Embedded System Design: From Electronics to Microkernel Development

Rodrigo Maximiano Antunes de Almeida
E-mail: rmaalmeida@gmail.com
Twitter: @rmaalmeida
Universidade Federal de Itajubá

Creative Commons License

DEFCON

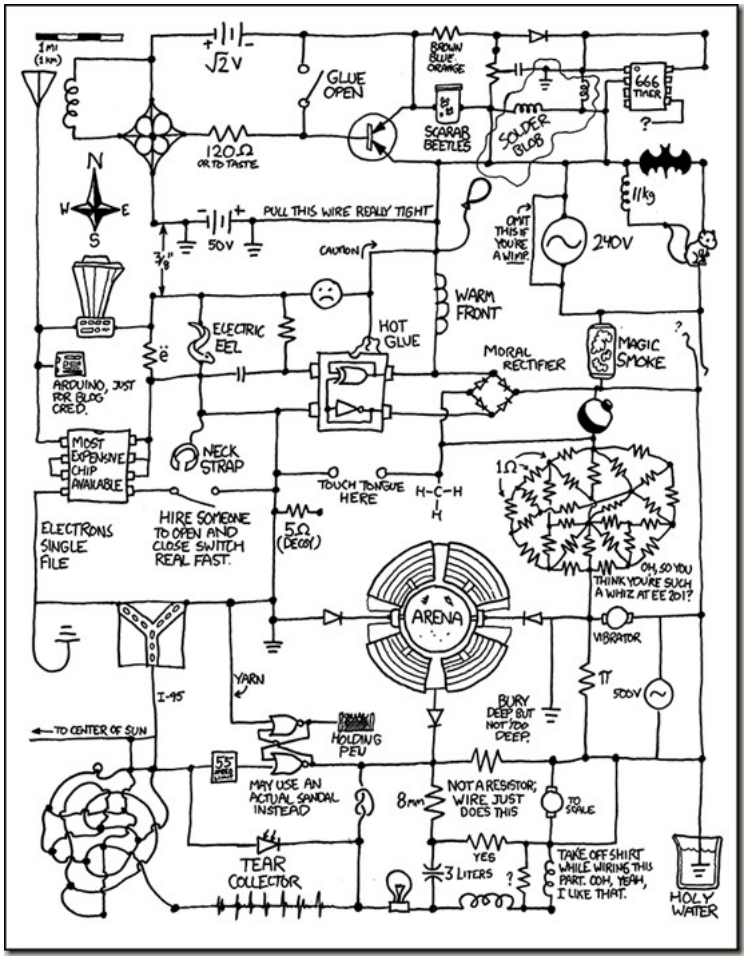
Workshop schedule

- Electronic building
- Board programming
- Kernel development

Electronic building

- Electronics review
  - Schematics
  - Protoboard/breadboard
- System design
  - Basic steps
  - Microcontroller
  - LCD
  - Potentiometer

Electronics review

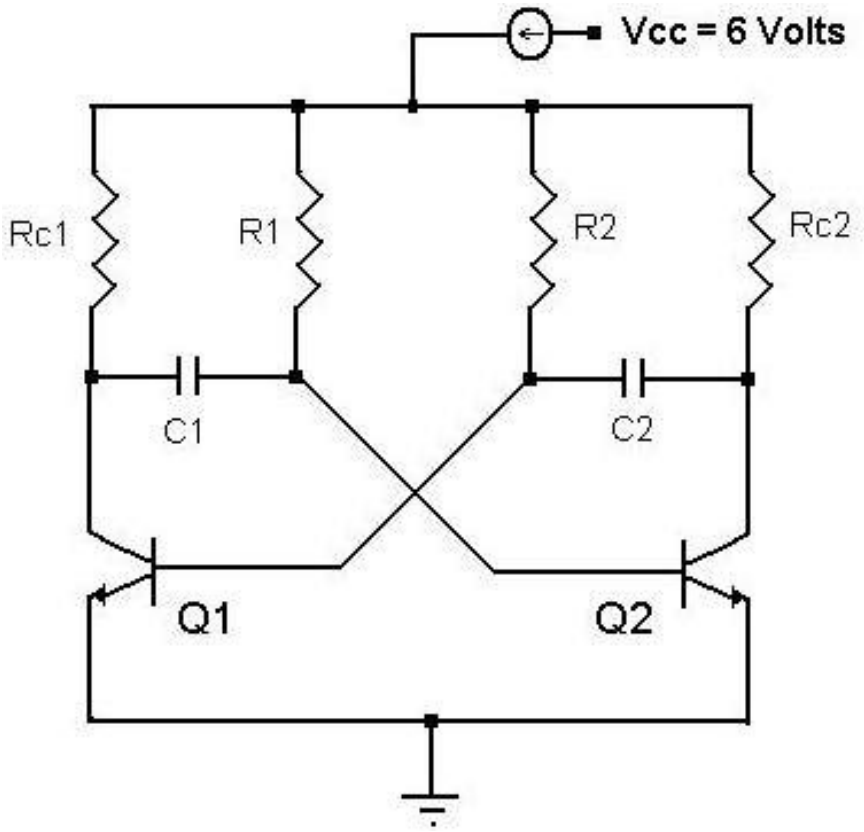


- http://xkcd.com/730/

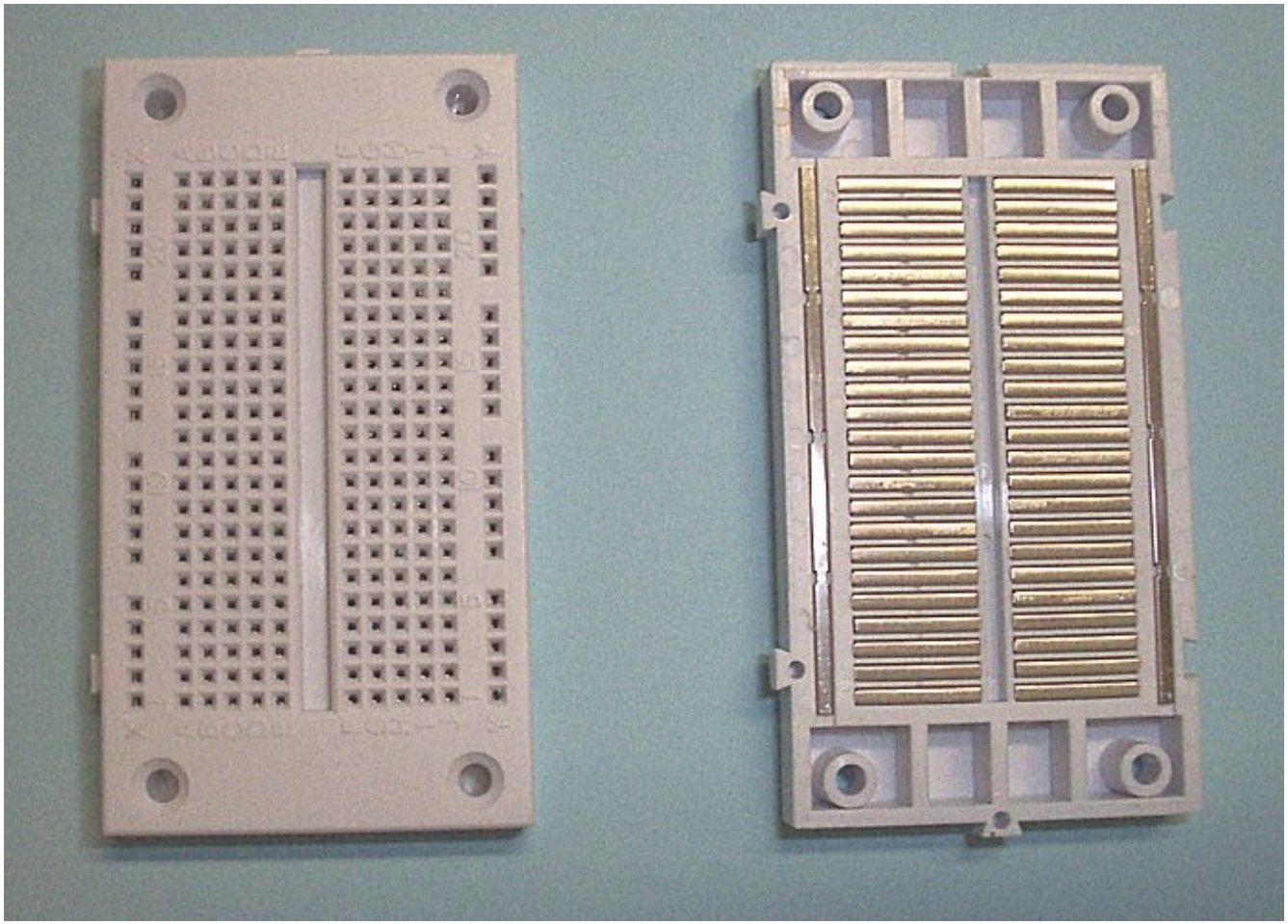
DEFCON

Schematics

- Way to represent the components and its connections
- Each component has its own symbol
- Crossing wires only are connected if joined with a dot



Protoboard/Breadboard



DEFCON

System design

- Steps on a generic electronic system design
  - Define the **objective**(s)
  - **Choose** the main components needed to achieve the objective
  - Get the use example and recommendations from component **datasheet**
  - Build the **schematics**
  - **Simulation** of HW elements
  - Board **layout**

Datasheets

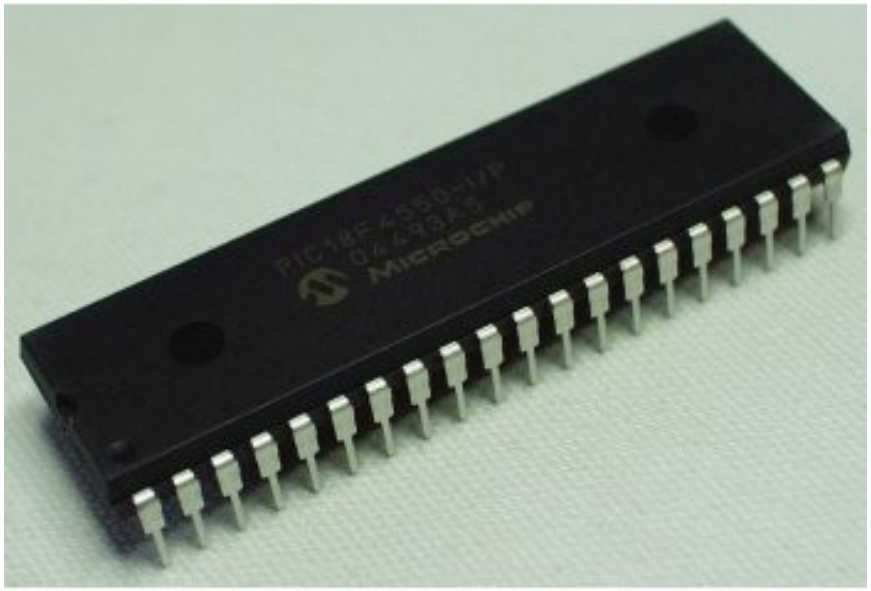
- The main source of information concerning electronics
- Presents
  - Electrical characteristics
  - Simplified schematics
  - Use example
  - Opcodes/API

System design

- Objective
  - Build digital voltage reader
- Main components
  - Microcontroller
  - LCD text display
  - Potentiometer

