Embedded Systems

- Students' expectations?
  what do you expect to learn, to know?
  how? why?

- What is an embedded system?
  - A computer system embedded in some other product: dish washer, car, radio, tv, alarm system, ice skate, ~~pre~~ remote control, ...
  - The primary objective is often <u>not</u> to process information (present)
  - Emphasis on sensing, actuation, control
  - Boundary is not rigid: phones (cordless, cellular, smartphones), games.
  - Related buzzwords: sensor networks, cyber-physical systems (CPS), ...

- What is special about developing embedded systems?
  - Hardware has a significant per-unit cost (software has almost only fixed dev costs, + support costs).
  - Optimization to ~~redu~~ eliminate parts, substitute cheaper parts, reduce board size/complexity, etc
  - Hardware & software co-developed: bugs can be in hardware or software
  - Hardware fails!
  - Communication between software & hardware developers is essential
  - Testing/debugging can require special test fixtures, equipment
  - Power may be a limiting factor (batteries, solar, etc)
  - Power & cost limitation ⇒ computational resource limitations
  - Real time constraints: easy to get μs accuracy, very hard in PC's, servers, etc
  - Unattended operation & limited or no UI require special attention to failure handling
  - Also, extensive know-how that is not deep but useful.

- Learning objectives of this course:
  - Know how & practical experience to set-up a dev environment, understand the documentaiton and code & debug
  - Understanding of the deep issues: concurrency, failures, timing, power, ...
  - Ability to communicate with hardware designers
  - [not a formal goal but not hard to attain after this ~~cou~~ course] ability to build ~~sim~~ simple embedded systems [easy, but high speed, high complexity, or reliable enough for a product is hard]

Main Platforms in this course

① LPC2148 Education Board

- Microcontroller (processor, memory, peripherals): LPC2148
- ARM7 (ARM7TDMI) core, NXP (Philips), 32-bit
  60 MHz, 512k flash 32+8k RAM, USB
- On board devices: character LCD, serial → USB, SD card
  joystick switch, 2 pots, LED's, LED display, RGB LED,
  stepper motor, digital temp sensor + EEPROM, buzzer
- Add-on boards (few): Ethernet, prototyping
- Not a real embedded system but superb for learning
  & prototyping
- programming through serial ↔ USB Bridge

② TelosB (T-mote sky; an old commercial name)

- MSP430 F 1611 low power MCU, 16-bit, 48+10k
- serial-to-USB bridge (including for programming)
- 802.15.4 transceiver chip (low power networking)
- Sensors: light ×2, temp & humidity, expansion connector
- Designed in UC Berkeley for WSN
- Runs Tiny OS & other WSN OS's, but OS's
  not used in this course (ok in projects)

