5. Prototyping and Validation

DR. B. I. MORSHED

ASSOCIATE PROFESSOR
COMPUTER SCIENCE
TEXAS TECH UNIVERSITY

Embedded Systems/Cyber Physical Systems

CS 4380 / CS 5331

Outline

Prototyping
- IDE
- Debuggers
- Simulators
- Testing

Functional Verification
- Considerations for Control Systems

Timing Verification
- Considerations for Scheduling

Formal Verification

Validation

Goal of Tests

1. **Production test**
2. Is there any way of using test patterns for production test already during the design?
3. **Test for faults after delivery to customer**

Why is testing of ES difficult?

- Embedded/cyber-physical systems integrated into a physical environment may be safety-critical. As a result, expectations for the product quality are higher than for non-safety critical systems.
- Testing of timing-critical systems has to validate the correct timing behavior. This means that just testing the functional behavior is not sufficient.
- Testing embedded/cyber-physical systems in their real environment may be dangerous.

Scope of Testing

Testing includes

- the application of test patterns to the inputs of the device under test (**DUT**)
- the observation of the results.

More precisely, testing requires the following steps:

1. test pattern generation,
2. test pattern application,
3. response observation, and
4. result comparison.

Testing Approaches

Typically, 3 test approaches are used:

1. Simulation (or emulation)
2. Use of a development system
3. Printed circuit board implementation

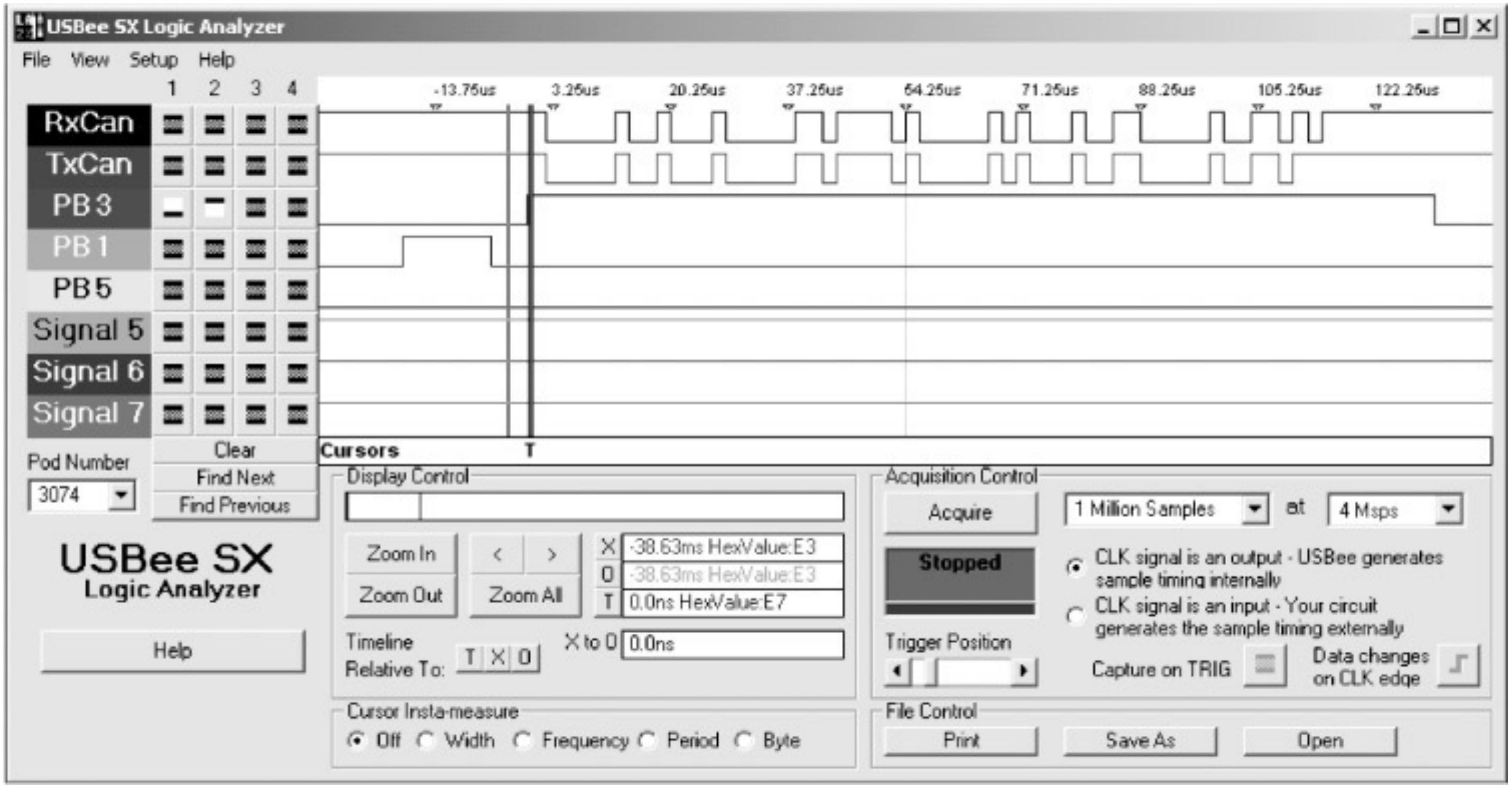
1. Simulator/Emulator:

A **simulator** captures the needed behaviour of the target system

An **emulator** captures all component of the target system

Simulator

Example of Output Waveform



Pod Number: 3074

X: -38.63ms HexValue:E3
0: -38.63ms HexValue:E3
T: 0.0ns HexValue:E7
X to 0: 0.0ns

1 Million Samples at 4 Msps

[✓ CLK signal is an output - USBee generates sample timing internally]
CLK signal is an input - Your circuit generates the sample timing externally

Cursor Insta-measure: [✓ Off] Width Frequency Period Byte

Validating functional behavior by simulation

Various levels of abstractions used for simulations:

- High-level of abstraction: fast, but sometimes not accurate
- Lower level of abstraction: slow and typically accurate
- Choosing a level is always a compromise

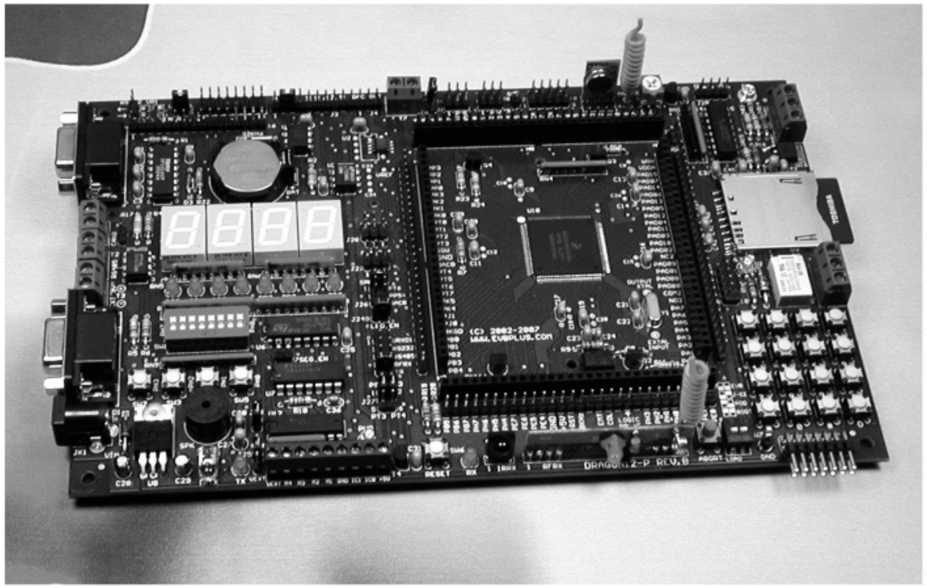
1. Rapid prototyping/Emulation

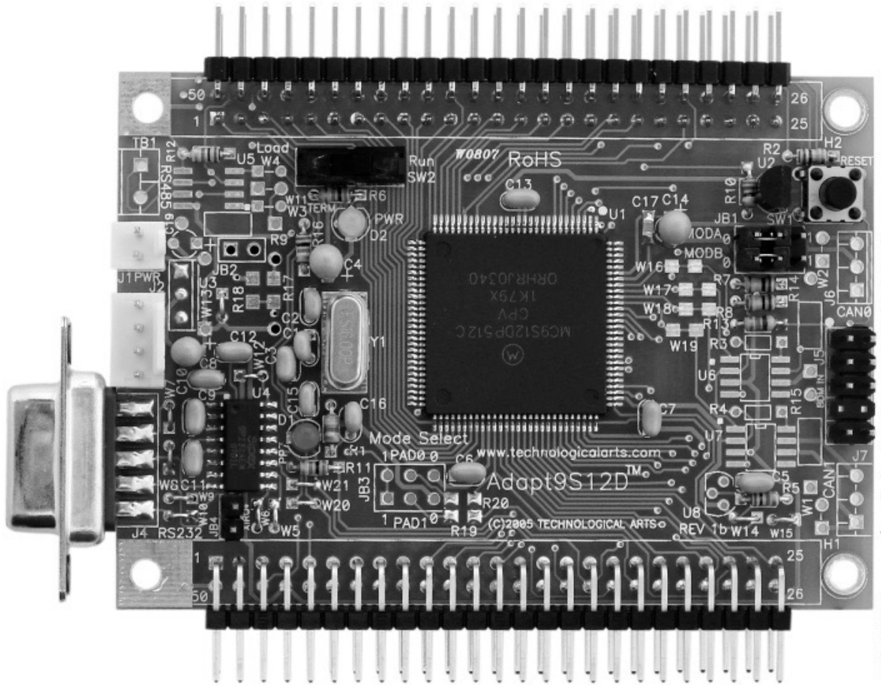
- **Prototype:** Embedded system that can be generated quickly and behaves very similar to the final product.
- May be larger, more power consuming and have other properties that can be accepted in the validation phase
- Can be built, for example, using FPGAs.