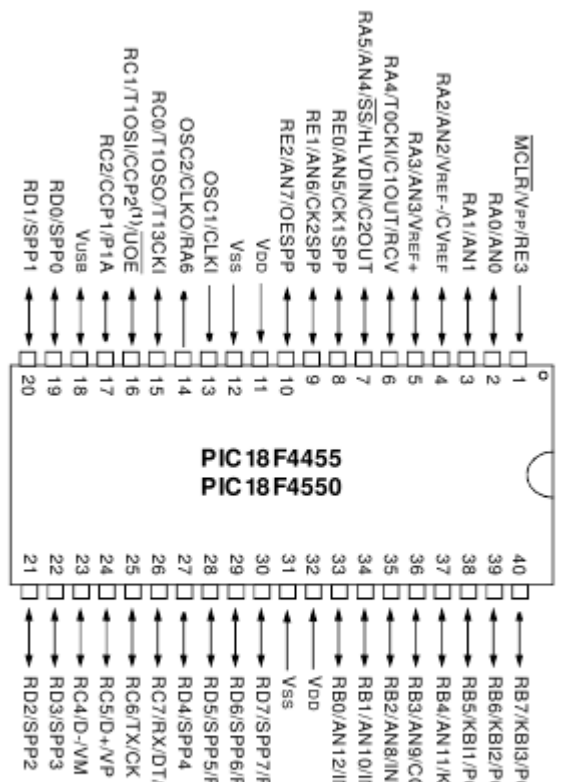
Microcontroller

- System-on-a-chip
  - Processor
  - Memory
  - Input/Output peripherals
  - Communication
  - Safety components



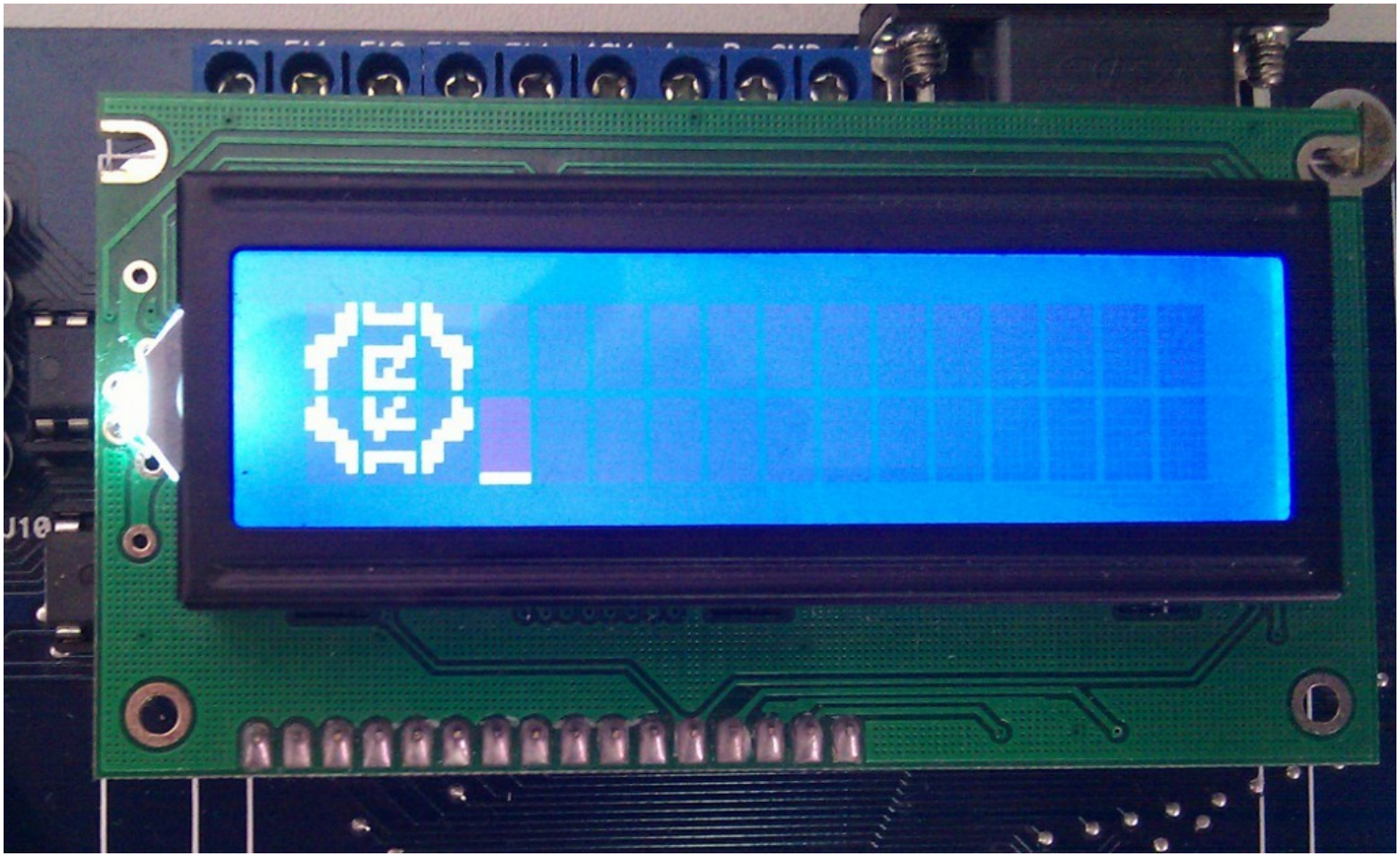
Microcontroller

- Xtal configuration
- Reset pin
- DC needs
- Many peripherals on the same pin



DEFCON

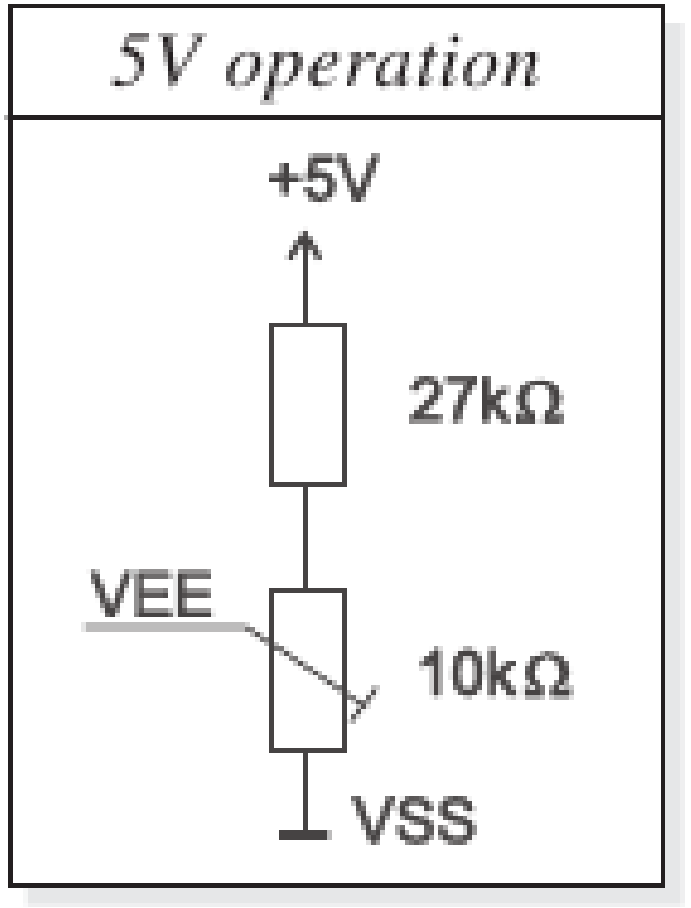
LCD Display



DEFCON

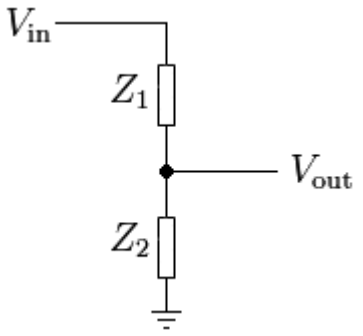
LCD Display

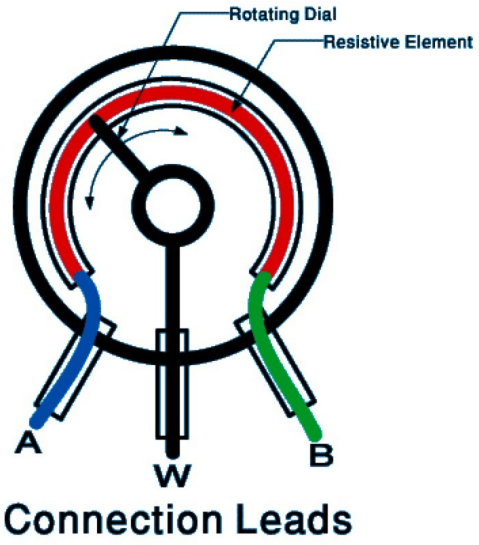
- Data connection
- Backlite
- Current consumption
- Power on time/routine



