Some Guidelines for Developing Real-Time Applications

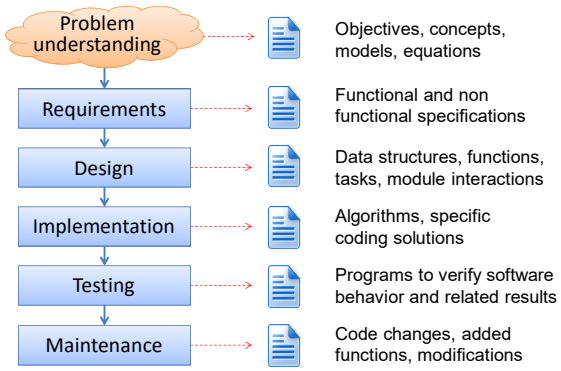
A practical view of Software Engineering

Main goals of SW development

- Contain software complexity by a clean organization of source code.
- Implement software code to be easily readable and easily understandable.
- Guarantee the desired performance in all anticipated worst-case scenarios.
- Simplify maintenance: think of future extensions and develop your code to be easily modified.

2

Software development process



3

Problem understanding

Given a system and a related problem, before designing any software it is very crucial to understand:

- the physics of the system
- the relation between variables
- the user needs
- the control objectives

As a good example, take the train braking system considered at the beginning of the course.

4

Software Design

There are two basic approaches to design:

Top Down

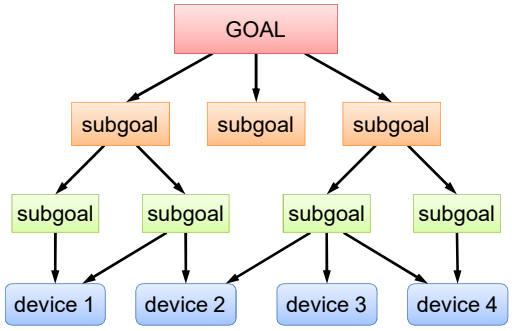
- Start from the goal and decompose it into simpler sub-goals.
- Recursively decompose sub-goals until you identify low-level module that can be mapped to physical devices.

Bottom Up

- Start from I/O devices and define low-level modules that abstract their behavior.
- Rise the level of abstraction defining higher-level modules with more complex behaviors, until you reach the goal.

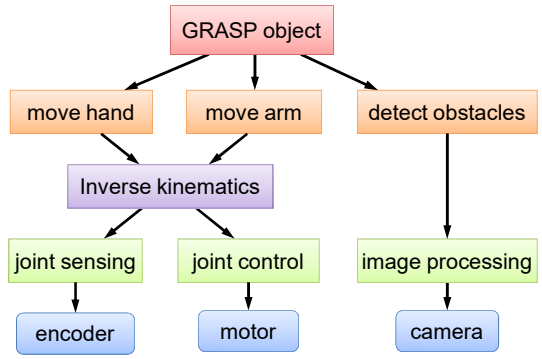
5

Top-Down Approach



6

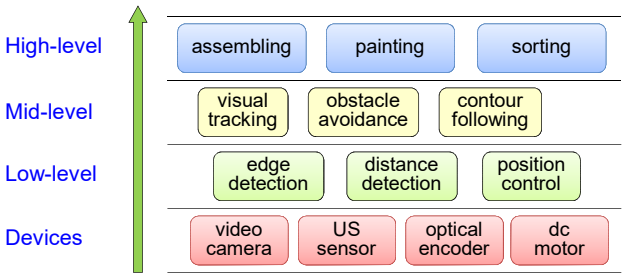
Top-Down Approach



7

Bottom-Up Approach

Develop new layers with higher level of abstractions:



8

Pro and cons

- **Risk of the Top-Down approach**:

  High-level reasoning hides practical problems, so we may design a system that is too abstract and difficult to be implemented.

- **Risk of the Bottom-Up approach**:

  We could get stuck at the low level making things working and never reach the goal.

A better method is to consider both approaches and meet somewhere in between.