Example:
Quickturn Cobalt System (1997), ~0.5M$ for 500kgate entry level system

Source & ©: http://www. eedesign. com/editorial/1997/toolsandtech9703.html

2. Development Boards

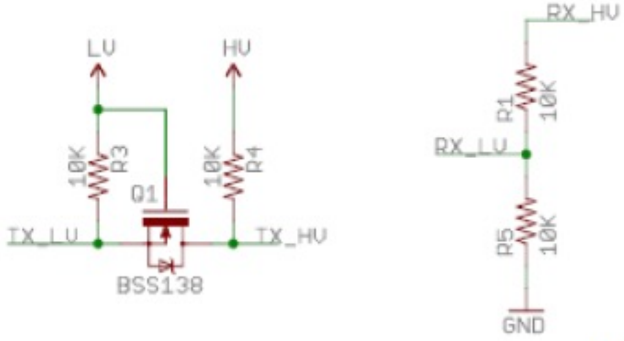

Dragon12-plus board from Wytec

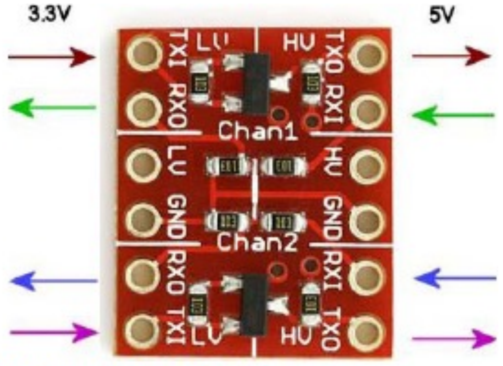

A 9S12DP512 system from Technological Arts (#AD9S12DP512MO).

Daughter boards

Plug-and-play boards

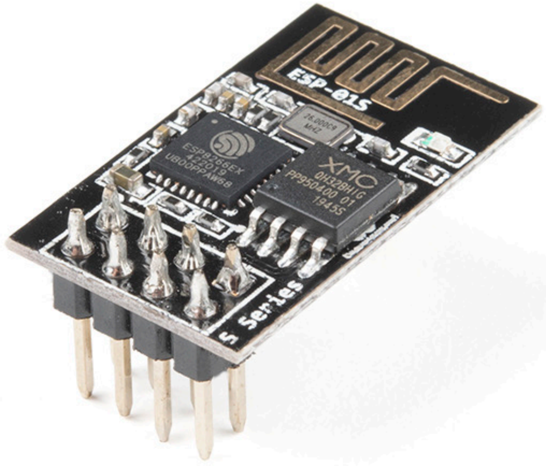
Example: Voltage level interface (5 V <-> 3.3 V)