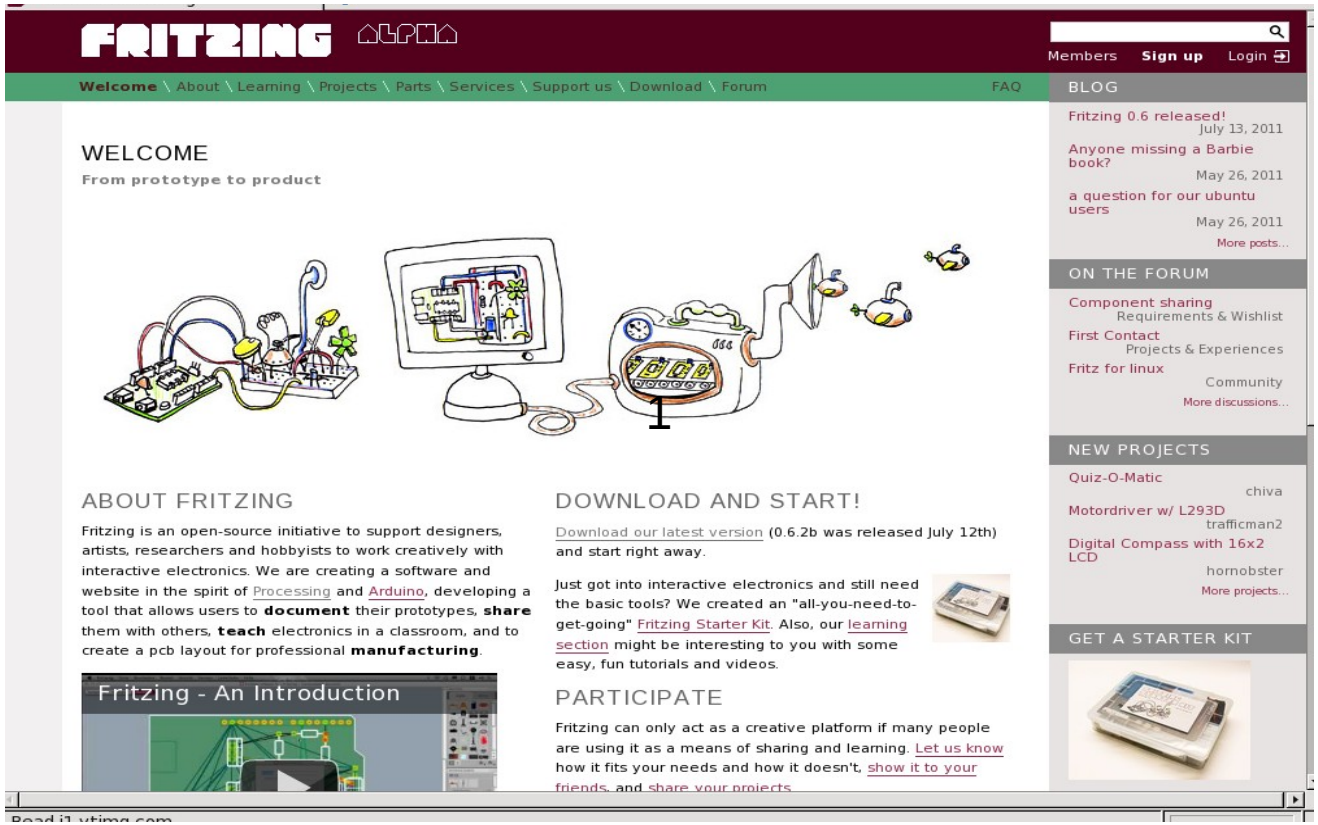
Potentiometer

- Linear/Log
- Used as voltage divisor
- Need an analog input
- Filter





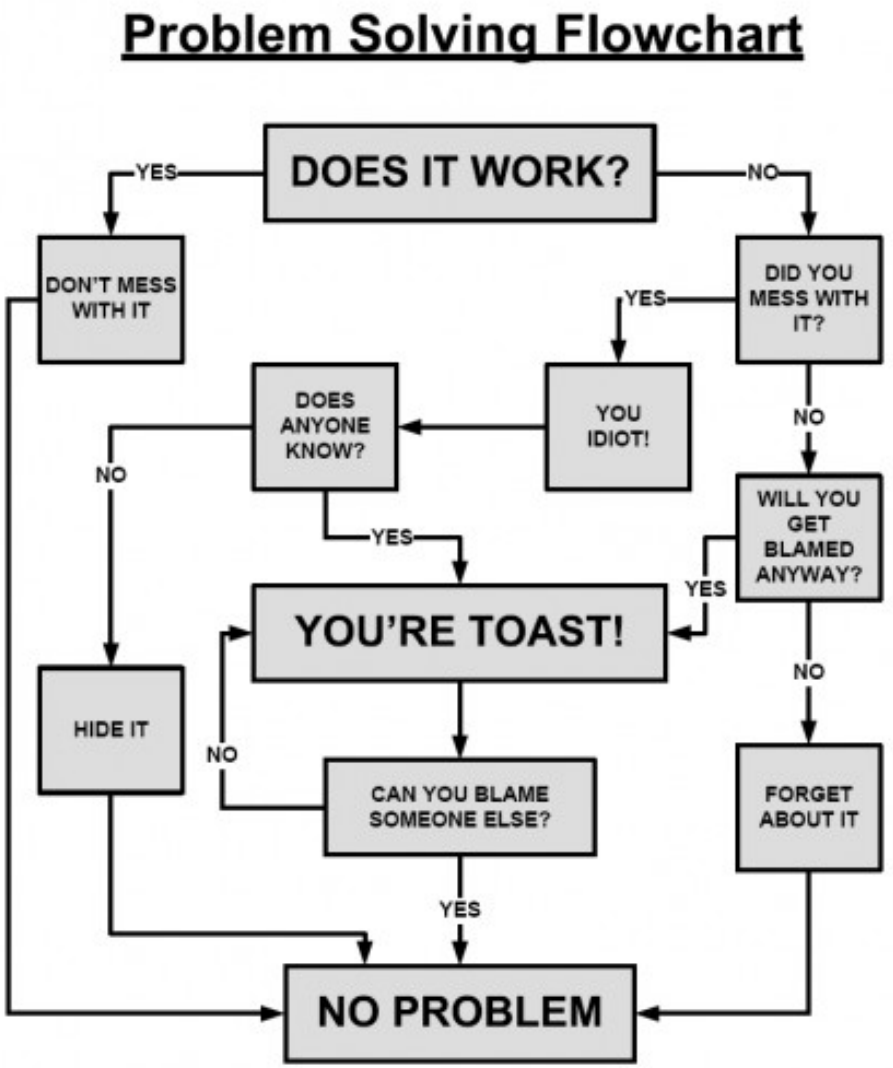
System design



- Cad tool: Fritzing (fritzing.org)

DEFCON

- Hands On!


Problem Solving Flowchart
DOES IT WORK?
YES
NO
DON'T MESS WITH IT
DID YOU MESS WITH IT?
YES
NO
DOES ANYONE KNOW?
YOU IDIOT!
NO
YES
WILL YOU GET BLAMED ANYWAY?
YES
YOU'RE TOAST!
NO
HIDE IT
NO
CAN YOU BLAME SOMEONE ELSE?
FORGET ABOUT IT
YES
NO PROBLEM

Board programming

- Programmer
- IDE
- Basic concepts
  - CPU Architecture
  - HW configuration
  - Memory access
- First program (He110 DEFC0N)
- Peripheral setup
- Second program (ADC read)

Board programming

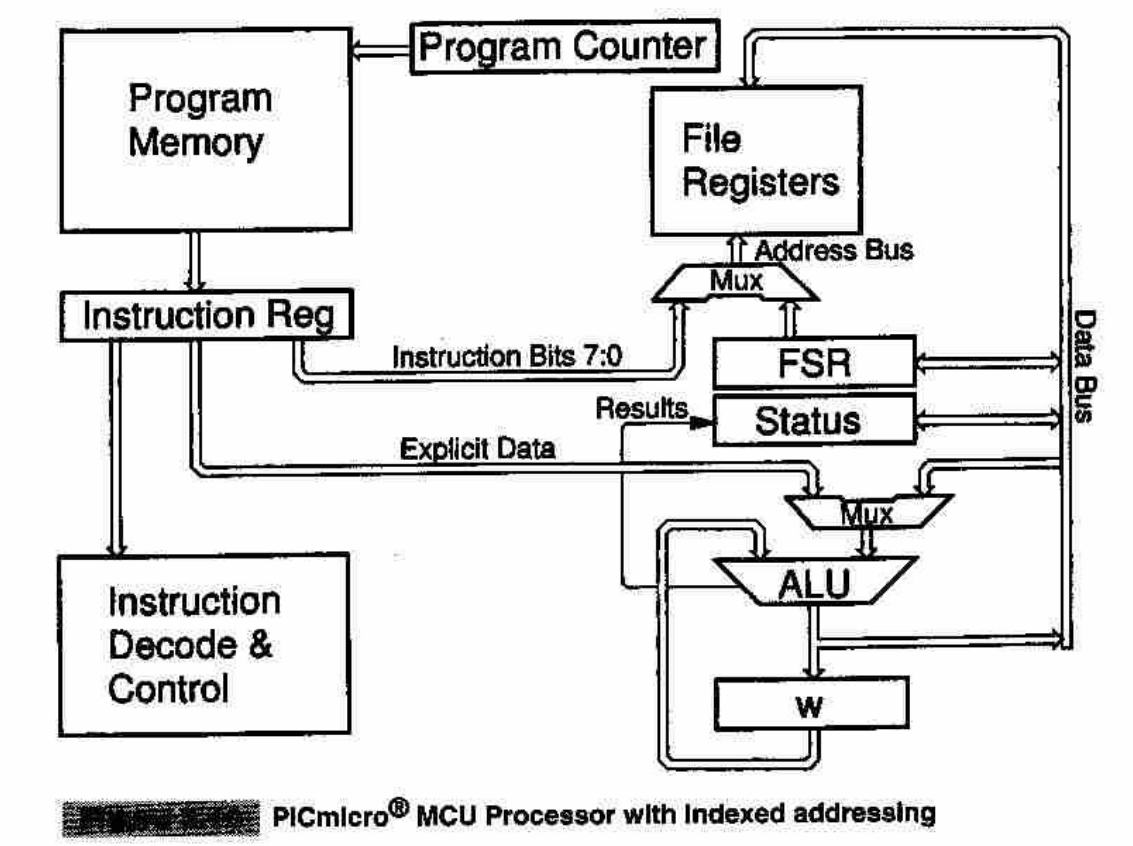
- Programmer
  - PICkit3
  - Can use ICSP
  - Can program a lot of Microchip products
  - Also a debugger

Board programming

- IDE
  - MPLABX
    - Based on Netbeans
- Compiler
  - SDCC
    - Based on GCC
  - GPUtils

Basic concepts

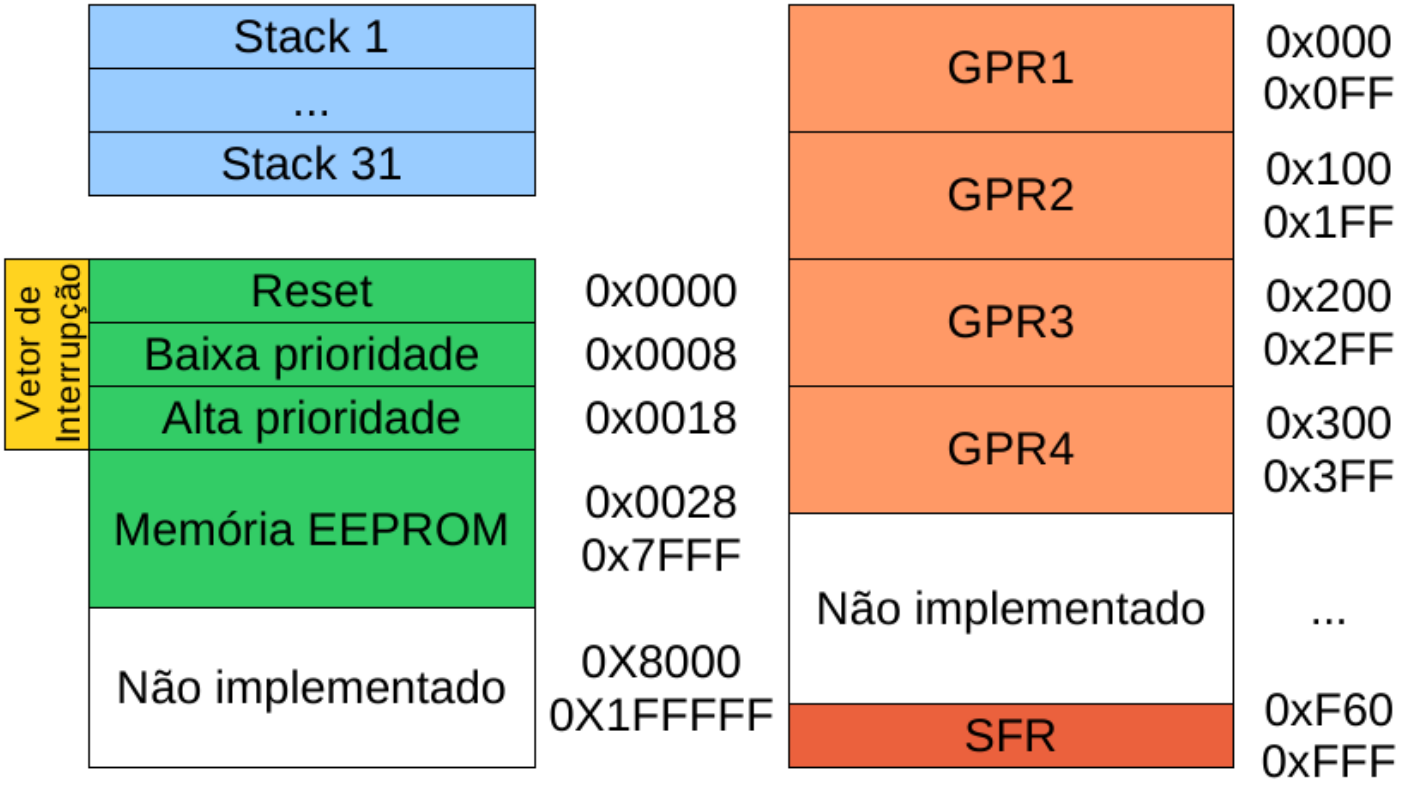
- PIC architecture



PICmicro® MCU Processor with Indexed addressing

Basic concepts

- Memory segmentation



Basic concepts

- HW configuration

TABLE 25-1: CONFIGURATION BITS AND DEVICE IDs

File Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default/ Unprogrammed Value
300000h	CONFIG1L	—	—	USBDIV	CPUDIV1	CPUDIV0	PLLDIV2	PLLDIV1	PLLDIV0	--00 0000
300001h	CONFIG1H	IESO	FCMEN	—	—	FOSC3	FOSC2	FOSC1	FOSC0	00-- 0101
300002h	CONFIG2L	—	—	VREGEN	BORV1	BORV0	BOREN1	BOREN0	$\overline{\text{PWRTEN}}$	--01 1111
300003h	CONFIG2H	—	—	—	WDTPS3	WDTPS2	WDTPS1	WDTPS0	WDTEN	---1 1111
300005h	CONFIG3H	MCLRE	—	—	—	—	LPT1OSC	PBADEN	CCP2MX	1--- -011
300006h	CONFIG4L	$\overline{\text{DEBUG}}$	XINST	ICPRT(3)	—	—	LVP	—	STVREN	100- -1-1
300008h	CONFIG5L	—	—	—	—	CP3(1)	CP2	CP1	CP0	---- 1111
300009h	CONFIG5H	CPD	CPB	—	—	—	—	—	—	11-- ----
30000Ah	CONFIG6L	—	—	—	—	WRT3(1)	WRT2	WRT1	WRT0	---- 1111
30000Bh	CONFIG6H	WRTD	WRTB	WRTC	—	—	—	—	—	111- ----
30000Ch	CONFIG7L	—	—	—	—	EBTR3(1)	EBTR2	EBTR1	EBTR0	---- 1111
30000Dh	CONFIG7H	—	EBTRB	—	—	—	—	—	—	-1-- ----
3FFFFEh	DEVID1	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	xxxx xxxx(2)
3FFFFFh	DEVID2	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	0001 0010(2)

Legend: x = unknown, u = unchanged, - = unimplemented. Shaded cells are unimplemented, read as '0'.