Other equipment we have access to

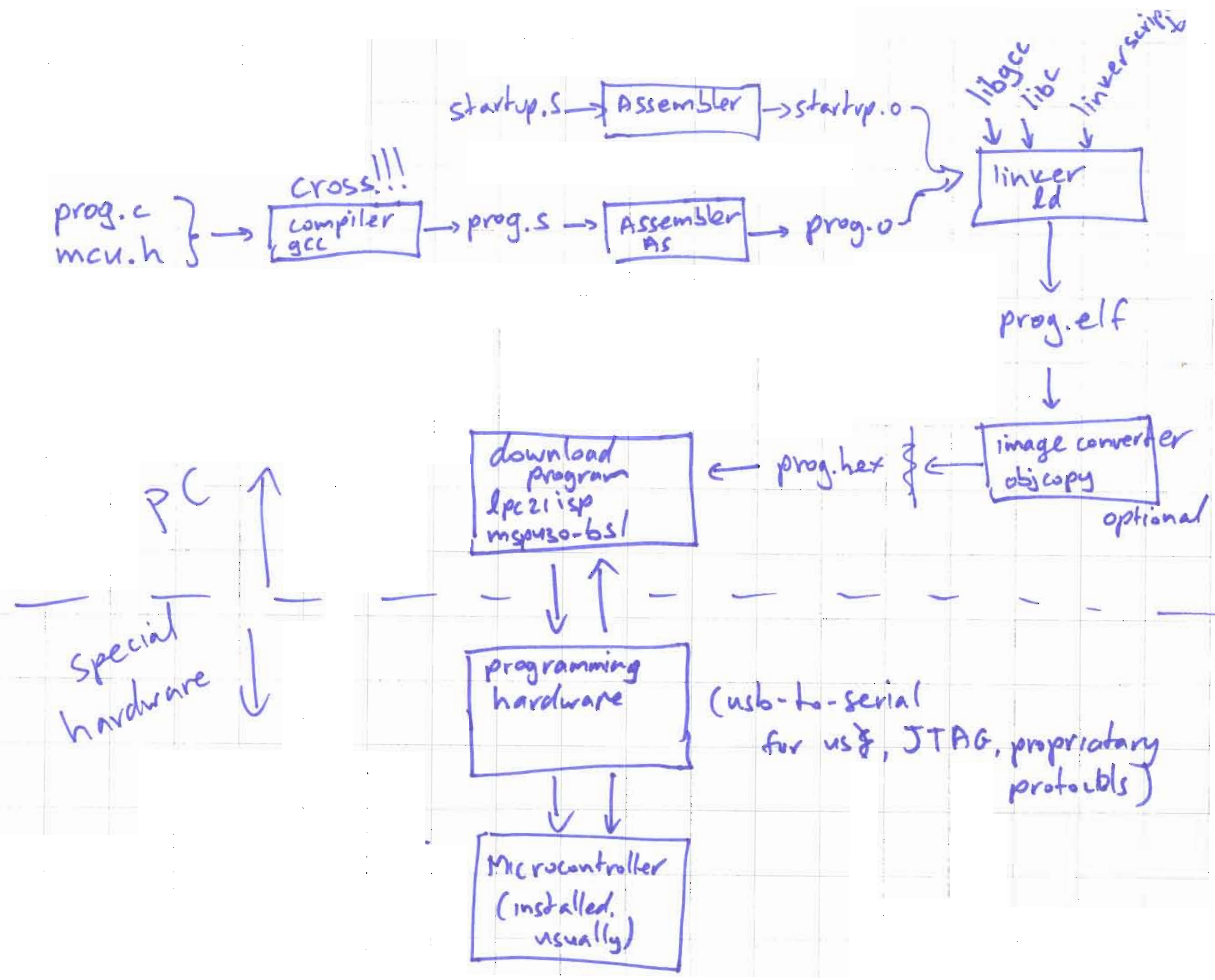
- More platforms: AVR Raven/Butterfly, LPC2148H, LPC2119, LPC2292, ez430 (+RF) GPS-OBD, various prototypes

