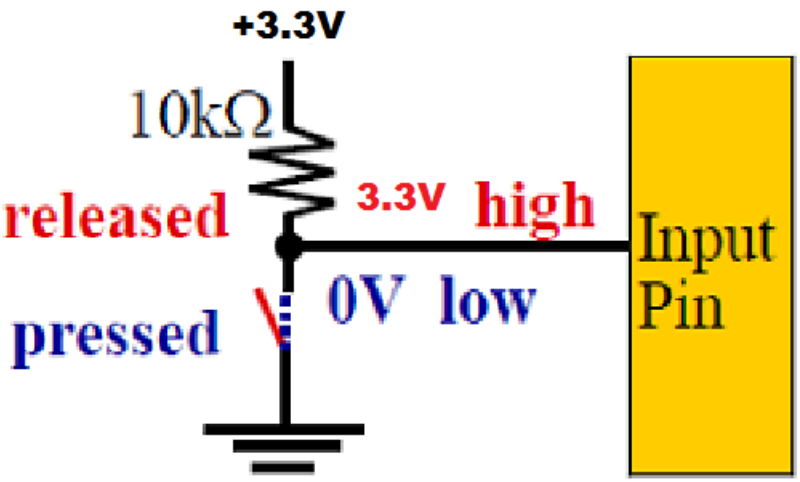
COMP462:
Embedded Systems

Lecture 4: Switch Interface, LED Interface, Branches, Stack Operations, Condition codes

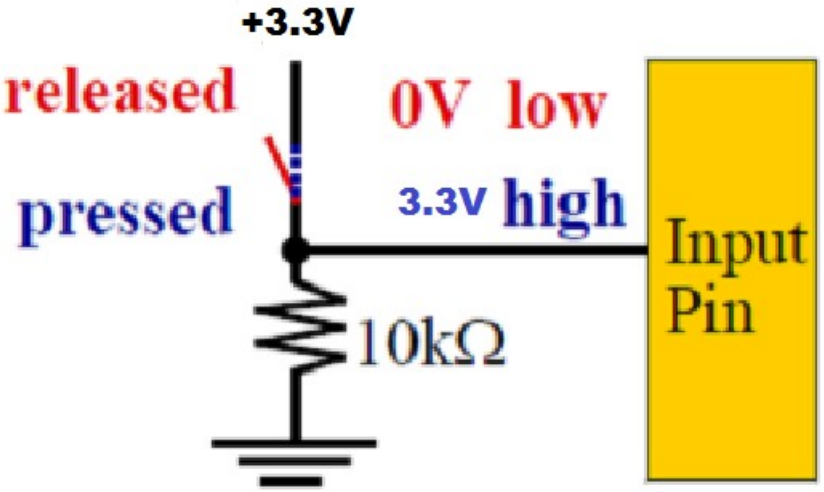
Agenda

- Recap, C programming
  - Variables (type, size, allocation)
  - Expressions
  - Conditionals
  - Loops
  - Functions (parameter passing)
- Outline
  - Switch interface
  - LED interface
  - Conditional Branches (condition codes)
  - Delay loops, duty cycle (PWM)
  - Abstraction & Refinement
    - Device Driver

Switch Configuration

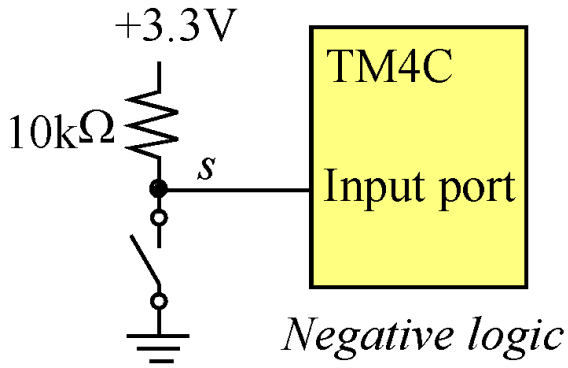


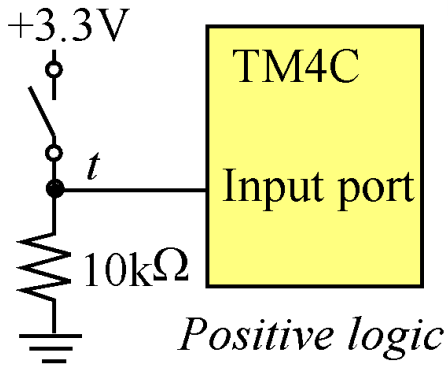
negative – pressed = ‘0’



positive – pressed = ‘1’

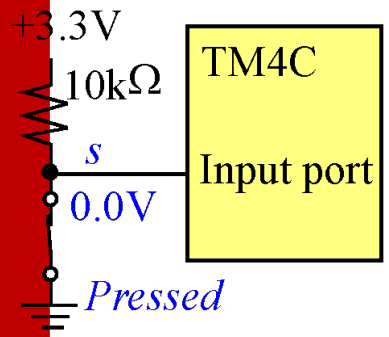
Switch Configuration

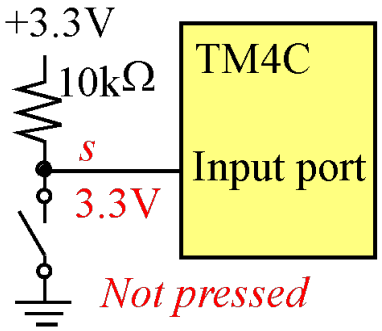


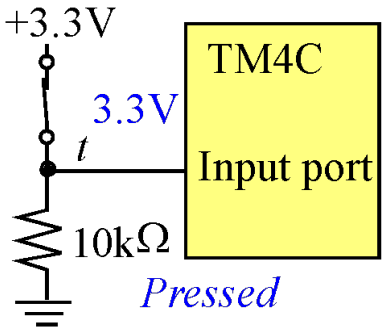


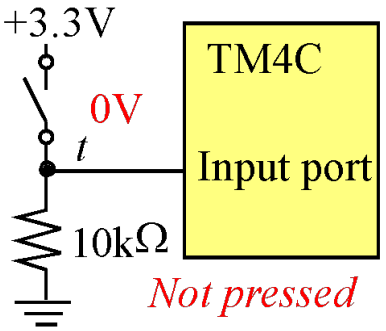
Negative Logic *s*
- pressed, 0V, false
- not pressed, 3.3V, true

