0<255;r0++) {} }
  }
}
```

Infinite Loop (cont.)

- Analysis of given simple implementation to blink an LED on 8051:
  - Assembly is more elaborate compared to C
  - C compiler does not necessarily produce optimum code, in-depth knowledge of compiler helps
  - Assembly code is platform dependent!
- Assembly or C?
  - Know your platform!
  - Know your task!
  - Bottom line: as long as it does its job!

Simple Implementation (BCM2835-ARM)

Infinite Loop (Assembly)

```
.section .boot
boot:
        ldr r0,=0x20200000
@set gpio as output
        mov r1,#1
        lsl r1,#21
        str r1,[r0,#16]
loop:
@clr gpio (on led!)
        mov r1,#1
        lsl r1,#15
        str r1,[r0,#44]
@loop delay
        mov r2,#0x3F0000
wait1:
        sub r2,#1
        cmp r2,#0
        bne wait1

...
```

```
...

@set gpio (off led!)
        mov r1,#1
        lsl r1,#15
        str r1,[r0,#32]
@loop delay
        mov r2,#0x3F0000
wait2:
        sub r2,#1
        cmp r2,#0
        bne wait2
@infinite loop
        b loop
```

Simple Implementation (BCM2835-ARM)

Infinite Loop (C)

```
unsigned int *gpio, loop;

void main(void)
{
  gpio = (unsigned int*) 0x20200000;
  gpio[4] = 1 << 21;

  while(1)
  {
    gpio[11] = 1 << 15;
    for(loop=0;loop<0x3F0000;loop++);
    gpio[8] = 1 << 15;
    for(loop=0;loop<0x3F0000;loop++);
  }
}
```

Event Triggered

- Interrupts! → Know your hardware...
- Code (interrupt handler) only triggers when a target interrupt occurs
  - Must have stack infrastructure
- Advantages:
  - Very efficient execution flow (system only executes when needed)
  - Easier to implement multi-tasking without a scheduler (static scheduling)
  - Enables power saving feature (only if supported by hardware)

Simple Implementation (8051)

Event Triggered

Assembly

```
led0     bit 0B4h ; P3.4
         cseg at 0000h
         jmp init
         cseg at 000bh
         jmp blink
         cseg at 0040h
init:    mov TMOD,#11h
         mov TH0,#4bh
         mov TL0,#0fdh
         setb EA
         setb ET0
         setb TR0
here:    jmp here
blink:   cpl led0
         mov TH0,#4bh
         mov TL0,#0fdh
         setb TR0
         reti
         end
```

C

```
__sbit __at (0xB5) led0;  /* P3.5 */

void blink() __interrupt 1
{
  led0 = !led0;
  TH0 = 0x4B; TL0 = 0xFD; TR0 = 1;
}