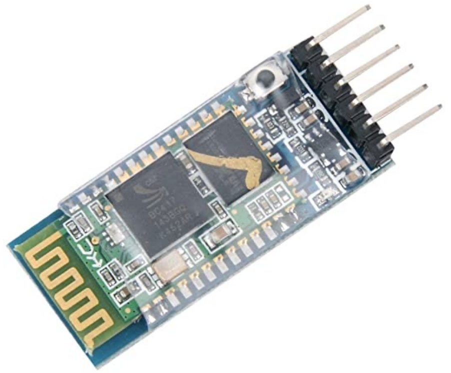


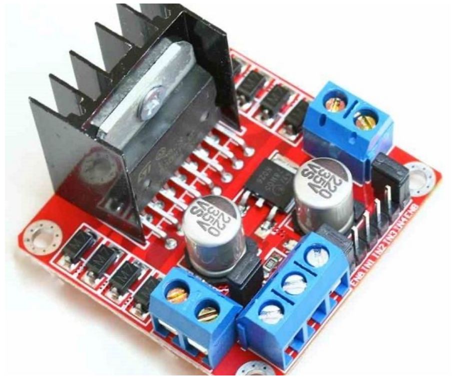
Sparkfun BOB-08745

WiFi Module for Arduino



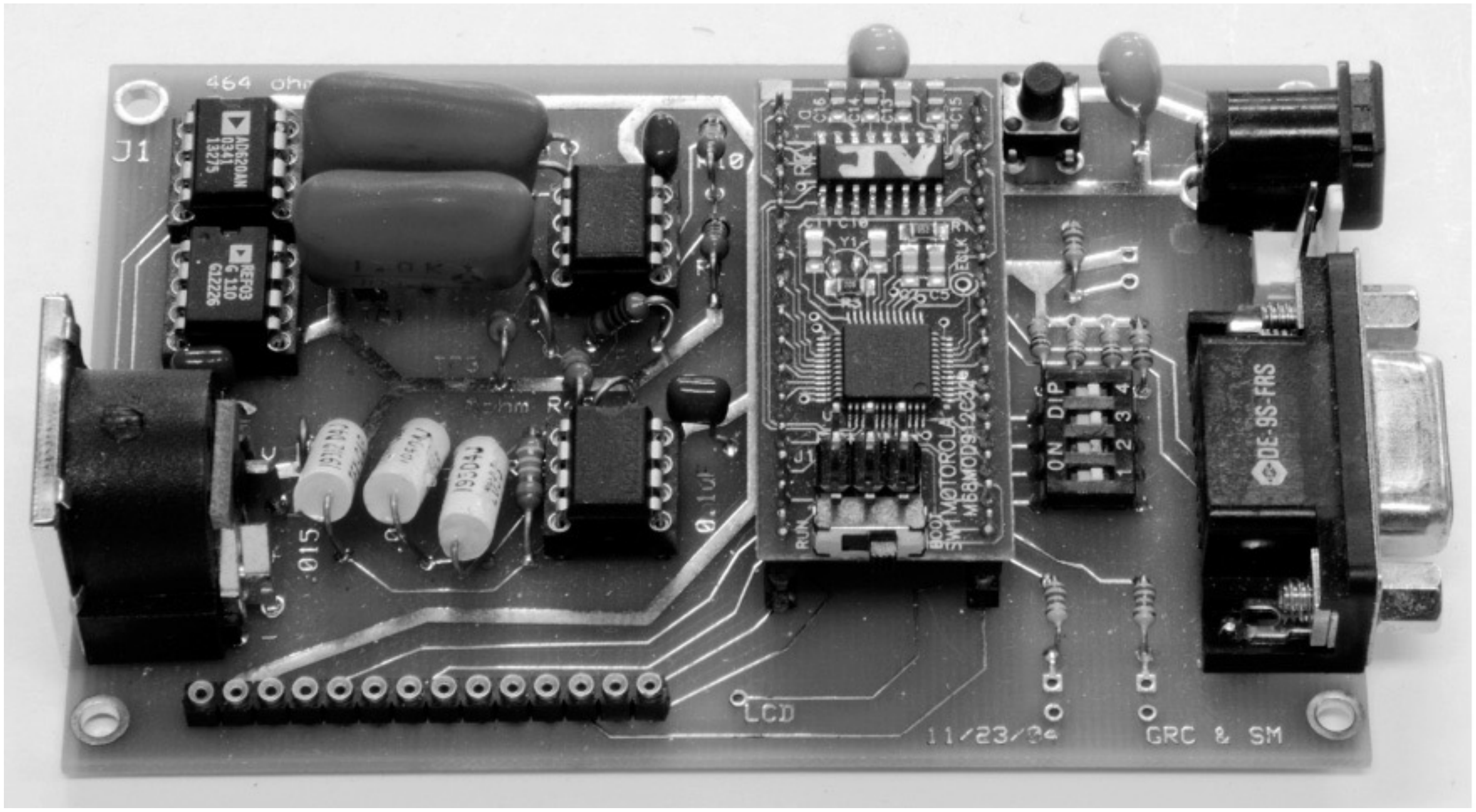
Bluetooth Module for Arduino





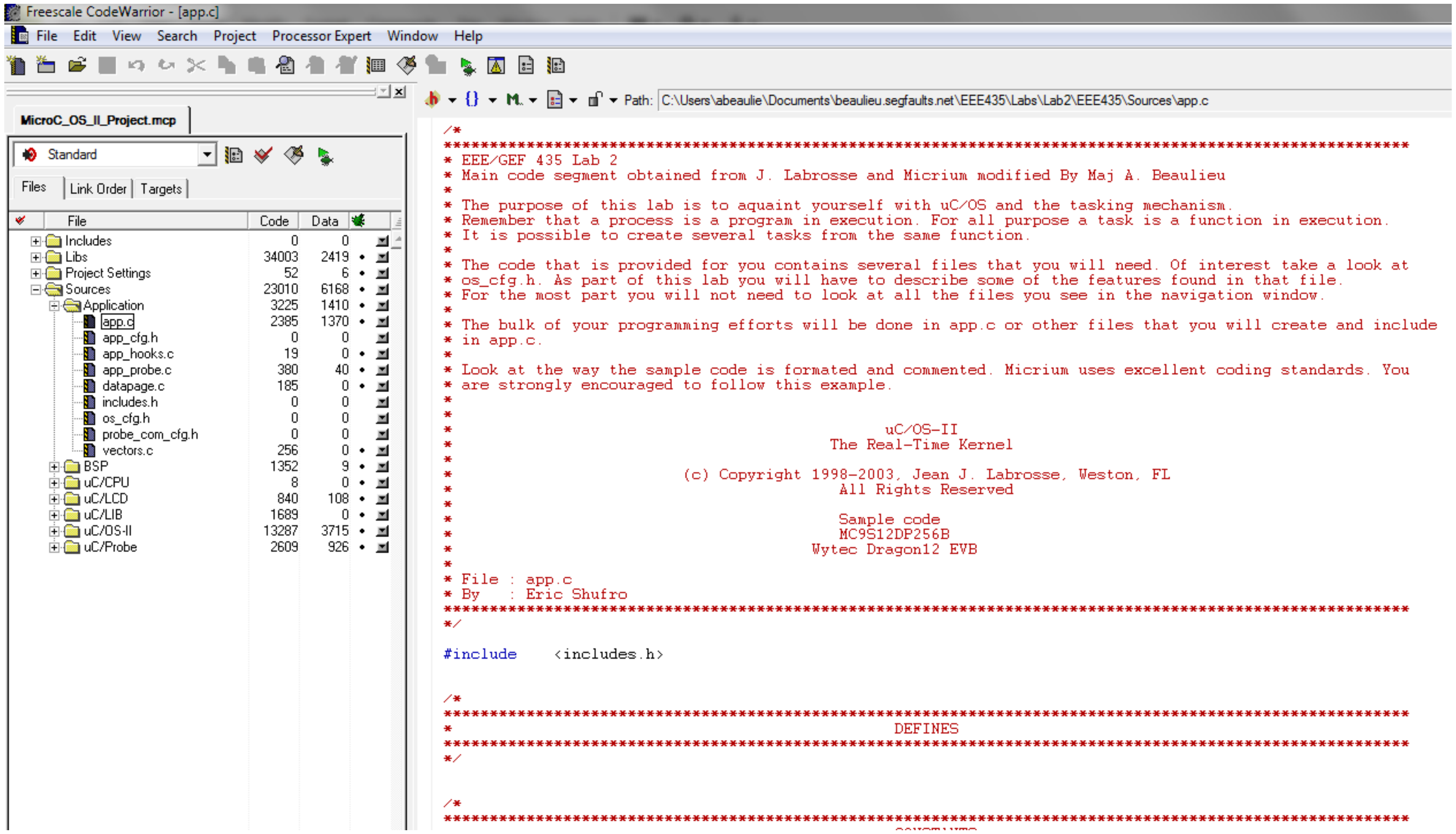
L298N DC motor drive module for Arduino

3. Prototyping with Customized Board (with PCB)



ourtesy of Jonathan Valvano

Integrated Development Environment (IDE)


Freescale CodeWarrior - [app.c]
File Edit View Search Project Processor Expert Window Help
MicroC_OS_II_Project.mcp
Standard
Files | Link Order | Targets

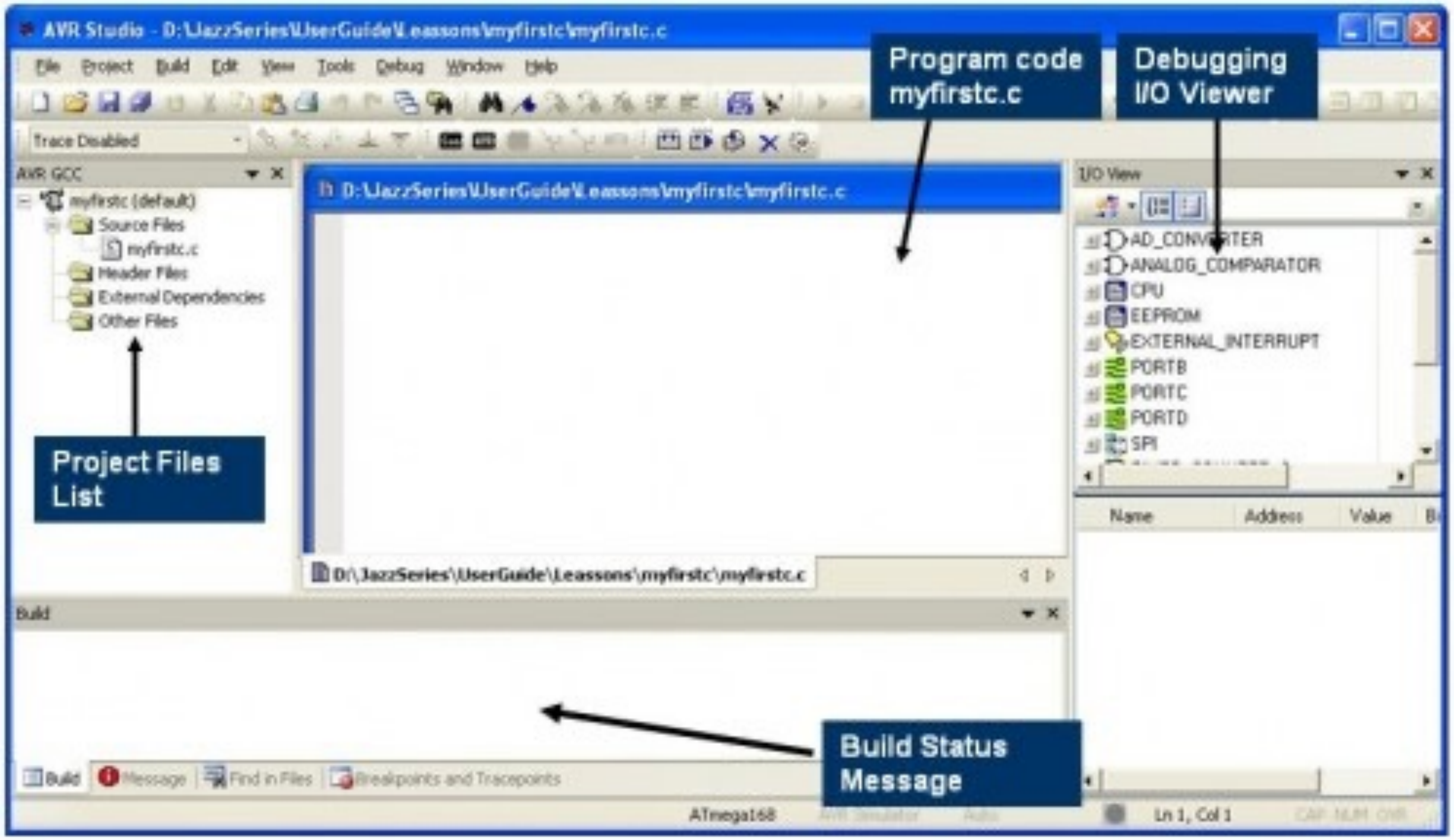
Path: C:\Users\abeaulie\Documents\beaulieu.segfaults.net\EEE435\Labs\Lab2\EEE435\Sources\app.c