Positive Logic *t*
- pressed, 3.3V, true
- not pressed, 0V, false

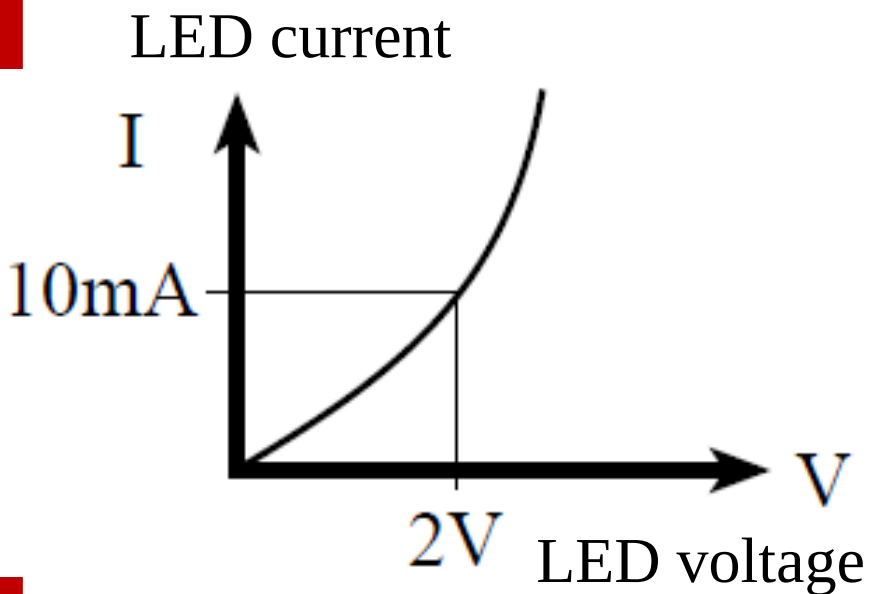


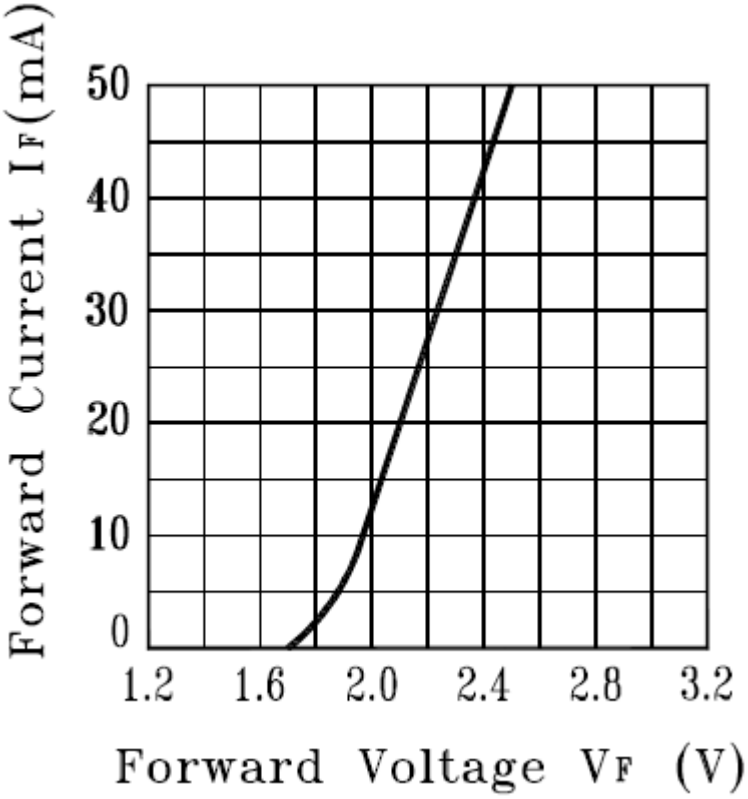






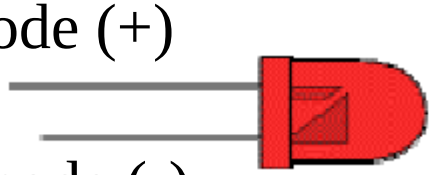
LED Interfacing





Brightness = power = V*I

Diode

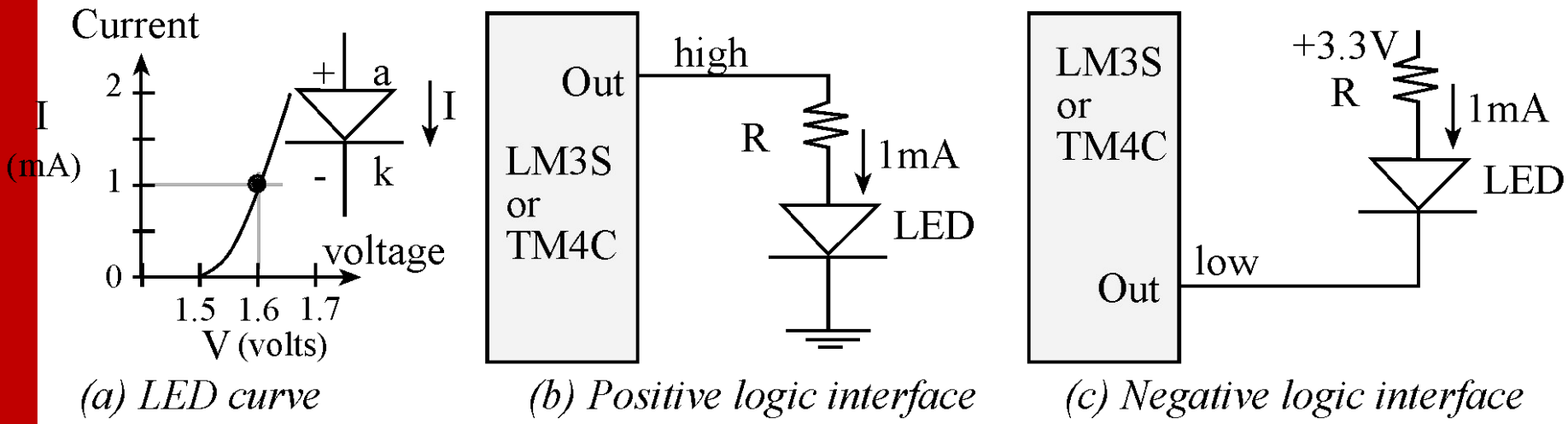


"Big voltage connects to big pin"