9

Software Design

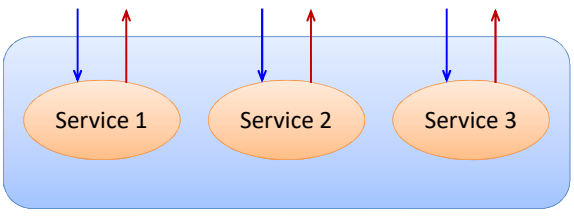
To apply both approaches simultaneously, we need to see the system at 3 different levels of abstractions:

- **User level** (requests and services)
- **Functional level** (modules and functionality)
- **Tasks level** (concurrent tasks and resources)

10

User level

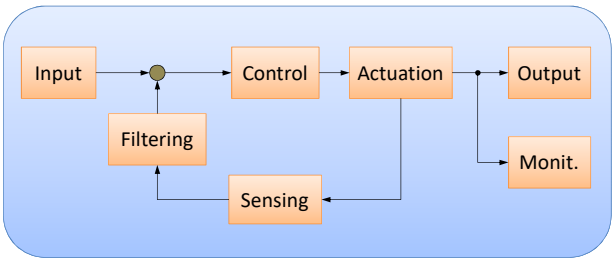
A system is seen as a set of services, each providing a response to a given user request.



11

Functional level

A system is seen as a set of interacting modules, with a precise interface and functionality.



12

Module Specification

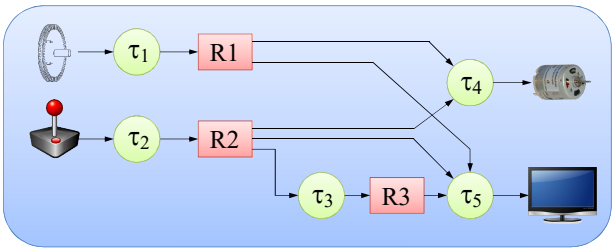
For each module is essential to precisely define:

- **Functionality**
  - describe what has to be done
- **Interface**
  - identify inputs and outputs
  - describe the interactions with the other modules
  - Explicitly state the assumptions: (units, ranges, etc.)
- **Performance requirements**
  - timing constraints (deadline, jitter, throughput)
  - energy consumption
  - fault tolerance issues

13

Task level

Once all modules are defined, you have to map them into a set of tasks interacting with shared resources. Note that a module can be split into more tasks or more modules can be merged in a single task.



14

Implementation Stage

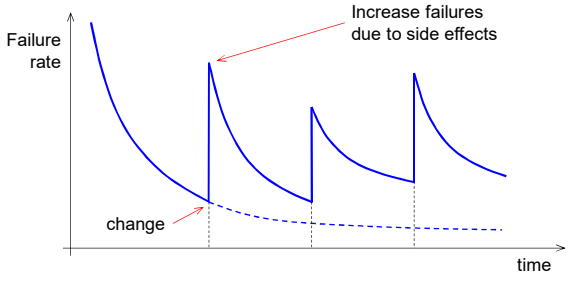
Even when to reach the implementation stage, there are many things to decide before writing any code:

- **User interface**
  - Layout of the graphical user interface (GUI)
  - Commands interpreter
- **Global Constants** (divide them into categories)
- **Global data structures** (Shared Resources)
- **Functions** (how many, what type, which parameters)
- **Tasks** (how many, what type, which constraints)

15

Things to keep in mind

Any program, no matter how well is developed, contains a lot of bugs.



16

How to reduce bugs?

- Put a lot of effort in the initial design phase.
- Write the code according to the given style rules.
- Heavily test your systems to catch existing bugs.
- Since not all bugs can be detected by testing, plan to
  - manage faults and exceptions;
  - use assertive checks to catch inconsistencies.

17

Assertive checks

- When writing a program, it is always good idea to insert **consistency checks** at strategic places for detecting violations of basic assumptions.
- They are very useful to discover various bugs.

Examples

- finishing time <= absolute deadline
- actual execution time <= $WCET_i$
- `((x >= XMIN) && (x <= XMAX))`
- (distance(a,b) >= 0)
- index `i` of `array[N]`: `((i >= 0) && (i < N))`

18

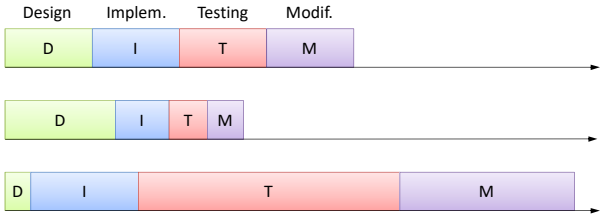
Other tips

- Do not use pointers, unless really necessary
  - bugs are very difficult to catch, since a faulty pointer can cause errors in different parts of the program
- Do not use dynamic memory allocation
  - difficult to check memory consumption and identify bugs;
  - the application can slow down in an uncontrolled manner
- Incremental development
  - Compile very often and check correctness every time you add a few instructions.
- Keep track of previous working versions
  - If you cannot fix a new version, you can always go back and restart from a previous working version.