File	Code	Data
Includes	0	0
Libs	34003	2419
Project Settings	52	6
Sources	23010	6168
Application	3225	1410
app.c	2385	1370
app_cfg.h	0	0
app_hooks.c	19	0
app_probe.c	380	40
datapage.c	185	0
includes.h	0	0
os_cfg.h	0	0
probe_com_cfg.h	0	0
vectors.c	256	0
BSP	1352	9
uC/CPU	8	0
uC/LCD	840	108
uC/LIB	1689	0
uC/OS-II	13287	3715
uC/Probe	2609	926

```
/*
*********************************************************************************************************
* EEE/GEF 435 Lab 2
* Main code segment obtained from J. Labrosse and Micrium modified By Maj A. Beaulieu
*
* The purpose of this lab is to aquaint yourself with uC/OS and the tasking mechanism.
* Remember that a process is a program in execution. For all purpose a task is a function in execution.
* It is possible to create several tasks from the same function.
*
* The code that is provided for you contains several files that you will need. Of interest take a look at
* os_cfg.h. As part of this lab you will have to describe some of the features found in that file.
* For the most part you will not need to look at all the files you see in the navigation window.
*
* The bulk of your programming efforts will be done in app.c or other files that you will create and include
* in app.c.
*
* Look at the way the sample code is formated and commented. Micrium uses excellent coding standards. You
* are strongly encouraged to follow this example.
*
*
*                                             uC/OS-II
*                                        The Real-Time Kernel
*
*                         (c) Copyright 1998-2003, Jean J. Labrosse, Weston, FL
*                                         All Rights Reserved
*
*                                            Sample code
*                                           MC9S12DP256B
*                                        Wytec Dragon12 EVB
*
* File : app.c
* By   : Eric Shufro
*********************************************************************************************************
*/

#include    <includes.h>


/*
*********************************************************************************************************
*                                              DEFINES
*********************************************************************************************************
*/


/*
*********************************************************************************************************
```

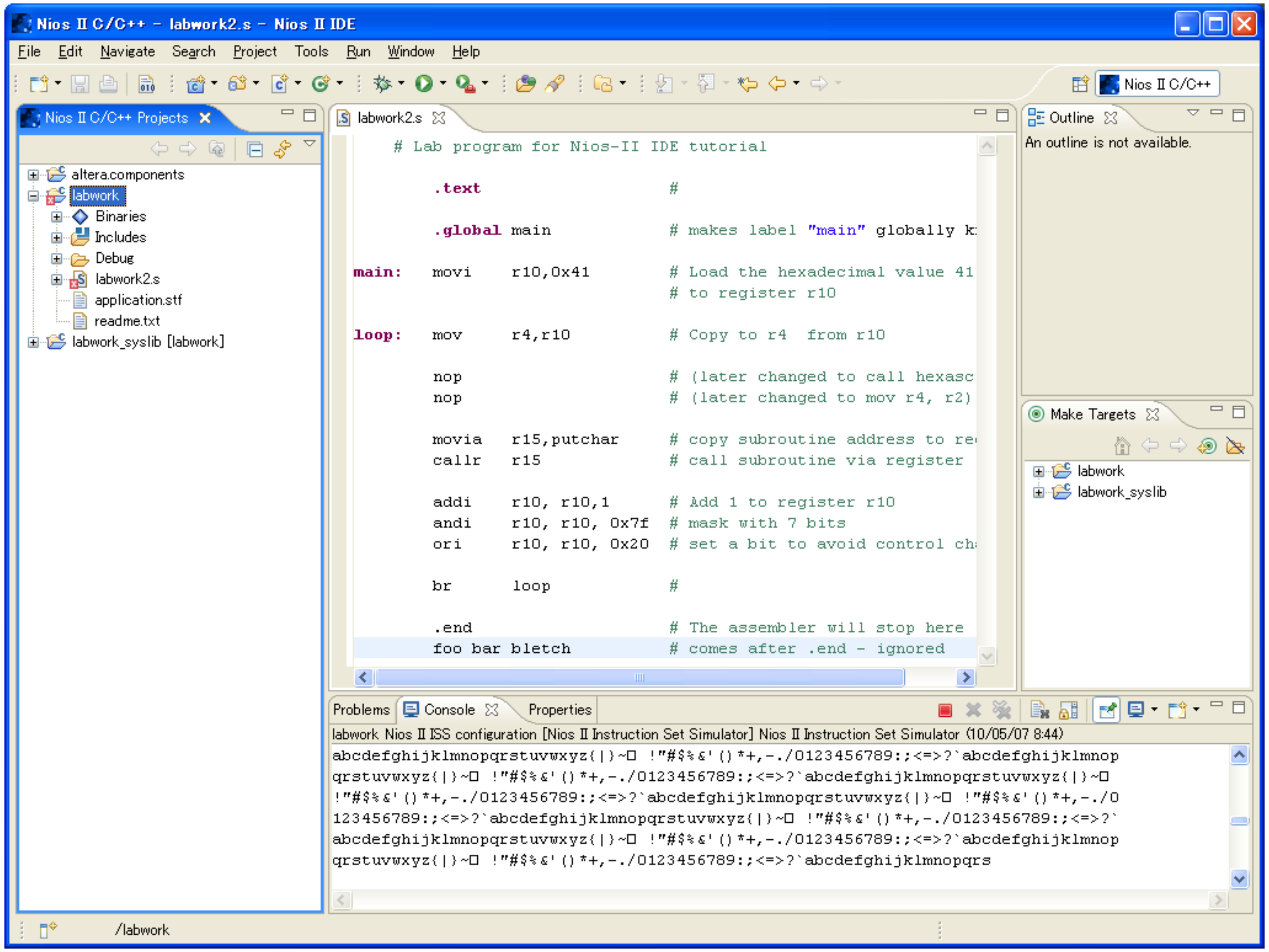
Code warrior (Motorola)

IDE for AVR Studio



AVR Studio (Atmel)

IDE for Altera FPGA



```
    # Lab program for Nios-II IDE tutorial

        .text                   #

        .global main            # makes label "main" globally k

main:   movi    r10,0x41        # Load the hexadecimal value 41
                                # to register r10

loop:   mov     r4,r10          # Copy to r4  from r10

        nop                     # (later changed to call hexasc
        nop                     # (later changed to mov r4, r2)

        movia   r15,putchar     # copy subroutine address to re
        callr   r15             # call subroutine via register

        addi    r10, r10,1      # Add 1 to register r10
        andi    r10, r10, 0x7f  # mask with 7 bits
        ori     r10, r10, 0x20  # set a bit to avoid control ch

        br      loop            #

        .end                    # The assembler will stop here
        foo bar bletch          # comes after .end - ignored
```

labwork Nios II ISS configuration [Nios II Instruction Set Simulator] Nios II Instruction Set Simulator (10/05/07 8:44)

```
abcdefghijklmnopqrstuvwxyz{|}~□ !"#$%&'()*+,-./0123456789:;<=>?`abcdefghijklmnop
qrstuvwxyz{|}~□ !"#$%&'()*+,-./0123456789:;<=>?`abcdefghijklmnopqrstuvwxyz{|}~□
!"#$%&'()*+,-./0123456789:;<=>?`abcdefghijklmnopqrstuvwxyz{|}~□ !"#$%&'()*+,-./0
123456789:;<=>?`abcdefghijklmnopqrstuvwxyz{|}~□ !"#$%&'()*+,-./0123456789:;<=>?`
abcdefghijklmnopqrstuvwxyz{|}~□ !"#$%&'()*+,-./0123456789:;<=>?`abcdefghijklmnop
qrstuvwxyz{|}~□ !"#$%&'()*+,-./0123456789:;<=>?`abcdefghijklmnopqrs
```

Altera FPGA

Debugging

Software level:

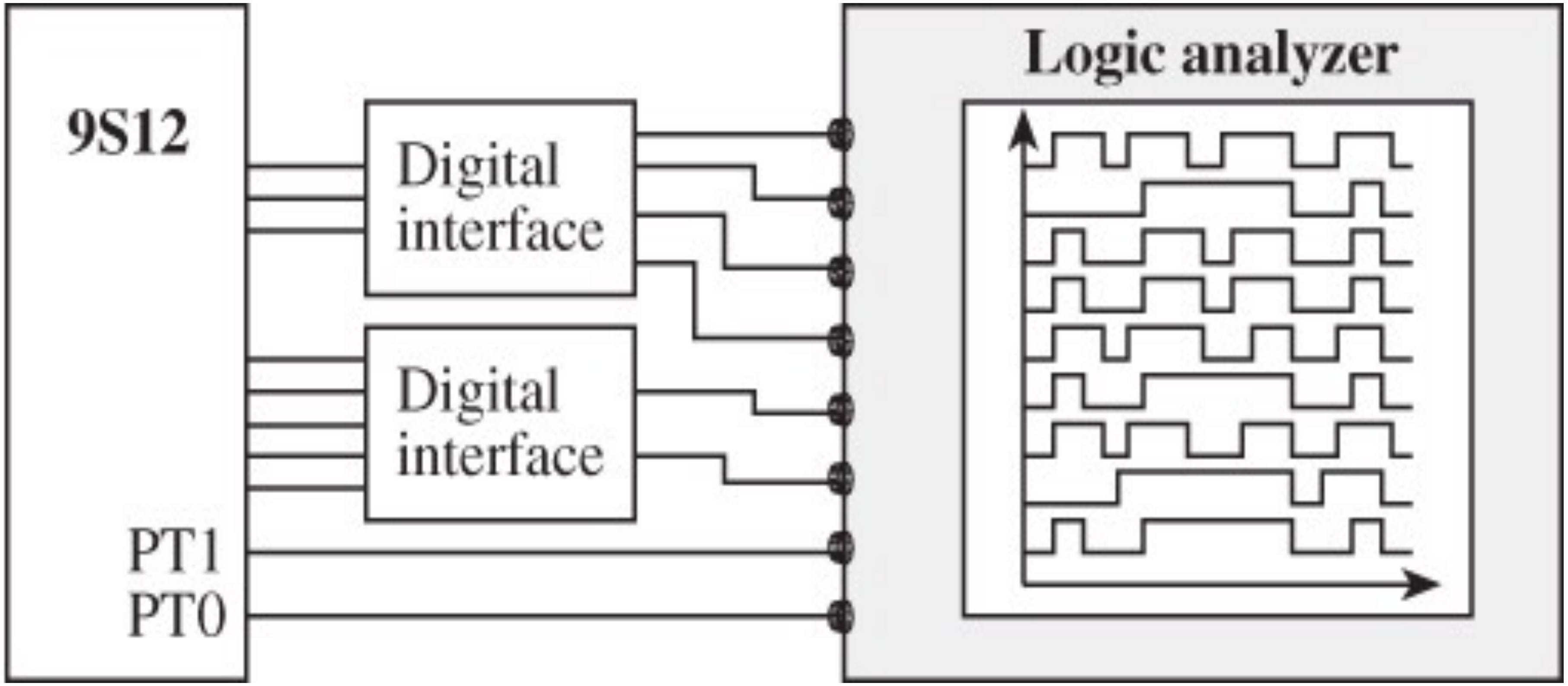
- Simulator
- Emulator
- Monitor
- Profiler