Low current LED Interface

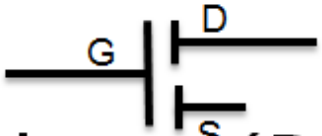
Current less than 8 mA

Diode

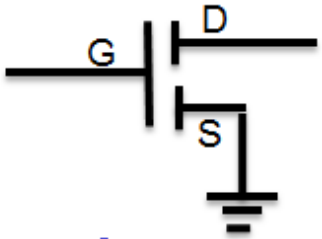


Transistors used as switches

- G high, on (D connected to S)
- G low, off (D disconnected from S)

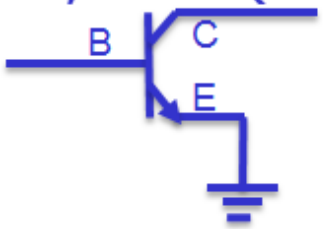


- G high, on (D is low)
- G low, off (D floats)



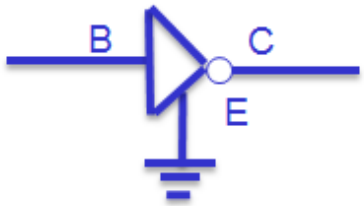
N channel MOS
G is gate (voltage)
D is drain
S is source

- B high, on (C is low)
- B low, off (C floats)



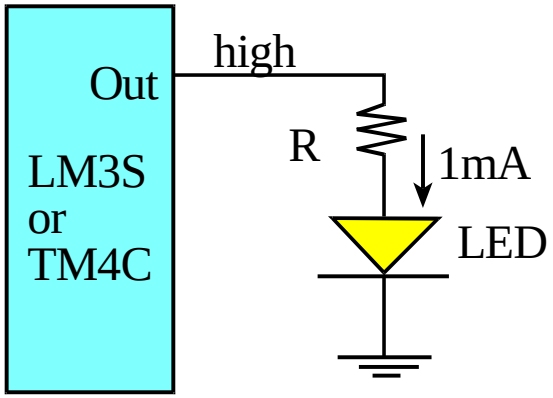
N channel Darlington
B is base (current)
C is collector
E is emitter

ULN2003B



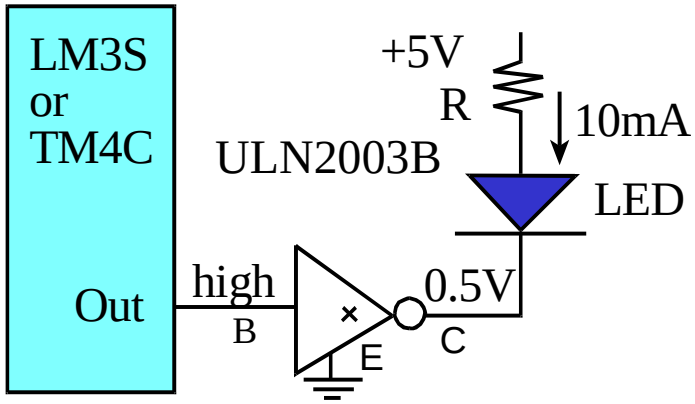
LED Interfacing

R = (3.3V – 1.6)/0.001
= 1.7 kOhm



LED current < 8 mA

R = (5.0-2-0.7)/0.01
= 230 Ohm



LED current > 8 mA

LED come in a variety of ways

- Voltage, current
- Brightness (efficiency)
- Size

Pulse Width Modulation

Oscillate LED

- **On for H time**
- **Off for L time**

Increase frequency until the eyes see it as continuous

- **Period is H+L**
- **Frequency is 1/(H+L)**

PWM

- **Fixed period P = H+L**
- **Duty cycle = H/(H+L) ranges from 0 to 1**
- **H ranges from 0 to P**
- **L = P-H**

Brightness is linear with duty cycle

- **Power = $(V_d * I_d) * H/(H+L)$**

For Loops for time delay


```
Delay SUBS R0,R0,#1
      BNE  Delay
      BX   LR
Main  BL   LED_Init
Loop  MOV  R0,#1
      BL   LED_Out
      LDR  R0,=250000
      BL   Delay
      MOV  R0,#0
      BL   LED_Out
      LDR  R0,=250000
      BL   Delay
      B    Loop
```

```
void Delay(uint32_t time){
  while(time){
    time--;
   }
}
int main(void){
  LED_Init();
  while(1){
    LED_Out(1);
    Delay(2500000);
    LED_Out(0);
    Delay(2500000);
  }
}
```

Use TExasDisplay to measure time delay
What if 2500000 were much smaller? E.g., 50000
What if delays were 10000 and 90000?
What if there were two loops (inner loops delaysH,100000-H) (outer loop varies H)

Device driver

Abstraction (separation of concerns)

Header file

- Prototypes for public functions
- What it does

```
void Switch_Init(void);
uint32_t Switch_In(void);
```

Code file

- Implementations
- How it works

```
void LED_Init(void);
void LED_Out(uint32_t data);
```

I/O Port Bit-Specific

- I/O Port bit-specific addressing is used to access port data register
  - Define address offset as $4*2^b$, where b is the selected bit position
  - Add offsets for each bit selected to base address for the port
  - Example: PF4 and PF0

If we wish to access bit	*Constant*
7	0x0200
6	0x0100
5	0x0080
4	0x0040
3	0x0020
2	0x0010
1	0x0008
0	0x0004