- Simple test equipment: multimeter, osciloscope power supplies

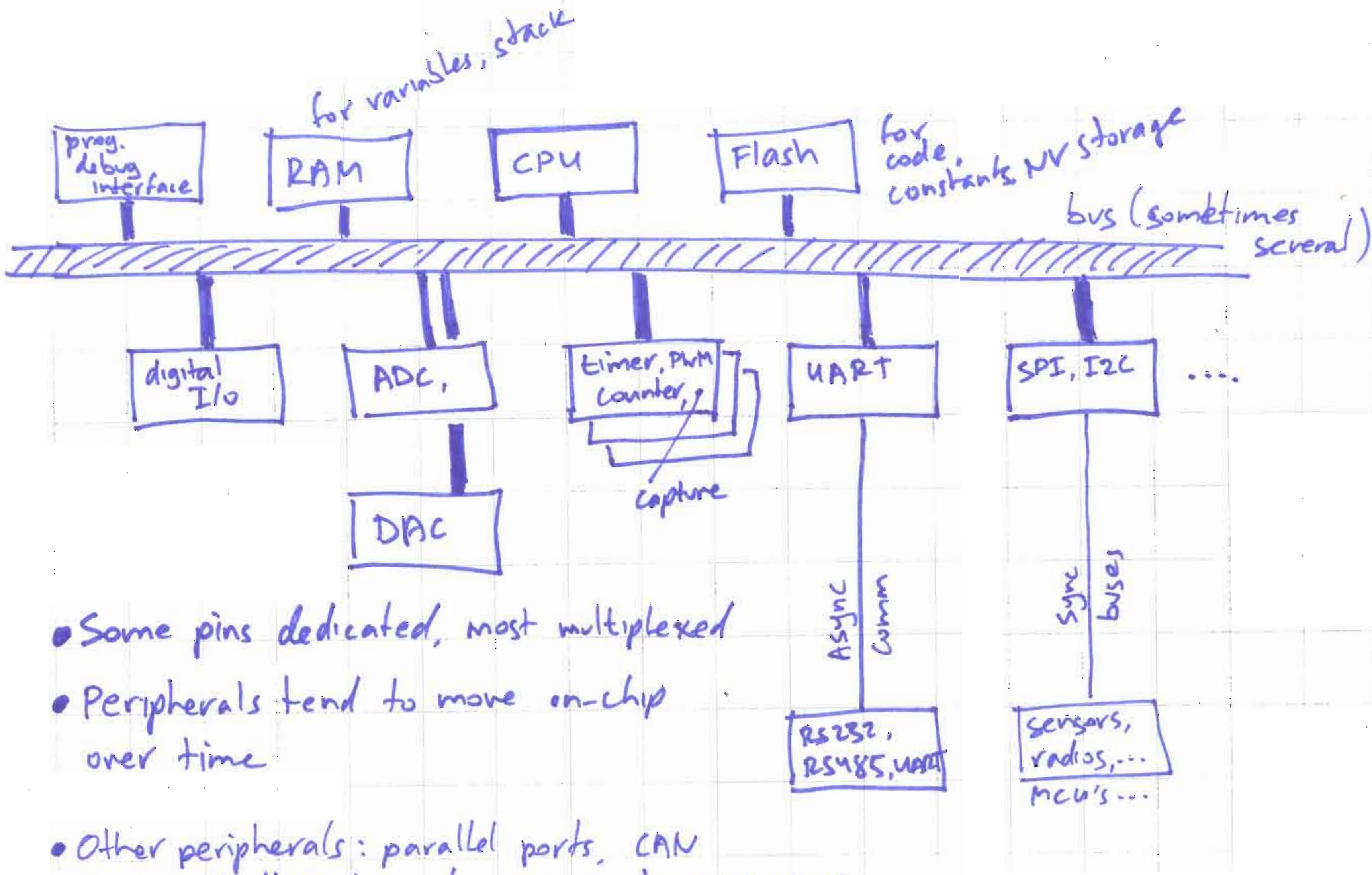
- Components for building systems:
  MCU's, other chips, electronic components, LCD's
  (more in the Engineering storeroom)
  Breadboards, protoboards, soldering equipment

- Programmers / debuggers: ez430, AVR Dragon, PIC kit 2 (low cost!)