Hardware level:

- Logic analyzer
- Background debug module (BDM)
- In-circuit debugger (ICD)

Logic Analyzer for Debugging

For 1 or 2 channels, use oscilloscope



Digital signals of a bus

MCU Programmer and Debugger

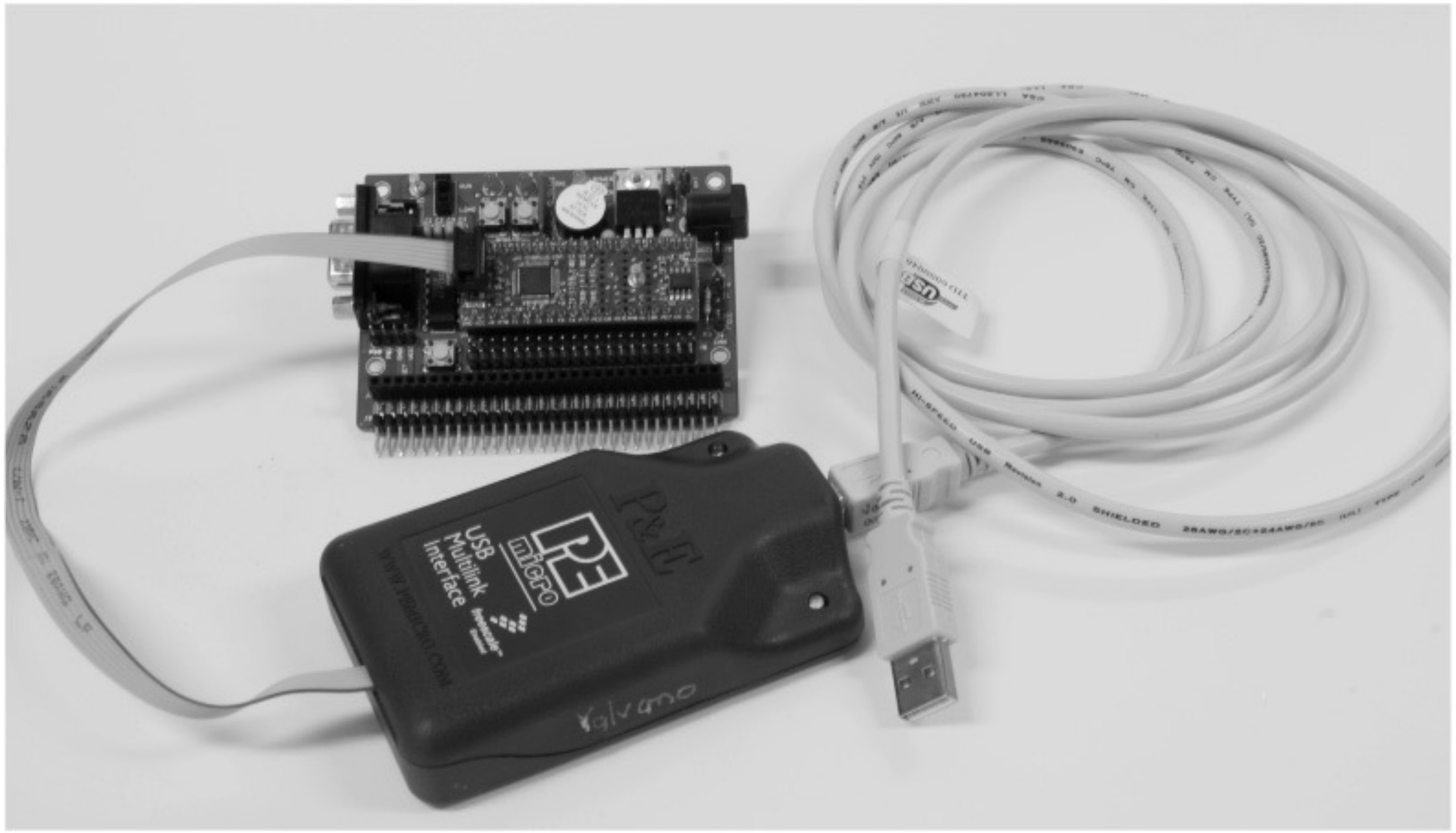


Figure 2.40
P&E Microcomputer Systems Multilink BDM connected to a DragonFly12 (**www.evbplus.com**)

BDM: Background Debug Mode

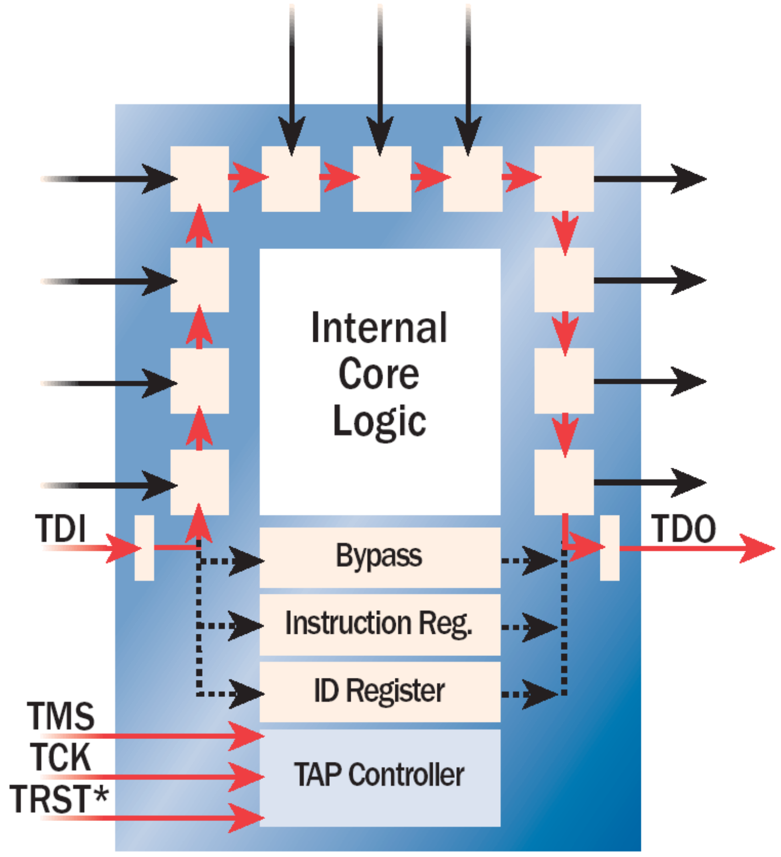
Design for testability

If testing comes in as an afterthought, testing may be very complicated and expensive

- ☞ Try to consider testing already during the design!
- ☞ Design for testability (DFT)
- ☞ Important for finite state machines (FSMs) and, hence, embedded systems.