Port F = 0x4002.5000

0x4002.5000+0x0004+0x0040

= 0x4002.5044

Provides friendly and atomic access to port pins

Show **PortF_Input2** function in **InputOutput_xxxasm**

Condition Codes

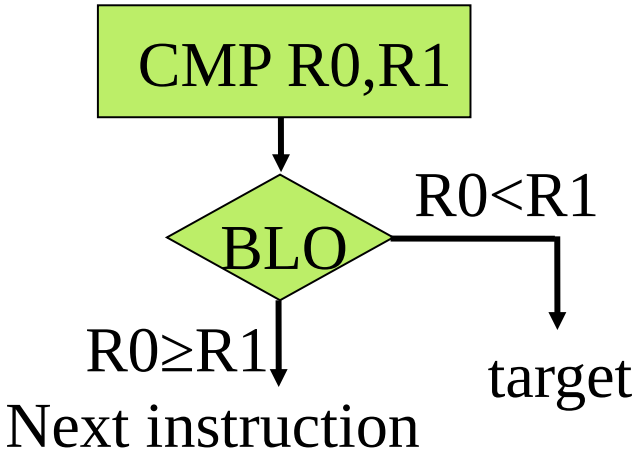
Bit	Name	Meaning after add or sub
N	negative	result is negative
Z	zero	result is zero
V	overflow	signed overflow
C	carry	unsigned overflow

- C set after an **unsigned** addition if the answer is wrong
- C cleared after an **unsigned** subtract if the answer is wrong
- V set after a **signed** addition or subtraction if the answer is wrong

Conditional Branch Instructions

- Unsigned conditional branch
  - follow **SUBS CMN** or **CMP**

```
BLO target    ; Branch if unsigned less than  (if C=0, same as BCC)
BLS target    ; Branch if unsigned less than or equal to (if C=0 or Z=1)
BHS target    ; Branch if unsigned greater than or equal to
                  (if C=1, same as BCS)
BHI target    ; Branch if unsigned greater than (if C=1 and Z=0)
```



Conditional Branch Instructions

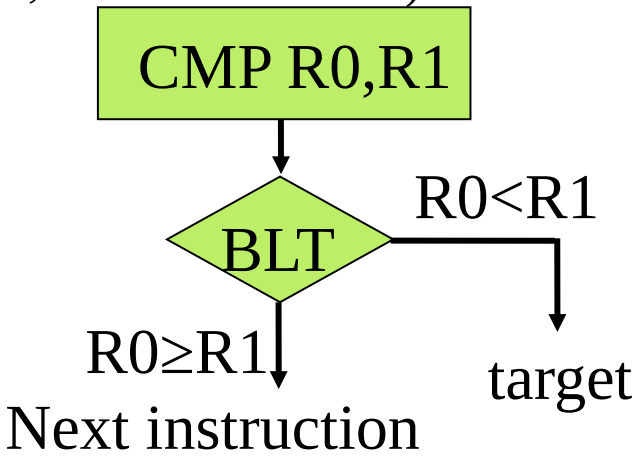
- Signed conditional branch
  - follow **SUBS CMN** or **CMP**

BLT target ; if signed less than (if (~N&V | N&~V)=1, i.e. if N≠V)

BGE target ; if signed greater than or equal to (if (~N&V | N&~V)=0, i.e. if N=V)

BGT target ; if signed greater than (if (Z | ~N&V | N&~V)=0, i.e. if Z=0 and N=V)

BLE target ; if signed less than or equal to (if (Z | ~N&V | N&~V)=1, i.e. if Z=1 or N≠V)



Equality Test

Assembly code	C code

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G == 7 ?`
`BNE next1 ; if not, skip`
`BL GEqual7 ; G == 7`
`next1`
	`uint32_t G;`
	`if(G == 7){`
	`GEqual7();`
	`}`

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G != 7 ?`
`BEQ next2 ; if not, skip`
`BL GNotEqual7 ; G != 7`
`next2`
	`if(G != 7){`
	`GNotEqual7();`
	`}`

Program 2.7.1. Conditional structures that test for equality.

Unsigned Conditional Structures

Assembly code	C code

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G > 7?`
`BLS next1 ; if not, skip`
`BL GGreater7 ; G > 7`
`next1`
	`uint32_t G;`
	`if(G > 7){`
	`  GGreater7();`
	`}`

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G >= 7?`
`BLO next2 ; if not, skip`
`BL GGreaterEq7 ; G >= 7`
`next2`
	`if(G >= 7){`
	`  GGreaterEq7();`
	`}`

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G < 7?`
`BHS next3 ; if not, skip`
`BL GLess7 ; G < 7`
`next3`
	`if(G < 7){`
	`  GLess7();`
	`}`

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G <= 7?`
`BHI next4 ; if not, skip`
`BL GLessEq7 ; G <= 7`
`next4`
	`if(G <= 7){`
	`  GLessEq7();`
	`}`

Program 2.7.2. Unsigned conditional structures.

Signed Conditional Structures

Assembly code	C code

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G > 7?`
`BLE next1 ; if not, skip`
`BL GGreater7 ; G > 7`
`next1`
	`int32_t G;`
	`if(G > 7){`
	`  GGreater7();`
	`}`

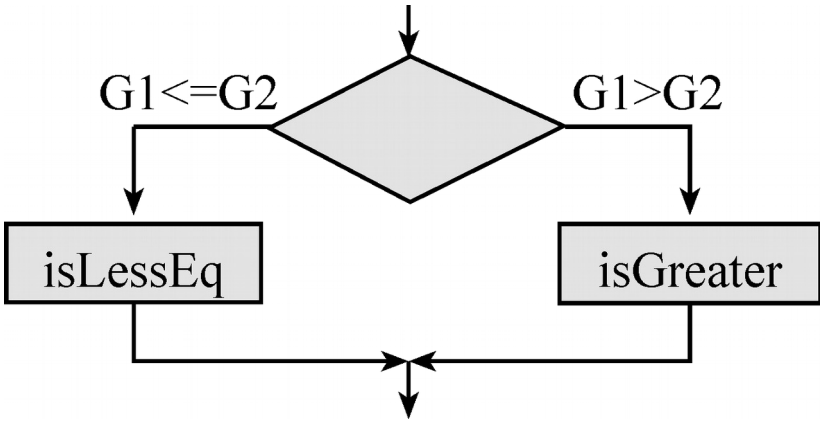
`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G >= 7?`
`BLT next2 ; if not, skip`
`BL GGreaterEq7 ; G >= 7`
`next2`
	`if(G >= 7){`
	`  GGreaterEq7();`
	`}`

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G < 7?`
`BGE next3 ; if not, skip`
`BL GLess7 ; G < 7`
`next3`
	`if(G < 7){`
	`  GLess7();`
	`}`

`LDR R2, =G ; R2 = &G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G <= 7?`
`BGT next4 ; if not, skip`
`BL GLessEq7 ; G <= 7`
`next4`
	`if(G <= 7){`
	`  GLessEq7();`
	`}`

Program 2.7.4. Signed conditional structures.