Note 1: Unimplemented in PIC18FX455 devices; maintain this bit set.

2: See Register 25-13 and Register 25-14 for DEVID values. DEVID registers are read-only and cannot be programmed by the user.

3: Available only on PIC18F4455/4550 devices in 44-pin TQFP packages. Always leave this bit clear in all other devices.

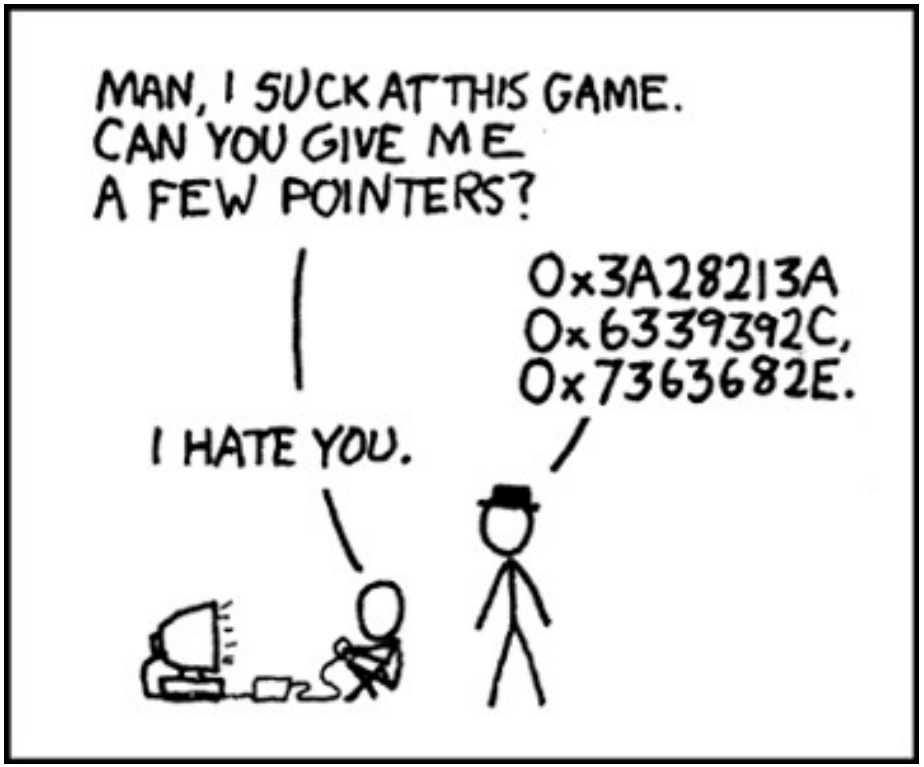
Basic concepts

- HW configuration
  - Some options must be set **before** the program start
  - This can only be accomplished by special instructions
  - Compiler datasheet

Basic concepts

```
//CONFIG.H

  // No prescaler used
__code char __at 0x300000 CONFIG1L = 0x01;
  // HS: High Speed Cristal
__code char __at 0x300001 CONFIG1H = 0x0C;
  // Disabled-Controlled by SWDTEN bit
__code char __at 0x300003 CONFIG2H = 0x00;
  // Disabled low voltage programming
__code char __at 0x300006 CONFIG4L = 0x00;
```

Basic concepts



Basic concepts

- Build a pointer to a specific memory address:

```
void main (void){
    char *ptr;
    //pointing to the port D
    ptr = 0xF83;
    //changing all outputs to high
    *ptr = 0xFF;
}
```

Basic concepts

- Building a header with all definitions
  - __near = sfr region
  - Volatile = can change without program acknowledge

```
//BASIC.H
#define PORTD   (*(volatile __near unsigned char*)0xF83)
#define TRISC   (*(volatile __near unsigned char*)0xF94)


 //this is not an infinite loop!
while(PORTD==PORTD);
```

First program

- Open MPLABX IDE
  - configure SDCC and PICkit
- Create a project to:
  - Initialize LCD
  - Print "`He110 DEFC0N`"

LCD communication

- The data is always an 8 bit information
  - It may be split in two 4 bit "passes"
- The data may represent a character or a command
  - They are distinguished by RS pin
- The data must be stable for "some time"
  - In this period the EN pin must be set

LCD communication

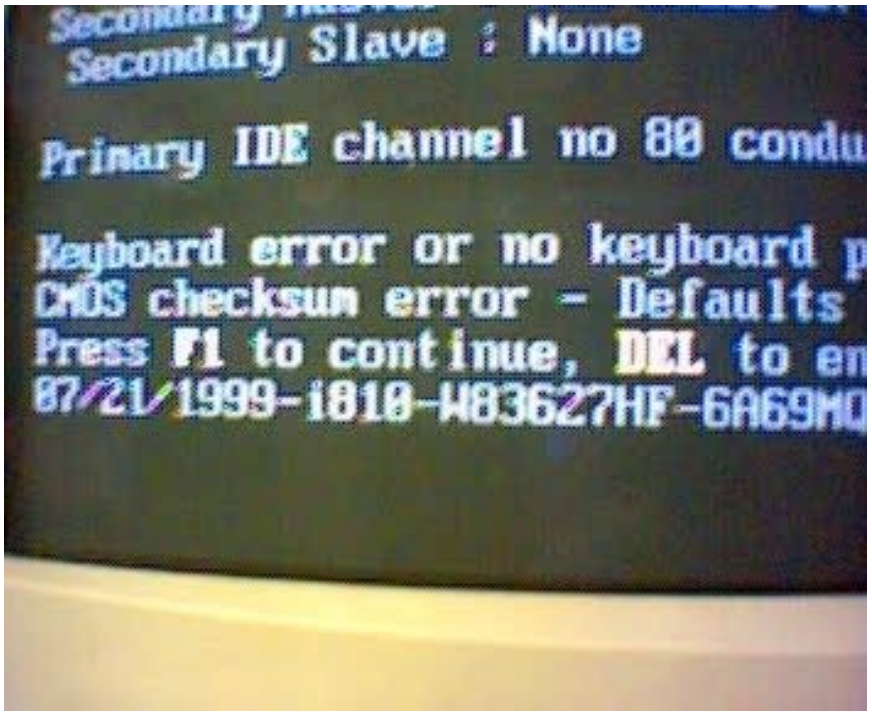
```
void LCD_comm(unsigned char data, char cmd){
    if (cmd)
        BitClr(PORTE,RS);
    else
        BitSet(PORTE,RS);
    BitClr(PORTE,RW); // writing
    PORTD = cmd;
    BitSet(PORTE,EN); //enable read
    Delay40ms();
    BitClr(PORTE,EN); //finish read
}
```

LCD communication

```
void LCD_init(void){
    char i;
    for (i=0; i<200; i++)
       Delay2ms(); // garante inicialização do LCD
// terminal configuration
    BitClr(TRISE,RS);    //RS
    BitClr(TRISE,EN);    //EN
    BitClr(TRISE,RW);    //RW
    TRISD = 0x00;        //data
//display initialization
    EnviaComando(0x38);  //8bits, 2 lines, 5x8
    EnviaComando(0x06);  //increment mode
    EnviaComando(0x0F);  //cursor ON
    EnviaComando(0x03);  //clear internal counters
    EnviaComando(0x01);  //clean display
    EnviaComando(0x80);  //initial position
}
```

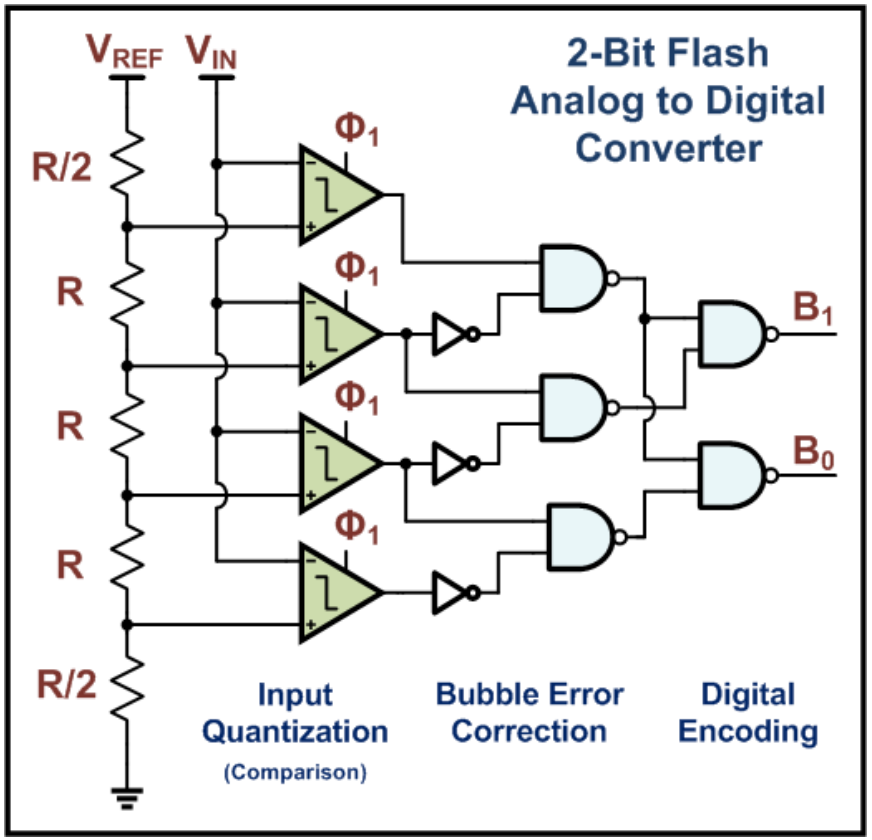
Peripherals setup

- Get on the datasheet the information about the peripheral registers and configuration info



Peripherals setup

- ADC



ADC setup

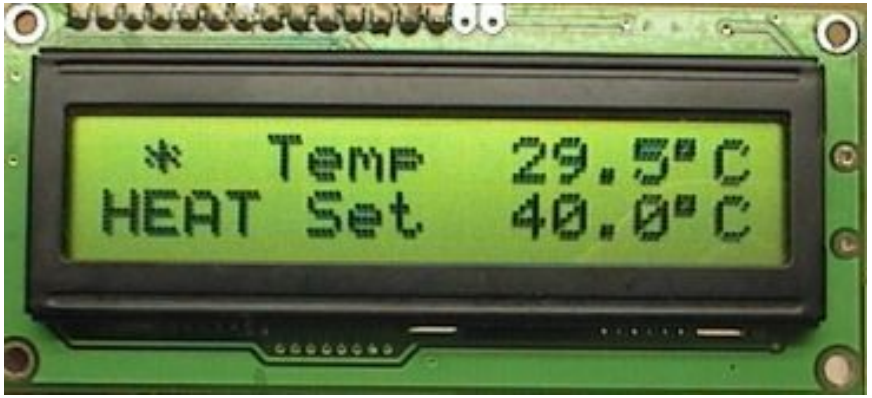
```
void ADC_init(void)
{
  BitSet(TRISA,0);       //pin setup
  ADCON0 = 0b00000001; //channel select
  ADCON1 = 0b00001110; //ref = source
  ADCON2 = 0b10101010; //t_conv = 12 TAD
}
```

ADC setup

```
unsigned int ADC_read(void)
{
  unsigned int ADvalue;
  BitSet(ADCON0,1);          //start conversion
  while(BitTst(ADCON0,1)); //wait
  ADvalue = ADRESH;          //read result
  ADvalue <<= 8;
  ADvalue += ADRESL;
  return ADvalor;
}
```

Second program

- Read ADC value and present in LCD



A little break to follow 3-2-1 DEFC0N rule!