JTAG (Boundary scan)

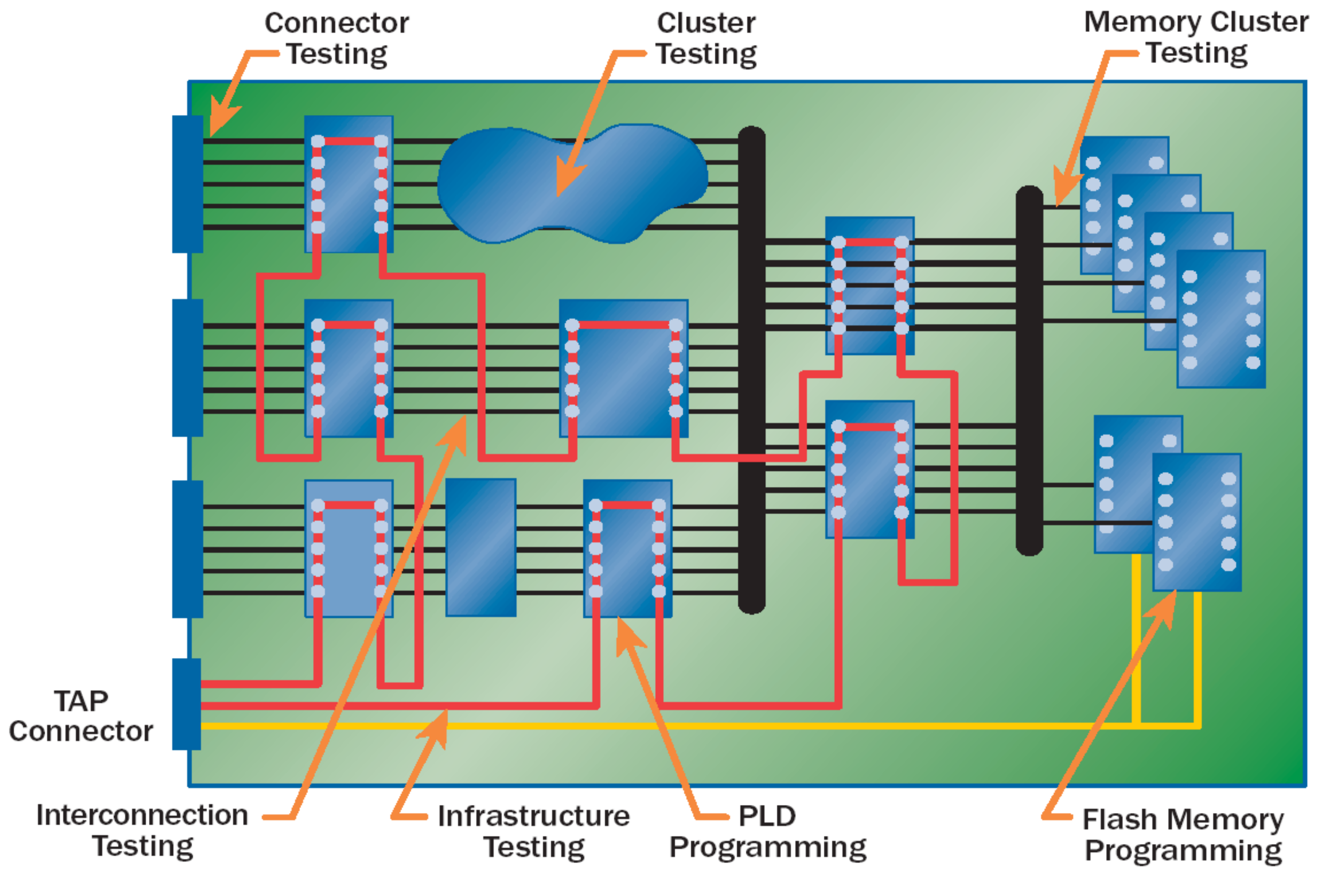
- JTAG defines a 4..5-wire serial interface to access complex ICs .. Any compatible IC contains shift registers & FSM to execute the JTAG functions.
- **TDI:** test data in; stored in instruction register or in one of the data registers.
  TDO: test data out
  TCK: clock
  TMS: controls the state of the test access port (TAP).
  Optional **TRST*** is reset signal.



IEEE 1149.1 Device Architecture
Source: http://www.jtag.com/brochure.php

JTAG (Boundary scan) on PCB

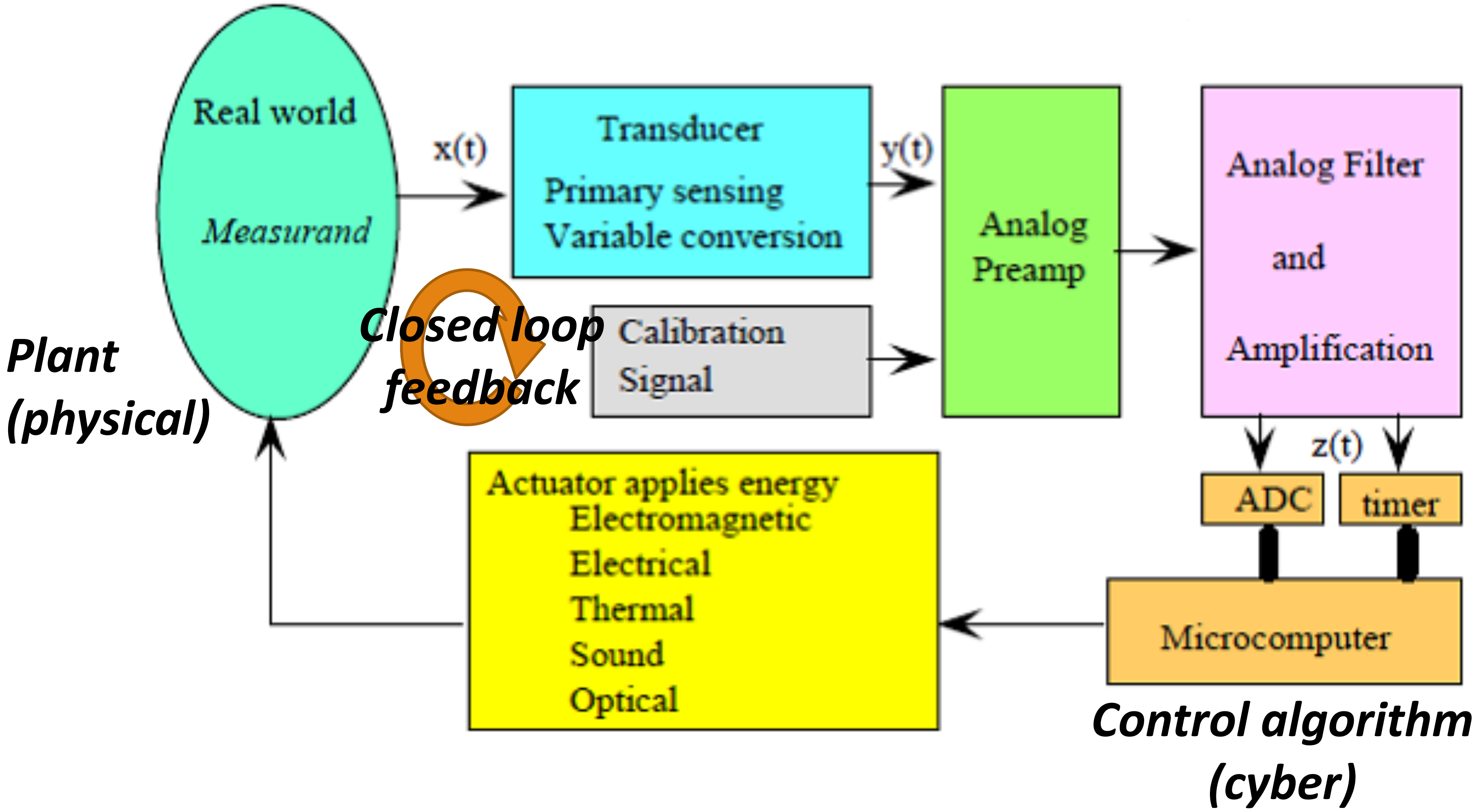
- Defines method for setting up a scan chain on a PCB



Applications of Boundary-Scan

Source: http://www.jtag.com/brochure.php

CPS Example

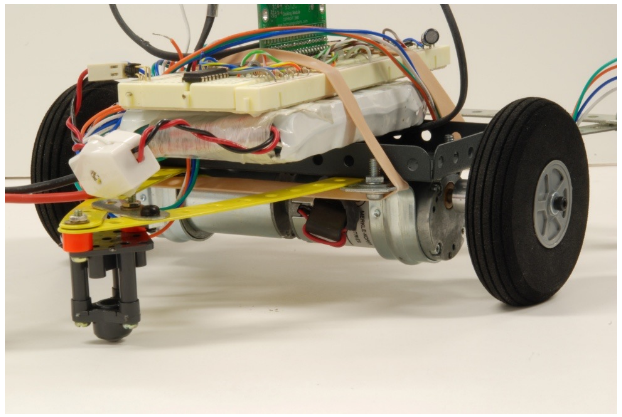

Real world
Measurand
x(t)
Transducer
Primary sensing
Variable conversion
y(t)
Analog Preamp
Analog Filter
and
Amplification
Calibration Signal
z(t)
ADC
timer
Microcomputer
Actuator applies energy
Electromagnetic
Electrical
Thermal
Sound
Optical
Closed loop feedback
Plant (physical)
Control algorithm (cyber)

Testing of CPS

Microcomputers are widely employed in CPS:

- automotive ABS, ignition and fuel systems
- household appliances
- smart weapons
- industrial robots
- pacemakers

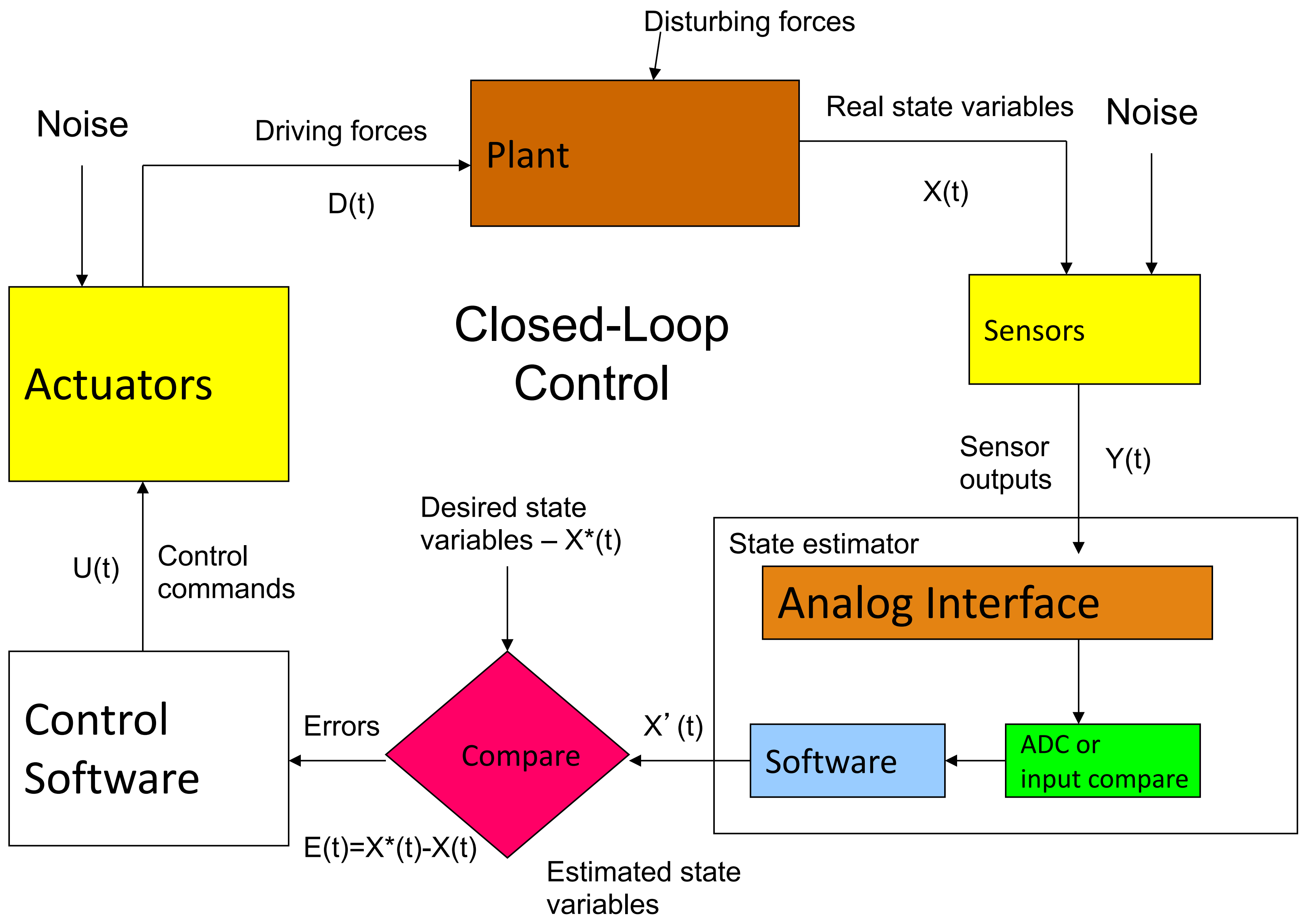
- Strategy
  - *plant* is a system that is intended to controlled
  - collect information concerning the plant – data acquisition system (DAS)
  - compare with desired performance
  - generate outputs to bring plant closer to desired performance



Closed-loop System

Closed-loop control

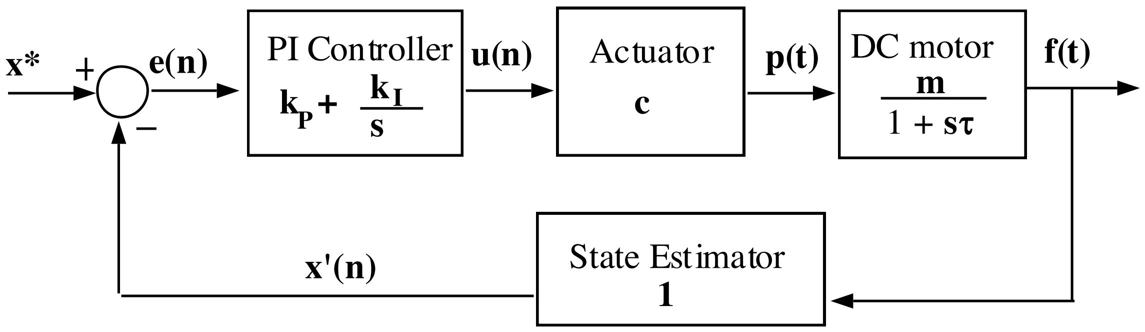
- feedback loop implementation
  - suitable for complex plant
- sensors and state estimator produce representation/estimation of state variables
- these values are compared to desired values
- control software generates control commands based upon the differences between estimated and desired values



Disturbing forces
Noise
Driving forces
Plant
Real state variables
Noise
D(t)
X(t)
Actuators
Closed-Loop Control
Sensors
Sensor outputs
Y(t)
Desired state variables – X*(t)
U(t)
Control commands
State estimator
Analog Interface
Control Software
Errors
Compare
X' (t)
Software
ADC or input compare
E(t)=X*(t)-X(t)
Estimated state variables

Closed Loop Controller Algorithms

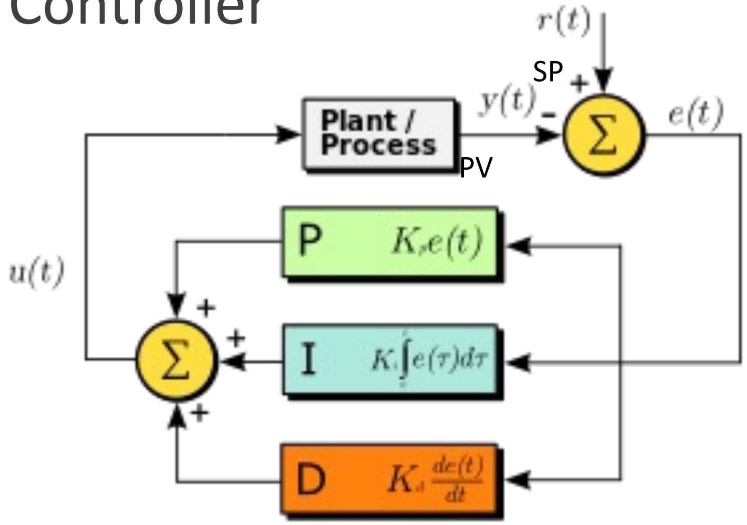
PI: Proportional Integral Controller



PID: Proportional Integral Derivative Controller

$$u(t) = \mathrm{MV}(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{d}{dt} e(t)$$

K_p : Proportional gain, a tuning parameter
K_i : Integral gain, a tuning parameter
K_d : Derivative gain, a tuning parameter
e : Error = *SP - PV*
t : Time or instantaneous time (the present)
τ : Variable of integration; takes on values from time 0 to the present *t*.