If-then-else

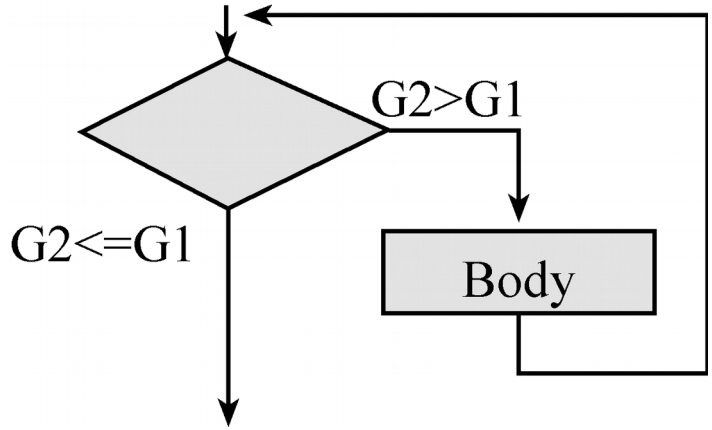



```
      LDR R2, =G1    ; R2 = &G1
      LDR R0, [R2]   ; R0 = G1
      LDR R2, =G2    ; R2 = &G2
      LDR R1, [R2]   ; R1 = G2
      CMP R0, R1     ; is G1 > G2 ?
      BHI high       ; if so, skip to high
low   BL  isLessEq   ; G1 <= G2
      B   next       ; unconditional
high  BL  isGreater  ; G1 > G2
next
```

```
uint32_t G1,G2;
if(G1>G2){
  isGreater();
}
else{
  isLessEq();
}
```

Program 2.7.6

While Loops



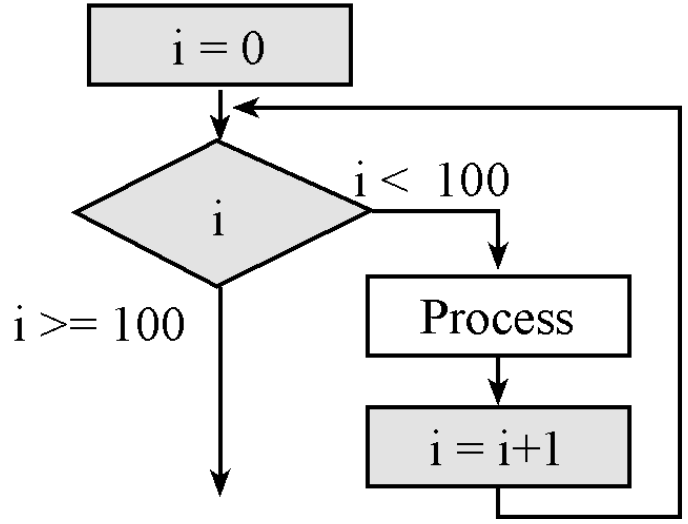

```
      LDR R4, =G1    ; R4 -> G1
      LDR R5, =G2    ; R5 -> G2
loop  LDR R0, [R5]   ; R0 =  G2
      LDR R1, [R4]   ; R1 =  G1
      CMP R0, R1     ; is G2 <= G1?
      BLS next       ; if so, skip to next
      BL  Body       ; body of the loop
      B   loop
next
```

```
uint32_t G1,G2;
while(G2 > G1){
  Body();
}
```

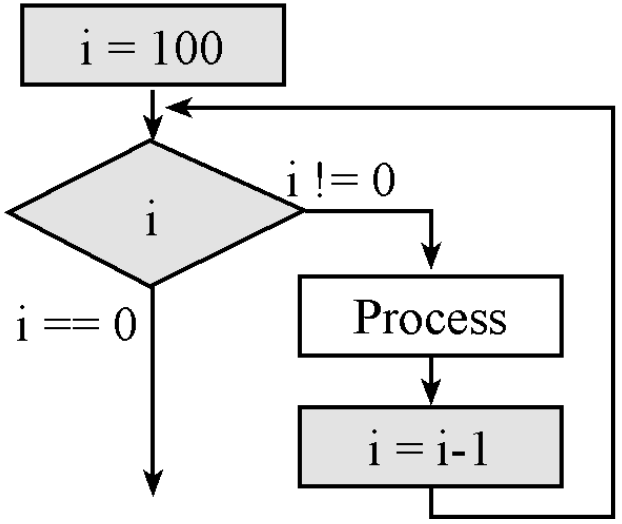
Program 2.7.8

For Loops

```
for(i=0; i<100; i++){
  Process();
}
```




```
for(i=100; i!=0; i--){
  Process();
}
```



For Loops


```
      MOV R4, #0      ; R4 = 0
loop  CMP R4, #100    ; index >= 100?
      BHS done        ; if so, skip to done
      BL  Process     ; process function*
      ADD R4, R4, #1  ; R4 = R4 + 1
      B   loop
done
```

```
for(i=0; i<100; i++){
  Process();
}
```

Count up

```
      MOV  R4, #100   ; R4 = 100
loop  BL   Process    ; process function
      SUBS R4, R4, #1 ; R4 = R4 - 1
      BNE  loop
done
```

```
for(i=100; i!=0; i--){
  Process();
}
```

Count down

Registers to pass parameters

High level program	Subroutine
1) Sets Registers to contain inputs	
2) Calls subroutine	
	3) Sees the inputs in registers
	4) Performs the action of the subroutine
	5) Places the outputs in registers
6) Registers contain outputs	