How much time do I need?

Well, it depends on your skill, but …

Design time in not wasted, but invested in the future

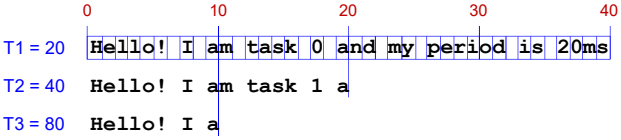


A simple example

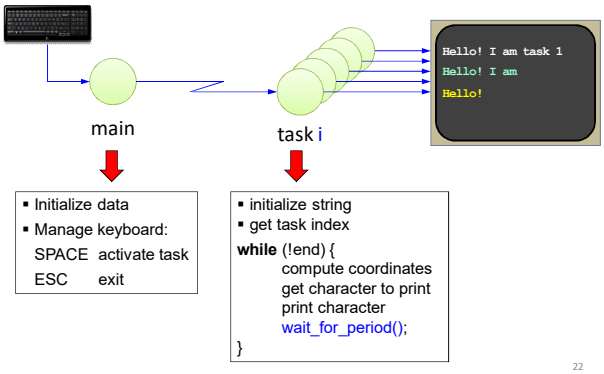
Let us write a simple Hello World application consisting of a set of periodic concurrent tasks.

Each task must simply write a string on the screen printing one character every period.

For example



Application structure



Header files

```
//--------------------------------------------------------
// HELLO.C:    SIMPLE DEMO WHERE EACH PERIODIC TASK
//             DISPLAYS A CHARACTER AT EVERY PERIOD
//--------------------------------------------------------

#include <stdlib.h>        // include standard lib first
#include <stdio.h>
#include <pthread.h>
#include <sched.h>
#include <allegro.h>
#include "ptask.h"         // a lib for periodic tasks
```


Global data

```
//--------------------------------------------------------
// GLOBAL CONSTANTS
//--------------------------------------------------------
#define  XWIN    640        // window x resolution
#define  YWIN    480        // window y resolution
#define  XBASE   40         // X start for the message
#define  YBASE   50         // Y level for the first task
#define  YINC    30         // Y increment for the other tasks
#define  BKG     0          // background color
//--------------------------------------------------------
#define  MAXT    10         // max number of tasks
#define  LEN     80         // max message length
#define  PER     30         // base period
#define  PINC    20         // period increment
//--------------------------------------------------------
// GLOBAL VARIABLES
//--------------------------------------------------------
int      end = 0;           // end flag
char     mes[MAXT][LEN+1];  // buf for MAXT mes of length LEN
```

Init function

```
void    init(void)
{
char    s[LEN];

        allegro_init();
        set_gfx_mode(GFX_AUTODETECT_WINDOWED, XWIN, YWIN, 0, 0);
        clear_to_color(screen, BKG);
        install_keyboard();
        ptask_init(SCHED_FIFO);

        sprintf(s, "Press SPACE to create a task");
        textout_ex(screen, font, s, 10, 10, 14, BKG);
}
```

25

Main task

```
int   main(void)
{
int   i;
char  scan;
      init();
      do {
          scan = 0;
          if (keypressed()) scan = readkey() >> 8;
          if (scan == KEY_SPACE && i < MAXT) {
              task_create(i, hello, PER+i*PINC, PER+i*PINC, 50, ACT);
              i++;
          }
      } while (scan != KEY_ESC);
      end = 1;
      for (i=0; i<=MAXT; i++) wait_for_task_end(i);
      allegro_exit();
      return 0;
}
```

26

Hello task

```
void*  hello(void* arg)
{
int    i, k = 0; // task and character index
int    x, y;
char   buf[2];   // temp buffer for 1 character string

       i = task_argument(arg);
       sprintf(mes[i], "I'M TASK %d, T = %d", i, task_period(i));
       while (!end) {
              x = XBASE + k*8;
              y = YBASE + i*YINC;
              sprintf(buf, "%c", mes[i][k]);
              textout_ex(screen, font, buf, x, y, 2+i, BKG);
              k = k + 1;
              if (mes[i][k] == '\0') {
                  k = 0;
                  textout_ex(screen, font, mes[i], XBASE, y, BKG, BKG);
              }
              wait_for_period(i);
       }
}
```

27