Could not found any picture of Garfield taking shower =/

DEFCON

//today topics

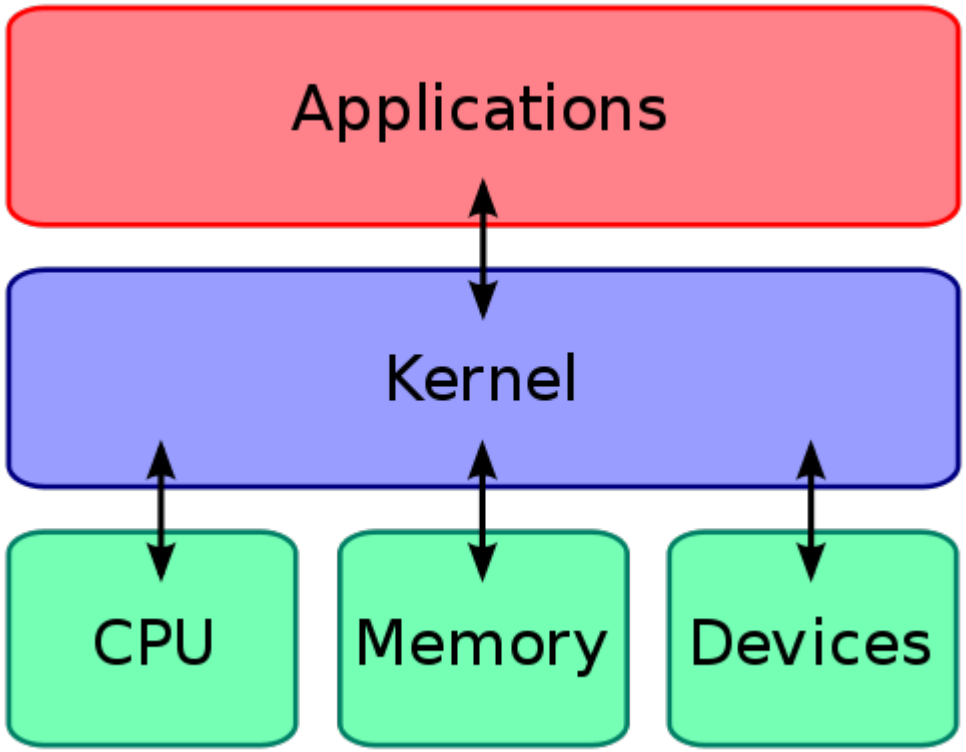
```
void main (void){
 //variable declaration
    kernel_project(1);
 //initialization
    concepts(2);
 //hard-work
    microkernel(3);
    device_driver_controller(4);
}
```

```
void kernel_project (float i){
    what_is_a_kernel(1.1);
    alternatives(1.2);
    monolithic_vs_microkernel(1.3);
    kernel_design_decisions(1.4);
    this_course_decisions(1.5);
}
```

```
kernel_project(1);
```

```
what_is_a_kernel(1.1);
```

kernel_project(1);



kernel_project(1);

- Kernel tasks:
  1) Manage and coordinate the processes execution using “some criteria”
  2) Manage the free memory and coordinate the processes access to it
  3) Intermediate the communication between the hardware drivers and the processes

```
kernel_project(1);
```

Develop my own kernel?

Why?

kernel_project(1);

- Improve home design
- Reuse code more efficiently
- Full control over the source
- Specific tweeks to the kernel
  - Faster context switch routine
  - More control over driver issues (interrupts)

```
kernel_project(1);
```

Develop my own kernel?

Why not?

kernel_project(1);

- Kernel overhead (both in time and memory)
- Free and paid alternatives
- Time intensive project
- Continuous development

kernel_project(1);

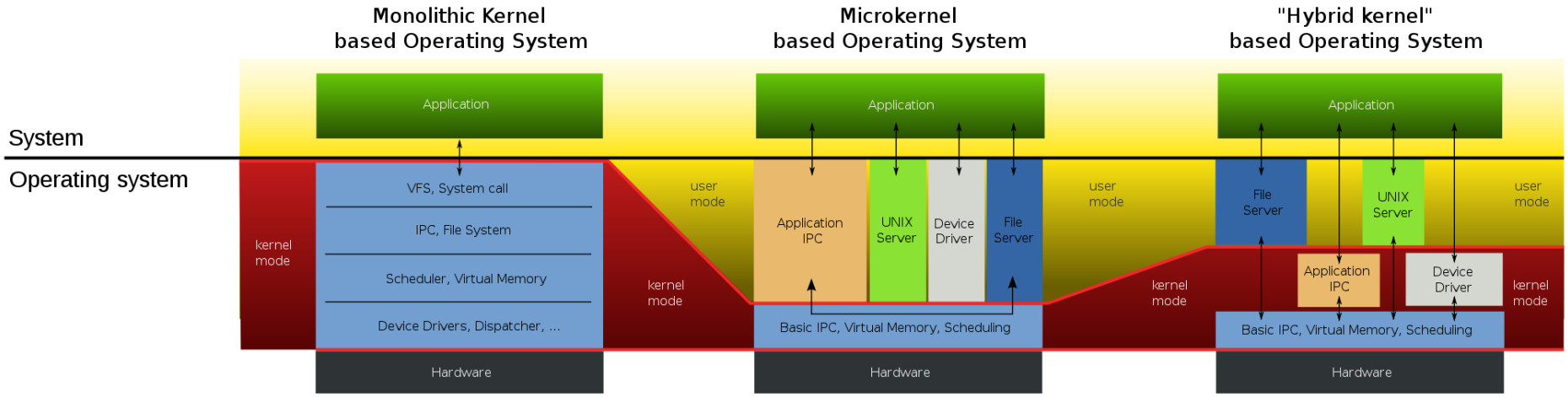
DEFCON

kernel_project(1);

- Alternatives
  - Windows Embedded Compact®
  - VxWorks®
  - X RTOS®
  - uClinux
  - FreeRTOS
  - BRTOS

kernel_project(1);

- Monolithic kernel versus microkernel



- Linus Torvalds and Andrew Tanenbaum

DEFCON

kernel_project(1);

- Kernel design decisions
  - I/O devices management
  - Process management
  - System safety

kernel_project(1);

- Our decisions:
  - Microkernel
  - Non-preemptive
  - Cooperative
  - No memory management
  - Process scheduled based on timer
  - Isolate drivers using a controller

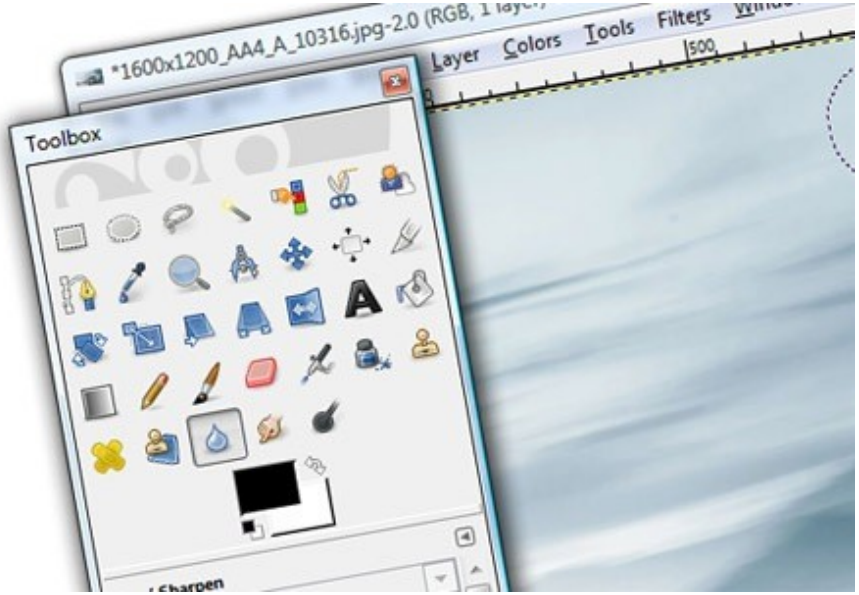
```
void concepts (float i){
    function_pointers(2.1);
    structs(2.2);
    circular_buffers(2.3);
    temporal_conditions(2.4);
    void_pointers(2.5);
}
```

```
concepts(2);
```

```
function_pointers(2.1);
```

concepts(2);

- Necessity:
  - Make an image editor that can choose the right function to call
- 1st Implementation
  - Use a option parameter as a switch operator



concepts(2);

```
image Blur(image nImg){}
image Sharpen(image nImg){}

image imageEditorEngine(image nImg, int opt){
  image temp;
  switch(opt){
    case 1:
      temp = Sharpen(nImg);
      break;
    case 2:
      temp = Blur(nImg);
      break;
  }
  return temp;
}
```

concepts(2);

- Function pointers
  - Work almost as a normal pointer
  - Hold the address of a function start point instead the address of a variable
  - The compiler need no known the function signature to pass the correct parameters and the return value.
  - Awkard declaration (it is best to use a typedef)

concepts(2);

```
//defining the type pointerTest
//it is a pointer to function that:
//   receives no parameter
//   returns no parameter
typedef void (*pointerTest)(void);

//Function to be called
void nop (void){ __asm NOP __endasm }

//creating an pointerTest variable;
pointerTest foo;
foo = nop;
(*foo)(); //calling the function via pointer
```

concepts(2);

Re-code the image editor engine using function pointers

concepts(2);

```
image Blur(image nImg){}

image Sharpen(image nImg){}

typedef image (*ptrFunc)(image nImg);

//image editor engine
image imageEditorEngine(ptrFunc function,
                        image nImg){
   image temp;
   temp = (*function)(nImg);
   return temp;
}
```

concepts(2);

- Good
  - New function additions do not alter the engine
  - The engine only needs to be tested once
  - Can change the function implementations dynamically
- Bad
  - More complex code (function pointers are not so easy to work with)
  - Not all compilers support function pointers

concepts(2);

structs(2.2);

concepts(2);