Stack to pass parameters

High level program	Subroutine
1) Pushes inputs on the Stack	
2) Calls subroutine	
	3) Sees the inputs on stack (pops)
	4) Performs the action of the subroutine
	5) Pushes outputs on the stack
6) Stack contain outputs (pop)	
7) Balance stack	

Assembly Parameter Passing

- Parameters passed using registers/stack
- Parameter passing need not be done using only registers or only stack
  - Some inputs could come in registers and some on stack and outputs could be returned in registers and some on the stack
- Calling Convention
  - Application Binary Interface (ABI)
  - ARM: use registers for first 4 parameters, use stack beyond, return using R0
- Keep in mind that you may want your assembly subroutine to someday be callable from C or callable from someone else's software

ARM Arch. Procedure Call Standard (AAPCS)

- ABI standard for all ARM architectures
- Use registers R0, R1, R2, and R3 to pass the first four input parameters (in order) into any function, C or assembly.
- We place the return parameter in Register R0.
- Functions can freely modify registers R0–R3 and R12. If a function needs to use R4 through R11, it is necessary to push their current register values onto the stack, use the register, and then pop the old value off the stack before returning.

Parameter-Passing: Registers

Caller

```
;--call a subroutine that
;uses registers for parameter passing
      MOV R0,#7
      MOV   R1,#3
      BL      Exp
  ;; R2 becomes 7^3 = 343 (0x157)
```

Call by value

- Suggest changes to make it AAPCS

Callee

```
;---------Exp-----------
; Input: R0 and R1 have inputs XX an YY
; Output: R2 has the result XX raised to YY
; Comments: R1 and R2 and non-negative
;          Destroys input R1
XX          RN  0
YY          RN  1
Pow         RN  2
Exp
        ADDS   XX,#0
        BEQ    Zero
        ADDS   YY,#0    ; check if YY is zero
        BEQ    One
        MOV    pow, #1  ; Initial product is 1
More    MUL    pow,XX    ; multiply product with XX
        SUBS   YY,#1     ; Decrement YY
        BNE    More
        B      Retn      ; Done, so return
Zero    MOV          pow,#0    ; XX is 0 so result is 0
        B       Retn
One     MOV    pow,#1     ;  YY is 0 so result is 1
Retn    BX     LR
```

Return by value

Parameter-Passing: Stack

Caller

```
;-------- call a subroutine that
; uses stack for parameter passing
     MOV  R0,#12
      MOV  R1,#5
      MOV  R2,#22
      MOV  R3,#7
      MOV  R4,#18
      PUSH {R0-R4}
    ; Stack has 12,5,22,7 and 18
    ; (with 12 on top)
      BL  Max5
; Call Max5 to find the maximum
;of the five numbers
      POP {R5}
;; R5 has the max element (22)
```

Callee

```
;---------Max5-----------
; Input: 5 signed numbers pushed on the stack
; Output: put only the maximum number on the stack
; Comments: The input numbers are removed from stack
numM  RN1   ; current number
max     RN  2   ; maximum so far
count   RN  0   ; how many elements
Max5
        POP    {max}     ; get top element (top of stack)
                         ; store it in max
        MOV   count,#4  ; 4 more to go
Again  POP   {numM} ; get next element
        CMP   numM,max
        BLT    Next
        MOV   max, numM  ; new numM is the max
Next   SUBS count,#1  ; one more checked
        BNE    Again
        PUSH  {max}      ; found max so push it on stack
        BX      LR
```

Call by value

Return by value

- Flexible style, but not AAPCS

Parameter-Passing: Stack & Regs

Caller

```
;------call a subroutine that uses
;both stack and registers for
;parameter passing
     MOV  R0,#6 ; R0 elem count
     MOV  R1,#-14
      MOV  R2,#5
      MOV  R3,#32
      MOV  R4,#-7
      MOV  R5,#0
      MOV  R6,#-5
      PUSH {R4-R6} ; rest on stack
; R0 has element count
; R1-R3 have first 3 elements;
; remaining on Stack
      BL  MinMax
;; R0 has -14 and R1 has 32
;; upon return
```

- Not AAPCS

Callee

```
;---------MinMax-----------
; Input: N numbers reg+stack; N passed in R0
; Output: Return in R0 the min and R1 the max
; Comments: The input numbers are removed from stack
numMM RN 3; hold the current number in numMM
max    RN  1       ; hold maximum so far in max
min    RN  2
N      RN  0       ; how many elements
MinMax
       PUSH {R1-R3}    ; put all elements on stack
        CMP  N,#0       ; if N is zero nothing to do
        BEQ  DoneMM
        POP  {min}      ; pop top and set it
        MOV  max,min    ;  as the current min and max
loop   SUBS N,#1        ;  decrement and check
        BEQ   DoneMM
        POP   {numMM}
        CMP   numMM,max
        BLT   Chkmin
        MOV   max,numMM  ; new num is the max
Chkmin CMP numMM,min
        BGT   NextMM
        MOV   min,numMM  ; new num is the min
NextMM B   loop
DoneMM MOV R0,min       ; R0 has min
          BX  LR
```

System Design

- What does being in a state mean?
  - List state parameters
- What is the starting state of the system?
  - Define the initial state
- What information do we need to collect?
  - List the input data
- What information do we need to generate?
  - List the output data
- How do we move from one state to another?
  - Actions we could do
- What is the desired ending state?
  - Define the ultimate goal