Mechanics: Compiling, linking, downloading, debugging



Microcontrollers



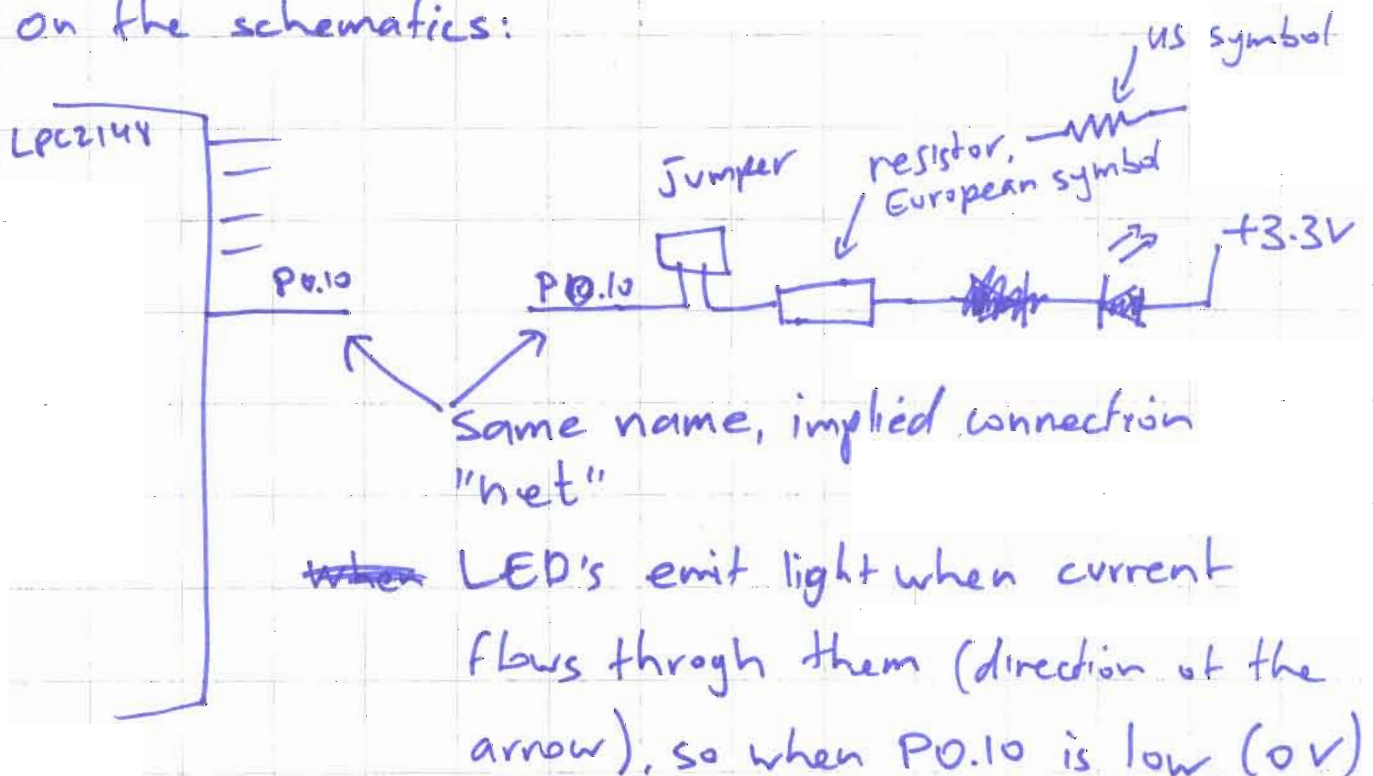
- Some pins dedicated, most multiplexed
- Peripherals tend to move on-chip over time

- Other peripherals: parallel ports, CAN LCD controllers, touch sensors, temp sensors encoder interfaces, radios (new), USB, Ethernet
- Large MCU's have external memory buses (to have enough mem for linux, etc)
- We will use bootloaders, not the hardware prog/debug interface.

A simple program, SFR's

~~← make pin P0.10 output (default is input, typical but not universal)~~
~~- delay loop~~

Goal: to blink an LED on the '2148 board



When P0.10 is high (~3.3V), no light

How do we tell the MCU to toggle P0.10?

We go to the user's manual (~~som~~ Ref manual, etc)

Then to GPIO (General ~~perr~~purpose IO, = digital I/O)

We learn:

SFR's (special function registers):

- IOØDIR controls direction (input / output) — 0 = input (default), 1 = output
- Address 0x E002 8008
- IOØPIN pin value (input or output), writing controls ~~changes~~ output pins

Bits have names eg P0 10DIR.

A header file provides symbolic names for thes registers (lpc2000/io.h → lpc2000/lpc214x.h)

```
# define IOPINØ (*(volatile uns. long *)(0xE0028010))
```

(IOPINØ: Error! but in the file)

sometimes the header also names bits, but not the headers of the LPC2000 family, so we define

```
#define BIT10 0x00000400
```

The program:

```
#include <lpc2000/io.h>

volatile int dummy;

const int delay = 100000;

int main(void) {
  int i;

  IODIR0 = BIT10;

  while (1) {
    for (i=0; i<delay; i++) dummy=1;
    IOPIN0 |=  BIT10;
    for (i=0; i<delay; i++) dummy=1;
    IOPIN0 &= ~BIT10;
  }
}
```

← delay loop

on or off?

why do we need the volatile dummy?

what happens before main?

almost nothing. The cpu starts at address ∅ on ARM's.