- Structs are composed variables.
- Group lots of information as if they were one single variable.
- A vector that each position stores a different type

```
// struct declaration
typedef struct{
   unsigned short int age;
   char name[51];
   float weight;
}people;
```

concepts(2);

```
void main(void){
   struct people myself = {26, "Rodrigo", 70.5};

    myself.age = 27;

   //using each variable from the struct
   printf("Age:    %d\n", myself.age);
   printf("Name:   %s\n", myself.name);
   printf("Weight: %f\n", myself.weight);

   return 0;
}
```

concepts(2);

```
 // struct declaration
typedef struct{
   unsigned short int *age;
   char *name[51];
   float *weight;
}people;

void main(void){
   struct people myself = {26, "Rodrigo", 70.5};
   //using each variable from the struct
   printf("Age:    %d\n", myself->age);
   printf("Name:   %s\n", myself->name);
   printf("Weight: %f\n", myself->weight);
   return 0;
}
```

```
concepts(2);
```

```
circular_buffers(2.3);
```

concepts(2);

- Circular Buffers
  - “Endless” memory spaces
  - Use FIFO aproach
  - Store temporary data
  - Can implemented using vectors or linked-lists

concepts(2);

- Vector implementation
  - Uses less space
  - Need special caution when cycling
  - Problem to differentiate full from empty

concepts(2);

```
#define CB_SIZE 10

int circular_buffer[CB_SIZE];

int index=0;

for(;;){
   //do anything with the buffer
   circular_buffer[index] = index;
   //increment the index
   index = (index+1)%CB_SIZE;
}
```

concepts(2);

```
#define CB_SIZE 10
int circular_buffer[CB_SIZE];
int start=0, end=0;

char AddBuff(int newData)
{
  //check if there is space to add a number
  if ( ((end+1)%CB_SIZE) != start)
  {
    circular_buffer[end] = newData;
    end = (end+1)%CB_SIZE;
    return SUCCESS;
  }
  return FAIL;
}
```

concepts(2);

temporal_conditions(2.4);

concepts(2);

In the majority part of embedded systems, we need to guarantee that a function will be executed in a certain frequency. Some systems may even fail if these deadlines are not met.

concepts(2);

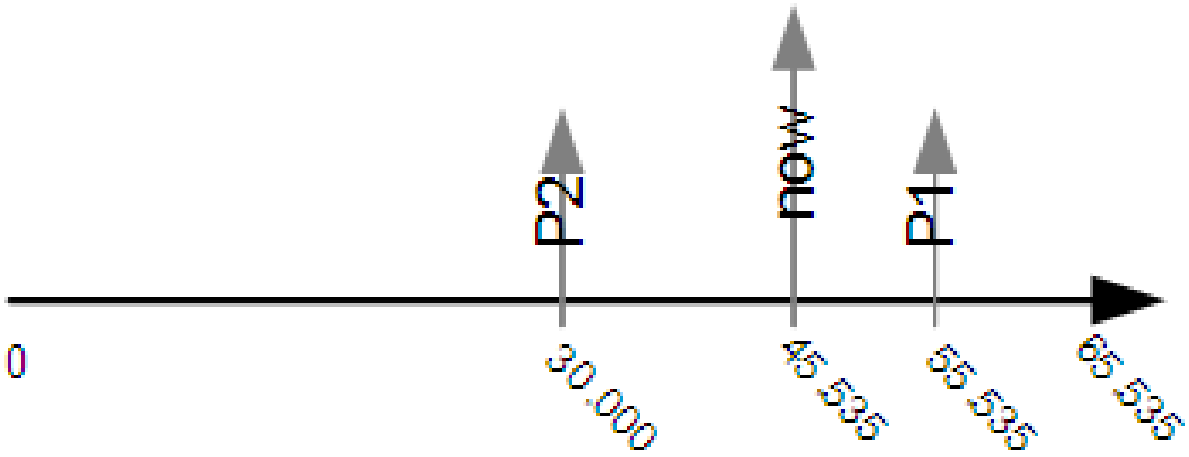
- To implement temporal conditions:
  1) There must be a tick event that occurs with a precise frequency
  2) The kernel must be informed of the execution frequency needed for each process.
  3) The sum of process duration must “fit” within the processor available time.

concepts(2);

- 1st condition:
  - Needs an internal timer that can generate an interrupt.
- 2nd condition:
  - Add the information for each process when creating it
- 3rd condition:
  - Test, test and test.
  - If fail, change chip first, optimize only on last case

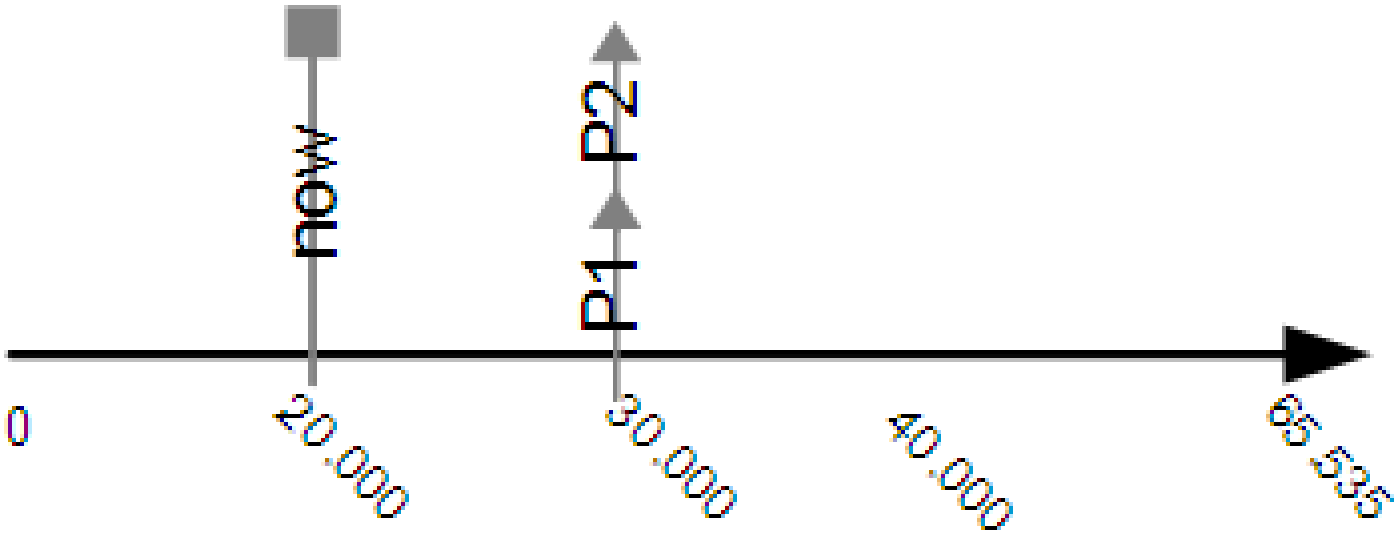
concepts(2);

- Scheduling processes:
  - Using a finite timer to schedule will result in overflow
  - Example: scheduling 2 processes for 10 and 50 seconds ahead.

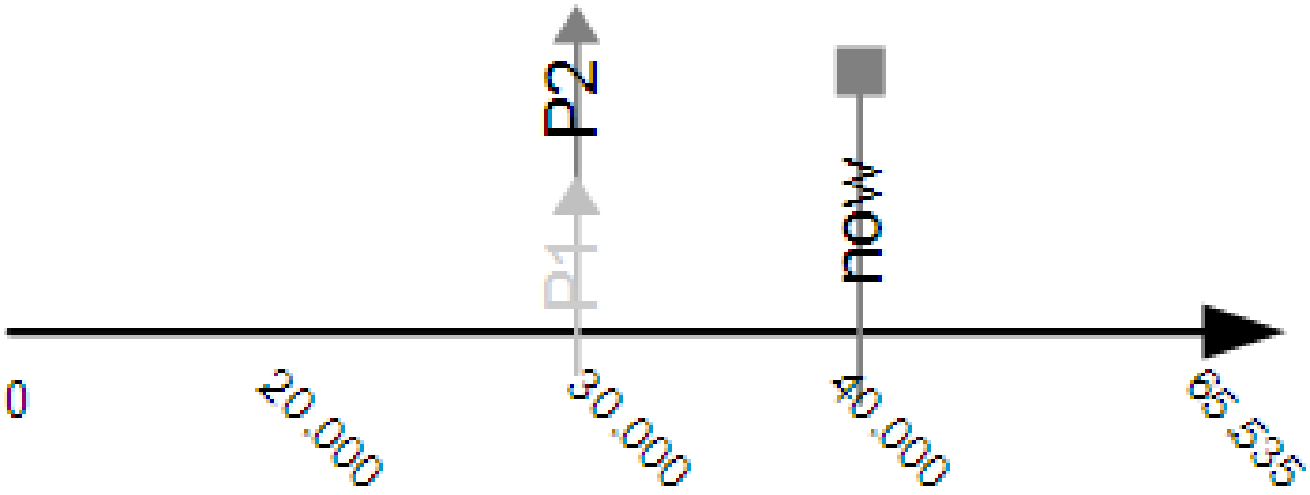


concepts(2);

- And if two process are to be called in the same time?



concepts(2);



- Question:
  - From the timeline above (only the timeline) is P2 late or it was scheduled to happen 55(s) from now?

concepts(2);

- Solution:
  - Use a downtime counter for each process instead of setting a trigger time.
- Problem:
  - Each counter must be decremented in the interrupt subroutine.
  - Is it a problem for your system?

```
concepts(2);
```

```
void_pointers(2.5);
```

concepts(2);

- Void pointers
  - Abstraction that permits to the programmer to pass parameters with different types to the same function.
  - The function which is receiving the parameter must know how to deal with it
  - It can not be used without proper casting!

concepts(2);

```
char *name = "Paulo";

double weight = 87.5;

unsigned int children = 3;

void main (void){
    //its not printf, yet.
    print(0, &name);
    print(1, &weight);
    print(2, &children);
}
```

concepts(2);

```
void print(int option; void *parameter){
  switch(option){
     case 0:
        printf("%s",(char*)parameter);
     break;
     case 1:
        printf("%f",*((double*)parameter));
     break;
     case 2:
        printf("%d",*((unsigned int*)parameter));
     break;
  }
}
```

```
void microkernel (float i){
    init_kernel(3.0);
    for(int i=1; i<4; i++;)
    {
        kernel_example(3+i/10);
    }
    running_the_kernel(3.4);
}
```

```
microkernel(3);
```

```
init_kernel(3.0);
```

microkernel(3);

- The examples will use a minimum of hardware or platform specific commands.
- Some actions (specifically the timer) needs hardware access.

```
microkernel(3);
```

```
//first implementation
kernel_example(3+1/10);
```

microkernel(3);

- In this first example we will build the main part of our kernel.
- It should have a way to store which functions are needed to be executed and in which order.
- This will be done by a static vector of pointers to function

```
//pointer function declaration
typedef void(*ptrFunc)(void);
//process pool
static ptrFunc pool[4];
```

microkernel(3);

- Each process is a function with the same signature of ptrFunc

```
void tst1(void){
   printf("Process 1\n");
}
void tst2(void){
   printf("Process 2\n");
}
void tst3(void){
   printf("Process 3\n");
}
```

microkernel(3);

- The kernel itself consists of three functions:
  - One to initialize all the internal variables
  - One to add a new process
  - One to execute the main kernel loop

```
//kernel internal variables
ptrFunc pool[4];
int end;
//kernel function's prototypes
void kernelInit(void);
void kernelAddProc(ptrFunc newFunc);
void kernelLoop(void);
```

microkernel(3);

```
//kernel function's implementation

void kernelInit(void){
   end = 0;
}

void kernelAddProc(ptrFunc newFunc){
   if (end <4){
      pool[end] = newFunc;
      end++;
   }
}
```

microkernel(3);

```
//kernel function's implementation

void kernelLoop(void){
    int i;
    for(;;){
        //cycle through the processes
        for(i=0; i<end; i++){
            (*pool[i])();
        }
    }
}
```

microkernel(3);

```
//main loop

void main(void){

   kernelInit();

   kernelAddProc(tst1);
   kernelAddProc(tst2);
   kernelAddProc(tst3);

   kernelLoop();
}
```