System Design

- Successive Refinement
- Stepwise Refinement
- Systematic Decomposition

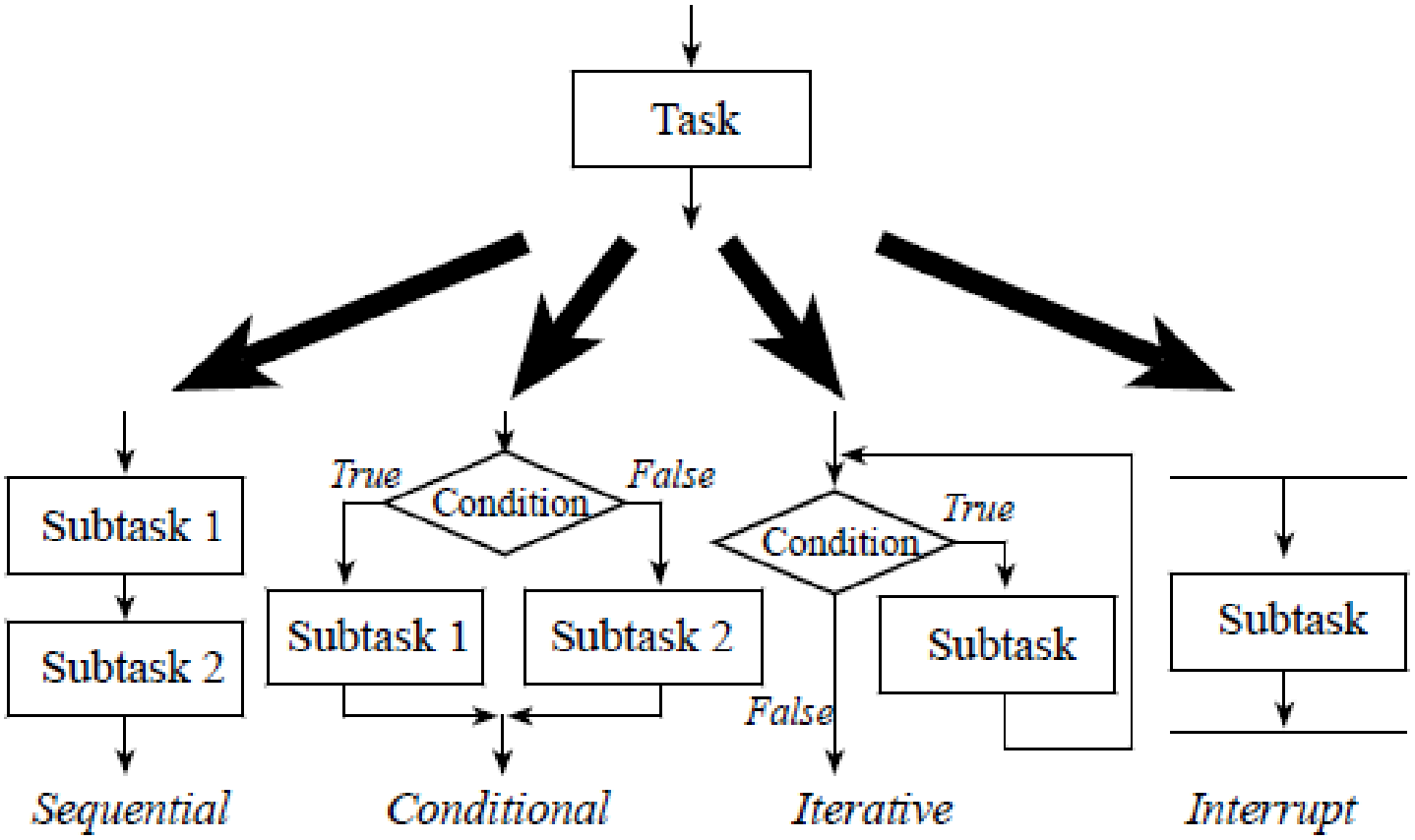
System Design

- Start with a task and decompose the task into a set of simpler subtasks
- Subtasks are decomposed into even simpler sub-subtasks
- Each subtask is simpler than the task itself
- Make design decisions
  - *document* decisions and subtask requirements
- Ultimately, subtask is so simple, it can be converted to software