In Startup.S address ∅ is a branch to code that

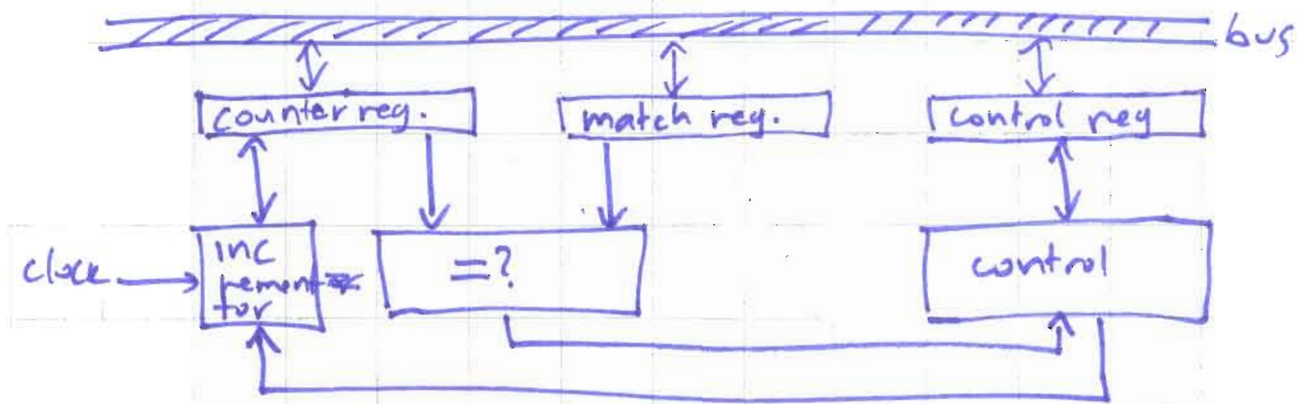
several on ARM...

* set up the stack pointer reg to end of RAM
* zeros uninitialized global vars
* copies initial value of global vars from flash to RAM
* calls main

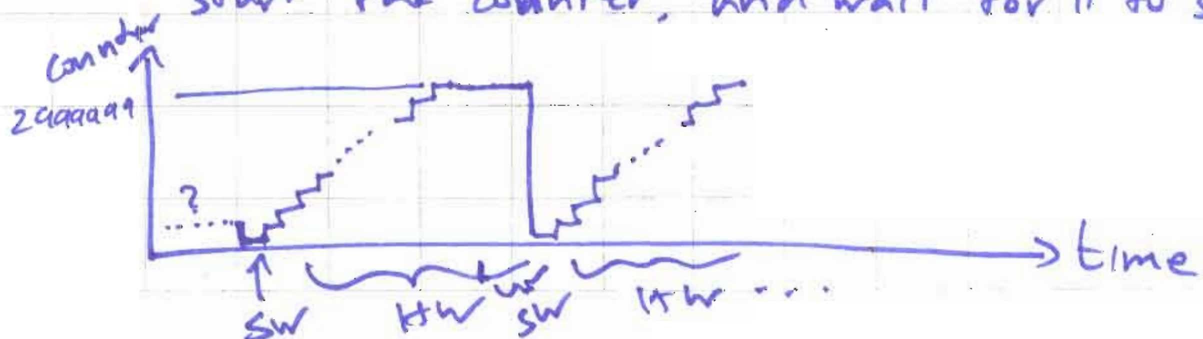
Timers / Counters

Goal: replace the delay loop by a more accurate & controlable mechanism

We will use a timer/counter peripheral



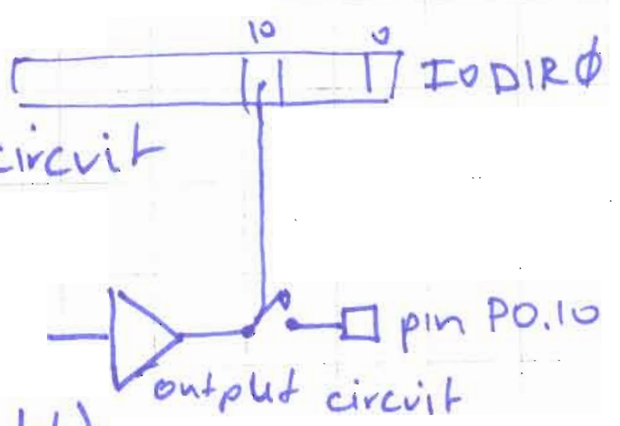
* Go to MCU's manual
  - ~~on our b~~ clock = "PCLK" ; on our board default freq is 3MHz (changeable)
  - Many SFRs, modes!
  - we will zero the counter, put 3000000-1 in MR, configure control to stop on match, start the counter, and wait for it to stop



```c
#include <lpc2000/io.h>

volatile int dummy;

const int delay = 100000;

int main(void) {
  int i;

  IODIR0 = BIT10;

  while (1) {
        T0TC  = 0;             /* zero the counter              */
    T0MR0 = 2999999;       /* match register                */
        T0MCR = BIT2;          /* stop on match                 */
        T0TCR = BIT0;          /* start timer0!                 */
        while (T0TCR & BIT0);

    IOPIN0 |=  BIT10;

    T0TC  = 0;             /* zero the counter              */
    T0MR0 = 2999999;       /* match register                */
        T0MCR = BIT2;          /* stop on match                 */
        T0TCR = BIT0;          /* start timer0!                 */
        while (T0TCR & BIT0);

        IOPIN0 &= ~BIT10;
  }
}
```