`microkernel(3);`

Simple?

```
microkernel(3);
```

```
//second implementation
//circular buffer and struct added
kernel_example(3+2/10);
```

microkernel(3);

- The only struct field is the function pointer. Other fields will be added latter.
- The circular buffer open a new possibility:
  - A process now can state if it wants to be rescheduled or if it is a one-time run process
  - In order to implement this every process must return a code.
  - This code also says if there was any error in the process execution

microkernel(3);

```
//return code
#define SUCCESS   0
#define FAIL      1
#define REPEAT    2

//function pointer declaration
typedef char(*ptrFunc)(void);

//process struct
typedef struct {
   ptrFunc function;
} process;

process pool[POOL_SIZE];
```

microkernel(3);

```
char kernelInit(void){
   start = 0;
   end = 0;
   return SUCCESS;
}
char kernelAddProc(process newProc){
   //checking for free space
   if ( ((end+1)%POOL_SIZE) != start){
      pool[end] = newProc;
      end = (end+1)%POOL_SIZE;
      return SUCCESS;
   }
   return FAIL;
}
```

microkernel(3);

```
void kernelLoop(void){
   int i=0;
   for(;;){
      //Do we have any process to execute?
      if (start != end){
         printf("Ite. %d, Slot. %d: ", i, start);
         //check if there is need to reschedule
         if ((*(pool[start].Func))() == REPEAT){
            kernelAddProc(pool[start]);
         }
         //prepare to get the next process;
         start = (start+1)%POOL_SIZE;
         i++; // only for debug;
      }
   }
}
```

microkernel(3);

```
//vetor de struct
process pool[POOL_SIZE];
//    pool[i]
//    pool[i].func
//  *(pool[i].func)
// (*(pool[i].func))()


//vetor de ponteiro de struct
process* pool[POOL_SIZE];
//    pool[i]
//    pool[i].func
//    pool[i]->func
//    pool[i]->func()
```

microkernel(3);

```
void kernelLoop(void){
   int i=0;
   for(;;){
      //Do we have any process to execute?
      if (start != end){
         printf("Ite. %d, Slot. %d: ", i, start);
         //check if there is need to reschedule
         if (pool[start]->Func() == REPEAT){
            kernelAddProc(pool[start]);
         }
         //prepare to get the next process;
         start = (start+1)%POOL_SIZE;
         i++; // only for debug;
      }
   }
}
```

microkernel(3);

- Presenting the new processes

```
void tst1(void){
   printf("Process 1\n");
   return REPEAT;
}
void tst2(void){
   printf("Process 2\n");
   return SUCCESS;
}
void tst3(void){
   printf("Process 3\n");
   return REPEAT;
}
```

microkernel(3);

```
void main(void){
   //declaring the processes
   process p1 = {tst1};
   process p2 = {tst2};
   process p3 = {tst3};
   kernelInit();
   //Test if the process was added successfully
   if (kernelAddProc(p1) == SUCCESS){
      printf("1st process added\n");}
   if (kernelAddProc(p2) == SUCCESS){
      printf("2nd process added\n");}
   if (kernelAddProc(p3) == SUCCESS){
      printf("3rd process added\n");}
   kernelLoop();
}
```

microkernel(3);

- Notes:
  - Only process 1 and 3 are repeating
  - The user don't notice that the pool is finite*

```
Console Output:
----------------------------
1st process added
2nd process added
3rd process added
Ite. 0, Slot. 0: Process 1
Ite. 1, Slot. 1: Process 2
Ite. 2, Slot. 2: Process 3
Ite. 3, Slot. 3: Process 1
Ite. 4, Slot. 0: Process 3
Ite. 5, Slot. 1: Process 1
Ite. 6, Slot. 2: Process 3
Ite. 7, Slot. 3: Process 1
Ite. 8, Slot. 0: Process 3
...
----------------------------
```

```
microkernel(3);
```

```
//third implementation
//time conditions added
kernel_example(3+3/10);
```

microkernel(3);

- The first modification is to add one counter to each process

```
//process struct
typedef struct {
   ptrFunc function;
   int period;
   int start;
} process;
```

microkernel(3);

- We must create an function that will run on each timer interrupt updating the counters

```
void isr(void) interrupt 1{
unsigned char i;
   i = ini;
   while(i!=fim){
      if((pool[i].start)>(MIN_INT)){
         pool[i].start--;
      }
      i = (i+1)%SLOT_SIZE;
   }
}
```

microkernel(3);

- The add process function will be the responsible to initialize correctly the fields

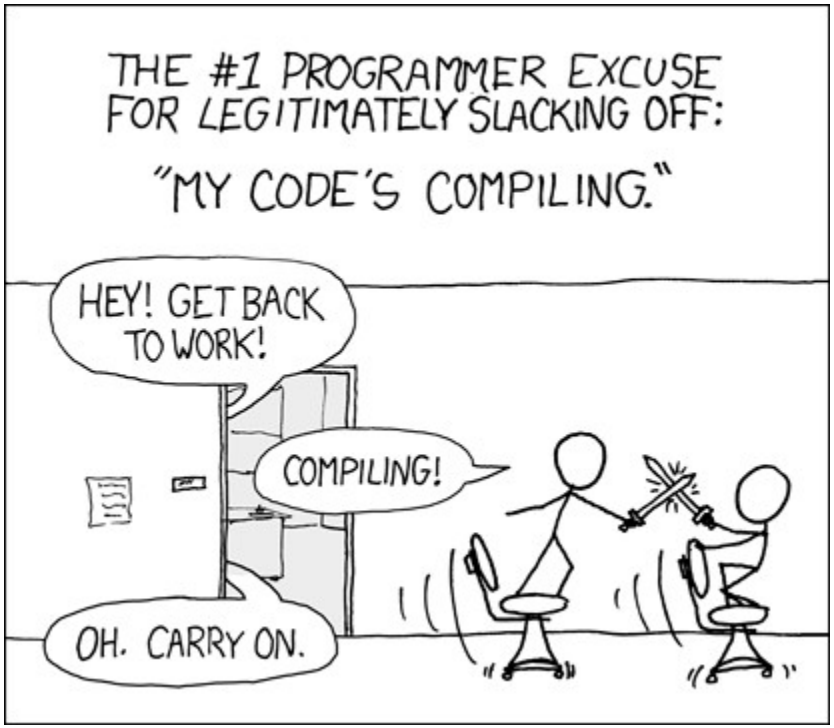
```
char AddProc(process newProc){
   //checking for free space
   if ( ((end+1)%SLOT_SIZE) != start){
      pool[end] = newProc;
      //increment start timer with period
      pool[end].start += newProc.period;
      end = (end+1)%SLOT_SIZE;
      return SUCCESS;
   }
   return FAIL;
}
```

microkernel(3);

```
if (start != end){
    //Finding the process with the smallest start
    j = (start+1)%SLOT_SIZE;
    next = start;
    while(j!=end){
        if (pool[j].start < pool[next].start){
            next = j;
        }
        j = (j+1)%SLOT_SIZE;
    }
    //exchanging positions in the pool
    tempProc = pool[next];
    pool[next] = pool[start];
    pool[start] = tempProc;
    while(pool[start].start>0){
    }//great place to use low power mode
    if ( (*(pool[ini].function))() == REPEAT ){
        AddProc(&(vetProc[ini]));
    }
    ini = (ini+1)%SLOT_SIZE;
}
```

```
microkernel(3);
```

```
running_the_kernel(3.4);
```



“My board's programming” also works =)

```
void dd_controler (float i){
    device_driver_pattern(5.1);
    controller_engine(5.2);
    isr_abstract_layer(5.3);
    driver_callback(5.4);
}
```

```
device_driver_controller(4);
```

```
device_driver_pattern(5.1);
```

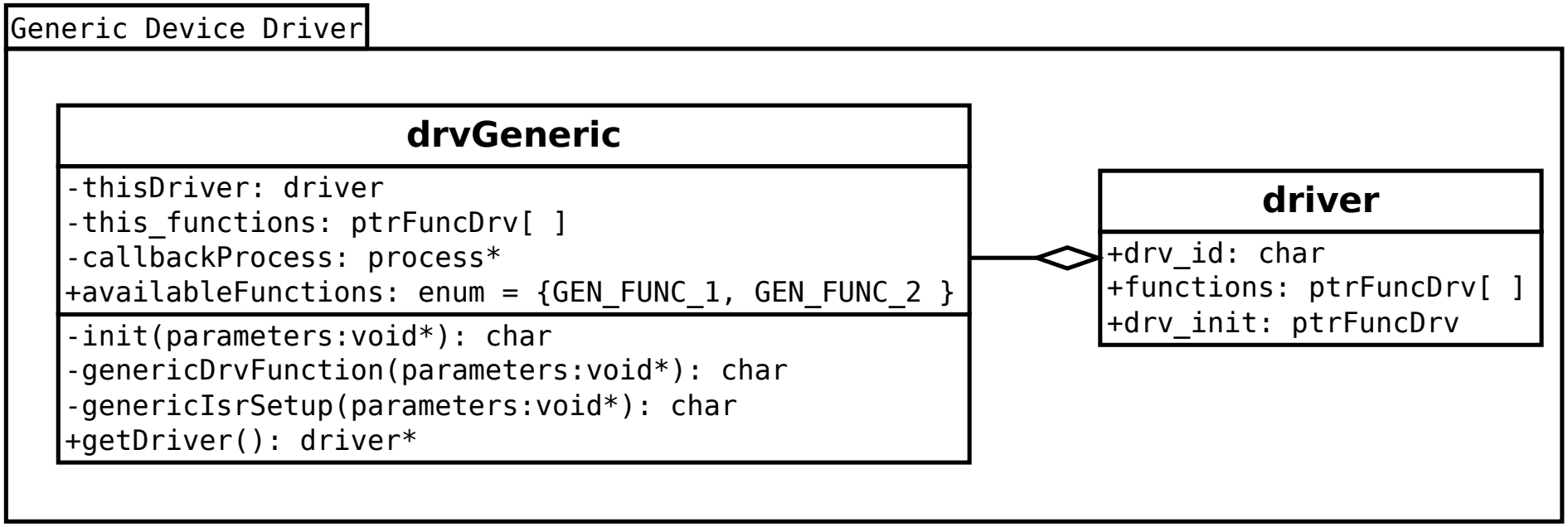
`device_driver_controller(4);`

- What is a driver?
  - An interface layer that translate hardware to software
- Device driver standardization
  - Fundamental for dynamic drivers load

`device_driver_controller(4);`

- Parameters problem
  - The kernel must be able to communicate in the same way with all drivers
  - Each function in each driver have different types and quantities of parameters
- Solution
  - Pointer to void

device_driver_controller(4);



```
device_driver_controller(4);
```

```
controller_engine(5.2);
```