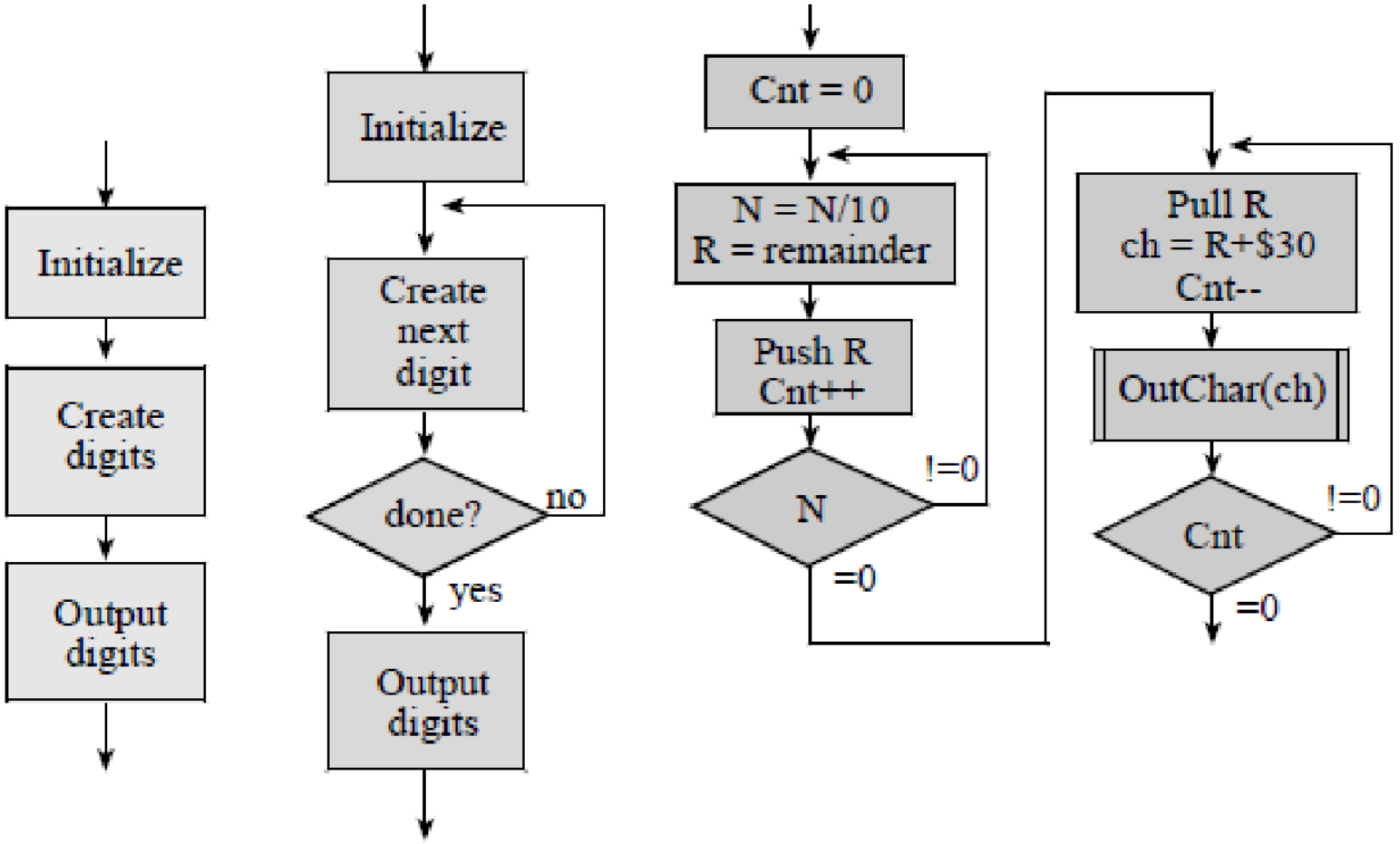
System Design

- Four building blocks (structured):
  - "do A then do B" → sequential
  - "do A and B in either order" → sequential (parallel)
  - "if A, then do B" → conditional
  - "for each A, do B" → iterative
  - "do A until B" → iterative
  - "repeat A over & over forever" → iterative (condition always true)
  - "on external event do B" → interrupt
  - "every t msec do B" → interrupt

Successive Refinement



Successive Refinement



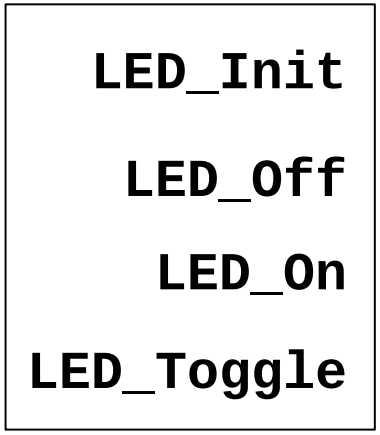
Successive refinement example for iterative approach

Abstraction - Device Driver

Abstraction allows us to modularize our code and give us the option to expose what we want users to see and hide what we don't want them to see.

A Device Driver is a good example where abstraction is used to expose *public* routines that we want users of the driver to call and use *private* routines to hide driver internals from the user (more on *private* routines later*)*

LED Driver (PE0)



*A user simply has to know what a routine expects and what it returns in order to call it (**calling convention**). Internals do not matter to caller*

Port E LED Abstraction

```
PE0 EQU 0x4005C004         ;bit-specific address Port E bit 0
LED_Init
    LDR R1, =SYSCTL_RCGCGPIO_R      ; R1 -> SYSCTL_RCGCGPIO_R
    LDR R0, [R1]                    ; previous value
    ORR R0, R0, #0x00000010         ; activate clock for Port E
    STR R0, [R1]
    NOP
    NOP                             ; allow time to finish activating
    LDR R1, =GPIO_PORTE_DIR_R       ; R1 -> GPIO_PORTE_DIR_R
    LDR R0, [R1]                    ; previous value
    ORR R0, R0, #0x01               ; PE0 output
    STR R0, [R1]                    ; set direction register
    LDR R1, =GPIO_PORTE_DEN_R       ; R1 -> GPIO_PORTE_DEN_R
    LDR R0, [R1]                    ; previous value
    ORR R0, R0, #0x01               ; enable PE0 digital port
    STR R0, [R1]                    ; set digital enable register
    BX  LR
```

Program 4.3. Software interface for an LED on PE0 (SSR_xxx.zip).

malloc and free


```
#include <stdlib.h>
int16_t *p; // shared pointer
void CreateIt(void){
  p = (char*)malloc(100*sizeof(int16_t));
}
void UseIt(int32_t n){
  p[n] = n;
}
void KillIt(void){
  free(p);
}
int main(void){ int16_t i;
  CreateIt();
  for(i=0;i<100;i++){
    UseIt(i);
  }
  KillIt();
  while(1){};
}
```

To use heap with Keil
1. Edit startup.s to create heap
2. Include standard library

Problems
- Heap becomes full
- Leakage
- Access before allocation
- Fragmentation
- Not deterministic

https://www.youtube.com/watch?v=5VnDaHBi8dM

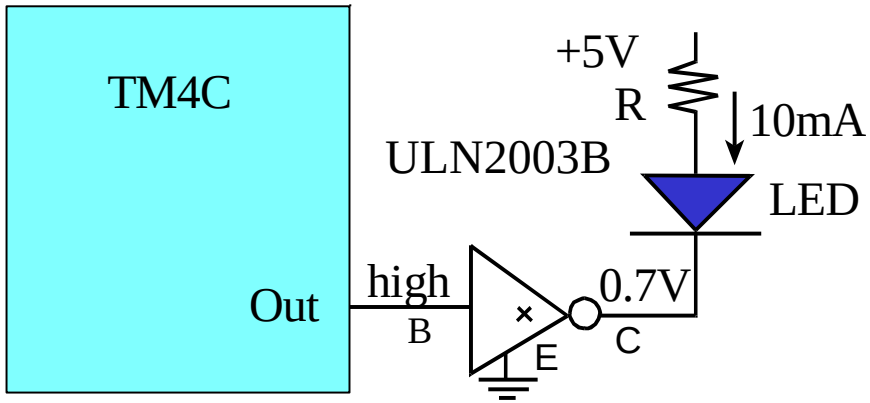
Scope and allocation

Scope \ Allocation	Temporary	Permanent
Private (restricted)	Local	Static
Public (unrestricted)	Heap	Global

LED Interfacing Review

LED current > 8 mA

R = (5.0-2-0.7)/0.01 = 230 Ohm
Use 220 ohm, it is close enough



LED is a diode

- P = V*I determines brightness
- Resistor used to set Vd,Id operating point
- Only conducts one way
- Id,Vd curve is exponential (very nonlinear)