3 SFR behaviors

① ~~controls~~ configures the circuit



② Dual ported memory (latch)
e.g. T0TC

③ Triggers an action: e.g. bit 0 in T0TCR starts the timer; bit 1 resets it (we didn't use it)

From Polling to Interrupts

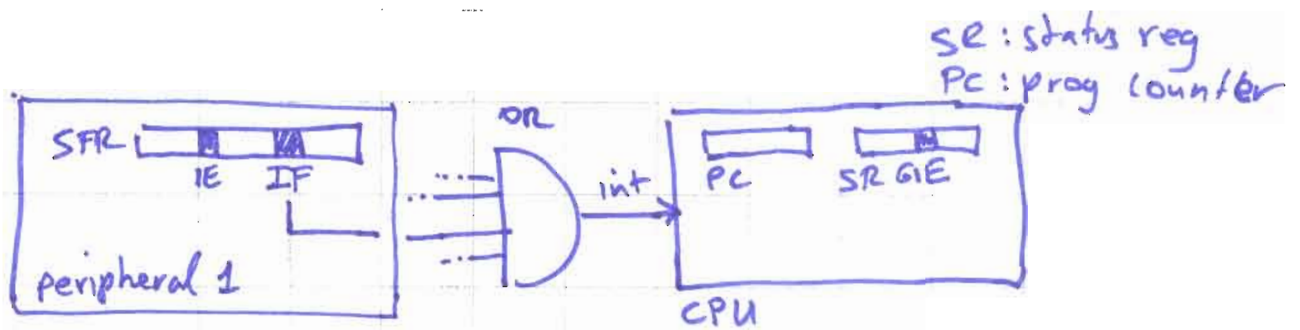
- The previous programs repeatedly check for some condition, then take action when the condition occurs.

- This is wastefull, and if the code must check for many conditions, may delay the reaction to an event

- It would be better if the program could get notified when an event/condition happens.

⇒ Set up timer so that when count reaches 2999999, it invokes a function & resets the counter.

Simple Interrupt Processing (eg PIC16)