`device_driver_controller(4);`

- Device Driver Controller
  - Used as an interface layer between the kernel and the drivers
  - Can “discover” all available drivers (statically or dynamically)
  - Store information about all loaded drivers
  - Responsible to interpret the messages received from the kernel

device_driver_controller(4);

```
char initDriver(char newDriver) {
  char resp = FAIL;

  if(dLoaded < QNTD_DRV) {
    //get driver struct
    drivers[dLoaded] = drvInitVect[newDriver]();

    //should test if driver was loaded correcly
    resp = drivers[dLoaded]->drv_init(&newDriver);
    dLoaded++;
  }
  return resp;
}
```

device_driver_controller(4);

```
char callDriver(char drv_id,
                char func_id,
                void *param) {
   char i;
   for (i = 0; i < dLoaded; i++) {
      //find the right driver
      if (drv_id == drivers[i]->drv_id) {

         return
  drivers[i]->func[func_id].func_ptr(param);

      }
   }
   return DRV_FUNC_NOT_FOUND;
}
```

device_driver_controller(4);

```
void main(void) {
   //system initialization
   //kernel also initializate the controller
   kernelInitialization();
   initDriver(DRV_LCD);
   callDriver(DRV_LCD, LCD_CHAR, 'D');
   callDriver(DRV_LCD, LCD_CHAR, 'E');
   callDriver(DRV_LCD, LCD_CHAR, 'F');
   callDriver(DRV_LCD, LCD_CHAR, 'C');
   callDriver(DRV_LCD, LCD_CHAR, '0');
   callDriver(DRV_LCD, LCD_CHAR, 'N');
   callDriver(DRV_LCD, LCD_CHAR, '@');
   callDriver(DRV_LCD, LCD_CHAR, 'L');
   callDriver(DRV_LCD, LCD_CHAR, 'A');
   callDriver(DRV_LCD, LCD_CHAR, 'S');
}
```

```
device_driver_controller(4);
```

Where are the defines?

device_driver_controller(4);

- In order to simplify the design, each driver build its function define enum.

```
enum {
    LCD_COMMAND, LCD_CHAR, LCD_INTEGER, LCD_END
};
```

- The controller builds a driver define enum

```
enum {
    DRV_INTERRUPT, DRV_TIMER, DRV_LCD, DRV_END
};
```

```
device_driver_controller(4);
```

```
isr_abstract_layer(5.3);
```

device_driver_controller(4);

- Interrupts are closely related to hardware
- Each architecture AND compiler pose a different programming approach

```
//SDCC compiler way
void isr(void) interrupt 1{
    thisInterrupt();
}
```

```
//C18 compiler way
void isr (void){
    thisInterrupt();
}
#pragma code highvector=0x08
void highvector(void){
    _asm goto isr _endasm
}
#pragma code
```

- How to hide this from programmer?

device_driver_controller(4);

```
//Inside drvInterrupt.c

//defining the pointer to use in ISR callback
typedef void (*intFunc)(void);

//store the pointer to ISR here
static intFunc thisInterrupt;

//Set interrupt function to be called
char setInterruptFunc(void *parameters) {
   thisInterrupt = (intFunc) parameters;
   return SUCESS;
}
```

device_driver_controller(4);

```
//Interrupt function set without knowing hard/compiler issues
void timerISR(void) {
    callDriver(DRV_TIMER, TMR_RESET, 1000);
    kernelClock();
}
void main (void){
    kernelInit();

    initDriver(DRV_TIMER);
    initDriver(DRV_INTERRUPT);

    callDriver(DRV_TIMER, TMR_START, 0);
    callDriver(DRV_TIMER, TMR_INT_EN, 0);
    callDriver(DRV_INTERRUPT, INT_TIMER_SET, (void*)timerISR);
    callDriver(DRV_INTERRUPT, INT_ENABLE, 0);

    kernelLoop();
}
```

```
device_driver_controller(4);
```

```
driver_callback(5.4);
```

`device_driver_controller(4);`

How to make efficient use of CPU peripherals without using pooling or hard-coding the interrupts?

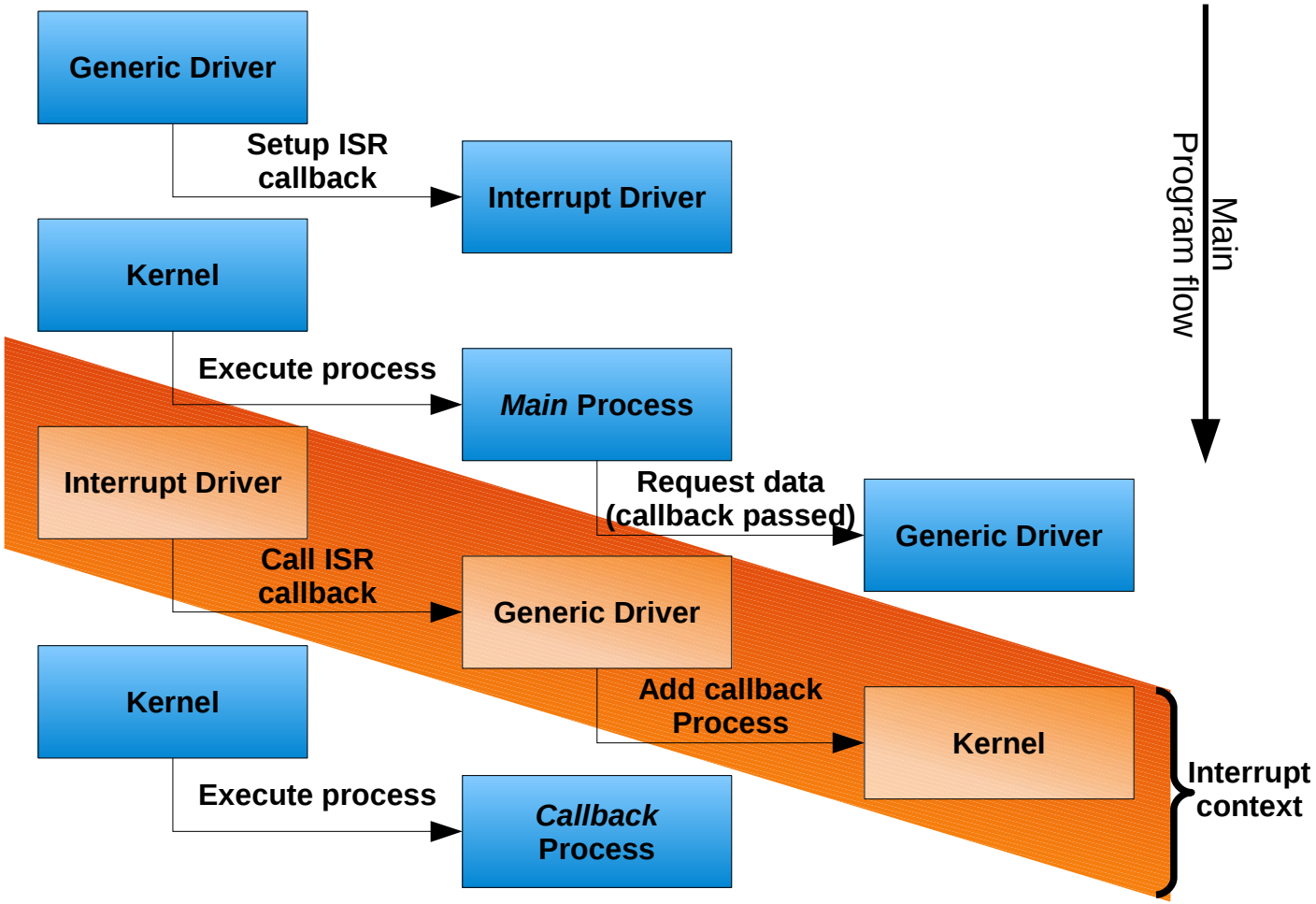
```
device_driver_controller(4);
```

Callback functions

`device_driver_controller(4);`

- Callback functions resemble events in high level programming
  - e.g.: When the mouse clicks in the button X, please call function Y.
- The desired hardware must be able to rise an interrupt
- Part of the work is done under interrupt context, preferable the faster part

device_driver_controller(4);



DEFCON

device_driver_controller(4);

```
//********** Excerpt from drvAdc.c **********
//  called from setup time to enable ADC interrupt
//  and setup ADC ISR callback
char enableAdcInterrup(void* parameters){
    callDriver(DRV_INTERRUPT,INT_ADC_SET,(void*)adcISR);
    BitClr(PIR1,6);
    return FIM_OK;
}

//********** Excerpt from drvInterrupt.c **********
//  store the pointer to the interrupt function
typedef void (*intFunc)(void);
static intFunc adcInterrupt;

//  function to set ADC ISR callback for latter use
char setAdcInt(void *parameters) {
    adcInterrupt = (intFunc)parameters;
    return FIM_OK;
}
```

device_driver_controller(4);

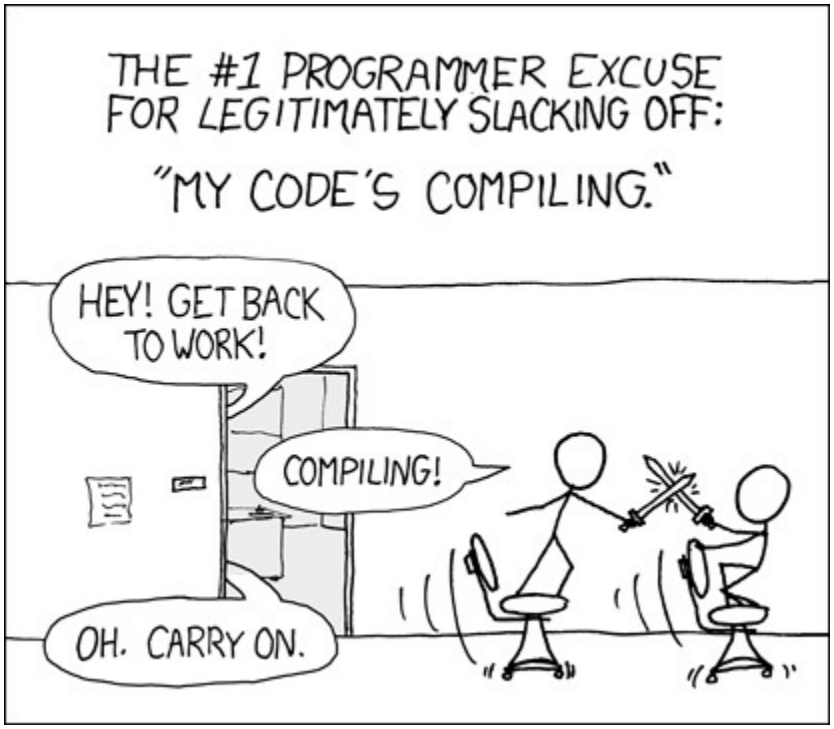
```
//********** Excerpt from main.c **********
//  Process called by the kernel
char adc_func(void) {
    //creating callback process
    static process proc_adc_callback = {adc_callback, 0, 0};
    callDriver(DRV_ADC,ADC_START,&proc_adc_callback);
    return REPEAT;
}

//********** Excerpt from drvAdc.c **********
//function called by the process adc_func (via drv controller)
char startConversion(void* parameters){
    callBack = parameters;
    ADCON0 |= 0b00000010;   //start conversion
    return SUCCESS;
}
```

device_driver_controller(4);

```
//********** Excerpt from drvInterrupt.c **********
//interrupt function
void isr(void) interrupt 1 {
    if (BitTst(INTCON, 2)) { //Timer overflow
    }
    if (BitTst(PIR1, 6)) {   //ADC conversion finished
       //calling ISR callback stored
       adcInterrupt();
    }
}
//********** Excerpt from drvAdc.c **********
//ADC ISR callback function
void adcISR(void){
    value = ADRESH;
    value <<= 8;
    value += ADRESL;
    BitClr(PIR1,6);
    kernelAddProc(callBack);
}
```

device_driver_controller(4);

```
//********** Excerpt from main.c **********
//callback function started from the kernel
char adc_callback(void) {
    unsigned int resp;
    //getting the converted value
    callDriver(DRV_ADC,ADC_LAST_VALUE,&resp);
    //changing line and printing on LCD
    callDriver(DRV_LCD,LCD_LINE,1);
    callDriver(DRV_LCD,LCD_INTEGER,resp);
    return SUCCESS;
}
```



My board's programming!

DEFCON

“Don't Reinvent The Wheel, Unless You Plan on Learning More About Wheels”
Jeff Atwood