void main()
{
  TMOD = 0x11;
  TH0 = 0x4B; TL0 = 0xFD;
  EA = 1; ET0 = 1; TR0 = 1;
  while(1)
  {
  }
}
```

Event Triggered (cont.)

- Analysis of given simple implementation to blink an LED on 8051:
  - Clearly, C is more compact!
  - In event-triggered implementations, main loop may be empty!
  - Hardware initialization required → platform!
  - Even in C, some knowledge on hardware is required!
- Things look easier in C?
  - Still, it is a matter of taste... for now!

Multitasking

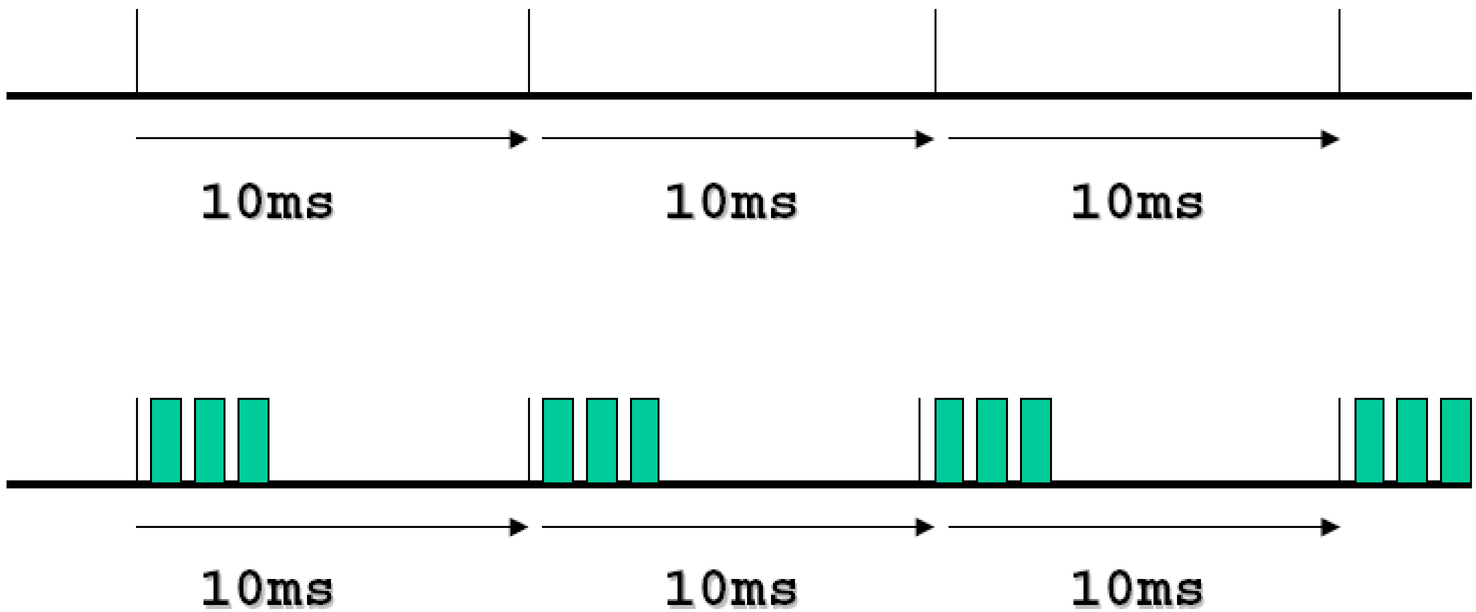
- Single process with proper timing (static scheduling)
- All tasks are known at design time
- All tasks must finish in one execution cycle
  - Either set limit for each task (limited task) OR,
  - Expand execution cycle (slower response)
- Can be seen as cooperative multitasking
- Advantages:
  - Does not need any special hardware requirements
  - Program flow is clearly defined and relatively simple to debug

Multitasking (cont.)

- Example:
  - Task 1 - Generate 12.5 Hz square wave
  - Task 2 - Generate 10 Hz square wave
  - Task 3 - Blink LED at rate ½ sec
- Time-triggered (event is based on timer)
  - Get an optimum system time slice!
  - Too small: not enough for tasks
  - Too large: system not responsive

Multitasking (cont.)

One CPU: Time slice



Bare metal on Raspberrry Pi

- Refer to codes in my1barepi
  - provides 'library' for easier coding experience
- GPIO access
  - include: gpio.h
  - initialize: gpio_init();
  - configure: gpio_config(<num>,<type>);
  - set to V_{DD}: gpio_set(<num>);
  - set to GND: gpio_clr(<num>);
  - read status: gpio_read(<num>);

Bare metal on Raspberrry Pi (cont.)

- Timer access (free-running counter)
  - include: timer.h
  - initialize: timer_init();
  - delay: timer_wait(<delay_us>);
  - read timer/counter value: timer_read();
- Timer access extension (control&interupt):
  - enable: timer_active(<enable>);
  - load value: timer_load(<32-bit_countdown);
  - enable irq: timer_setirq(<enable>);
  - clear irq pending bit: timer_irq_clear();
  - check irq pending bit: timer_irq_masked();

Bare metal on Raspberrry Pi (cont.)

- Interrupt access
  - include: interrupt.h
  - initialize: interrupt_init();
  - enable: interrupt_enable(<set>,<select>);
  - disable: interrupt_disable(<set>,<select>);
  - read pending bits: interrupt_pending(<set>,<mask>);
  - need special boot.s (assembly code configuration)
  - checkout example in my1barepi/t03_interrupt

Examples

- Lighthouse control
  - detect daylight (light-up when dark)
  - rotate at given rate (blink?)
- Traffic light control
  - generic 4-junction, timer based
- Pedestrian light control
  - single/dual request button
  - blinking green instead of yellow

The End

of Part 3