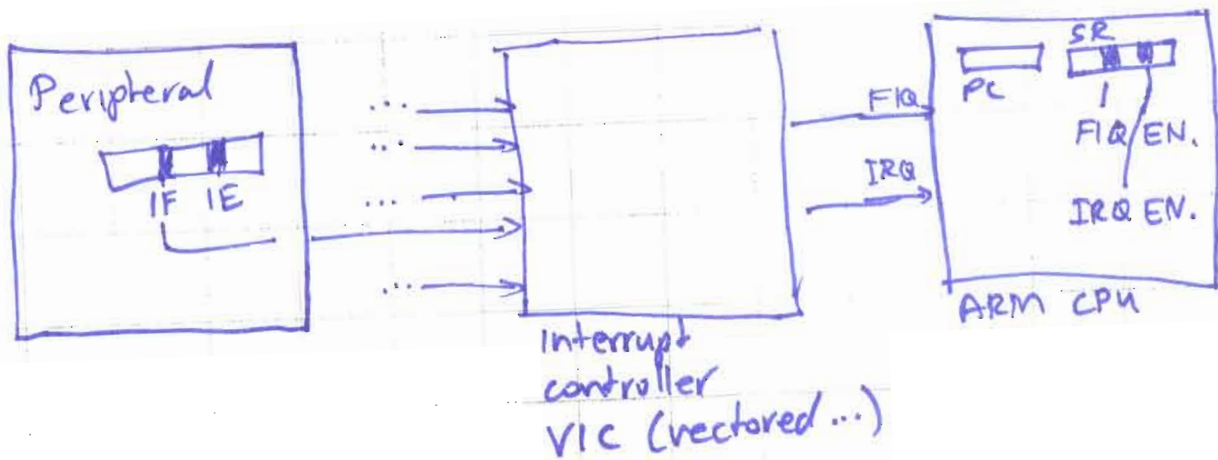


Peripheral: at event, if IE (interrupt enable), set IF (interrupt flag)

IF flag is tied to a signal that exits the peripheral.

CPU: before every instruction, if interrupt signal is active (say high) and GIE (Global IE), clear GIE, save PC (on stack or some special reg) and load into PC a fixed constant
the constant usually points to a branch to an Interrupt Service Routine (ISR).
Because GIE is cleared on entry to ISR, the ISR is not invoked recursively.

Interrupt Processing on LPC2000 (simplified)

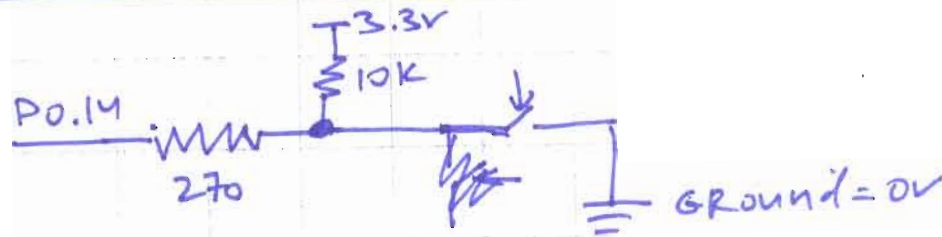


VIC determines routing (→FIQ or →IRQ) and priority of IRQ ISR's.
FIQ ISR can interrupt an IRQ isR, and is invoked faster

Compiler generated special code!

```c
#include <lpc2000/io.h>
#include <lpc2000/interrupt.h>

void __attribute__ ((interrupt("FIQ"))) fiq_isr(void) {
  T0IR = BIT0;            /* clear the interrupt flag    */

                         /* toggle the LED               */
  if (IOPIN0 & BIT10) IOPIN0 &= ~BIT10;
  else                IOPIN0 |=  BIT10;
}

int main(void) {
  IODIR0 = BIT10;

  T0MR0 = 2999999;        /* match register               */
  T0MCR = BIT0 | BIT1;    /* reset & interrupt on MR0     */
  T0TCR = BIT0;           /* start timer0!                */

  VICIntSelect = BIT4;    /* TIMER0 is the FIQ            */
  VICIntEnable = BIT4;    /* enable TIMER0 interrupts     */

  enable_interrupts();

  while (1) {
  }
}
```

External Interrupt from a Push Button



We can poll P0.14 using IOPIN0, or we can generate an interrupt signal when P0.14 goes down (or up, or when it is high or low)

warning: key bounce!

```
#include <lpc2000/io.h>
#include <lpc2000/interrupt.h>

void __attribute__ ((interrupt("FIQ"))) fiq_isr(void) {
  EXTINT   = BIT1;          /* clear the interrupt flag    */

                            /* toggle the LED              */
  if (IOPIN0 & BIT10) IOPIN0 &= ~BIT10;
  else                IOPIN0 |=  BIT10;
}

int main(void) {
  IODIR0 = BIT10;

  PINSEL0 = BIT29;          /* select EINT1 for P0.14      */
  EXTMODE = BIT1;           /* INT1 is edge sensitive      */
                            /* falling edge by default     */
  EXTINT   = BIT1;          /* clear the interrupt flag    */

  VICIntSelect = BIT15; /* EINT1 is the fast interrupt */
  VICIntEnable = BIT15; /* enable EINT1                */

  enable_interrupts();

  while (1) {
  }
}
```

connect EXTINT peripheral to pin, disconnect GPIO; more later (annotation pointing to `PINSEL0 = BIT29;`)