Adaptive Intelligent Controllers

Fuzzy logic

Artificial Neural Network (ANN)

Genetic Algorithm (GA)

Neuro-Fuzzy logic

Characteristics:

- Non-linear
- Adaptive
- Can be trained
- Intelligence

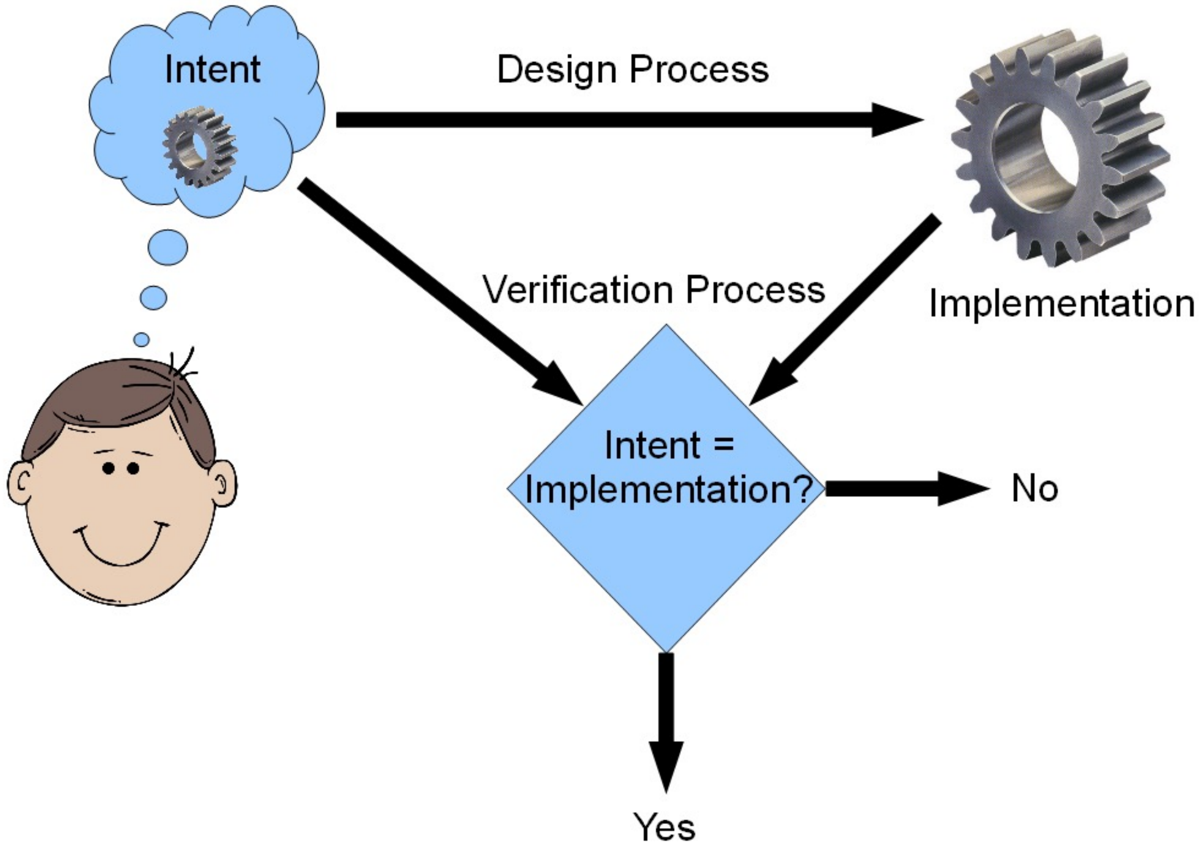
Control Systems Metrics

Performance metrics

- steady-state controller error
  - an average value of the difference between desired and actual performance
- transient response
  - how quickly the system responds to change
- stability
  - system output changes smoothly – without oscillation or unlimited excursions

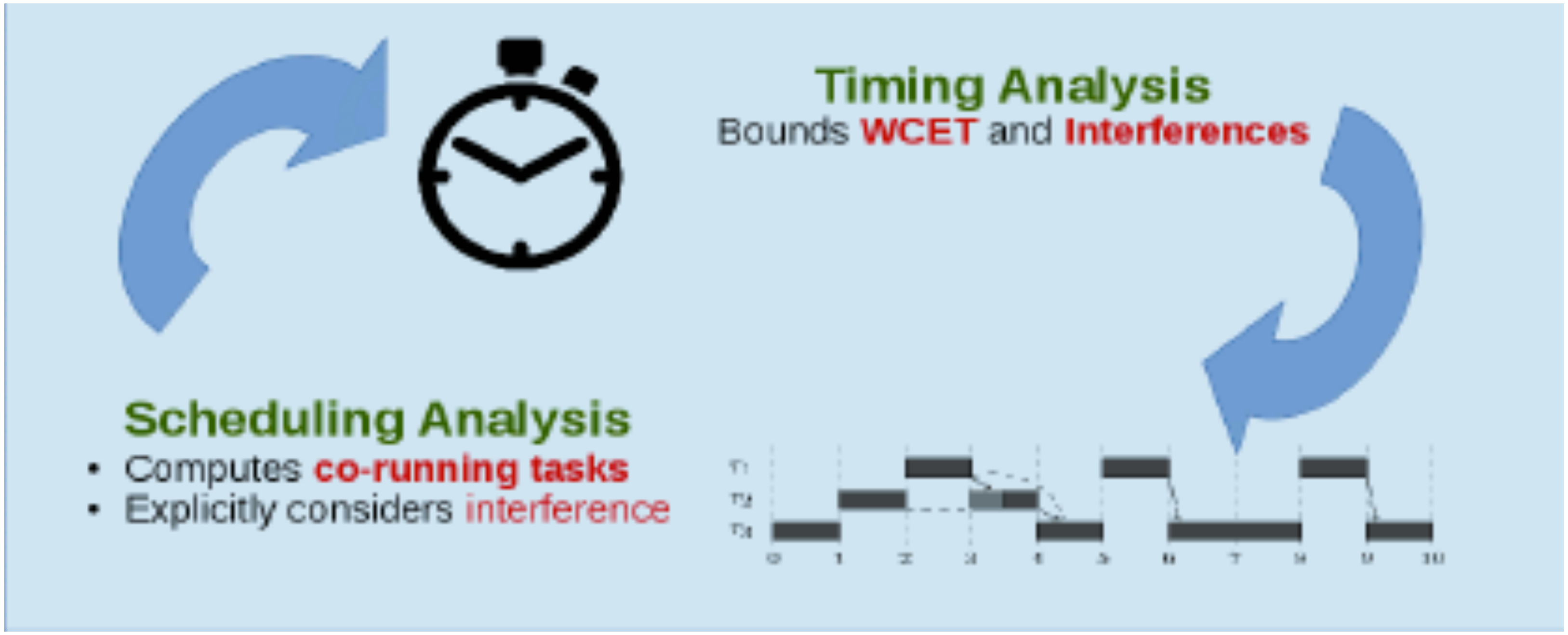
Functional Verification

Functional Verification is defined as the process of verifying that a design meets its specification from a functional perspective.



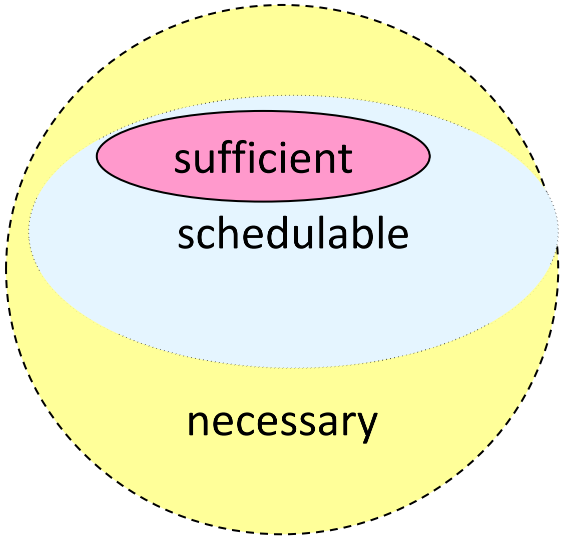
Timing Verification

Timing verification refers to verifying that a system is fast enough to run without errors at the targeted clock rate.


Timing Analysis
Bounds WCET and Interferences
Scheduling Analysis
• Computes co-running tasks
• Explicitly considers interference

Timing Scheduling

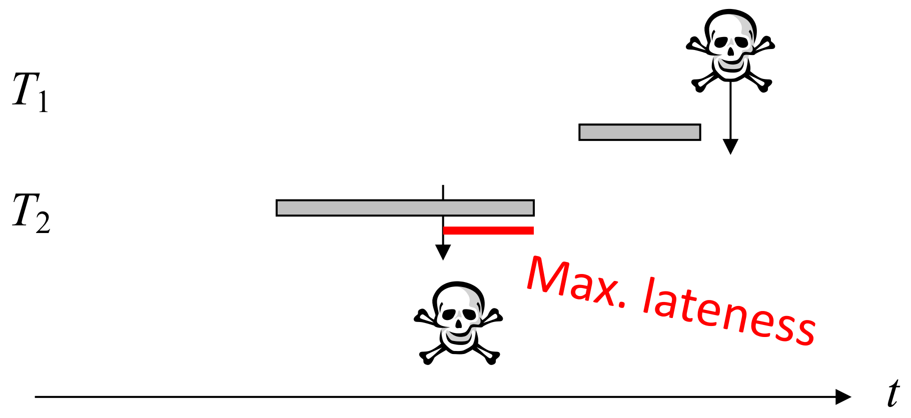
- Set of tasks is **schedulable** under a set of constraints, if a schedule exists for that set of tasks & constraints.
- **Exact tests** are NP-hard in many situations.
- **Sufficient tests**: sufficient conditions for schedule checked. (Hopefully) small probability of not guaranteeing a schedule even though one exists.
- **Necessary tests**: checking necessary conditions. Used to show no schedule exists. There may be cases in which no schedule exists & we cannot prove it.



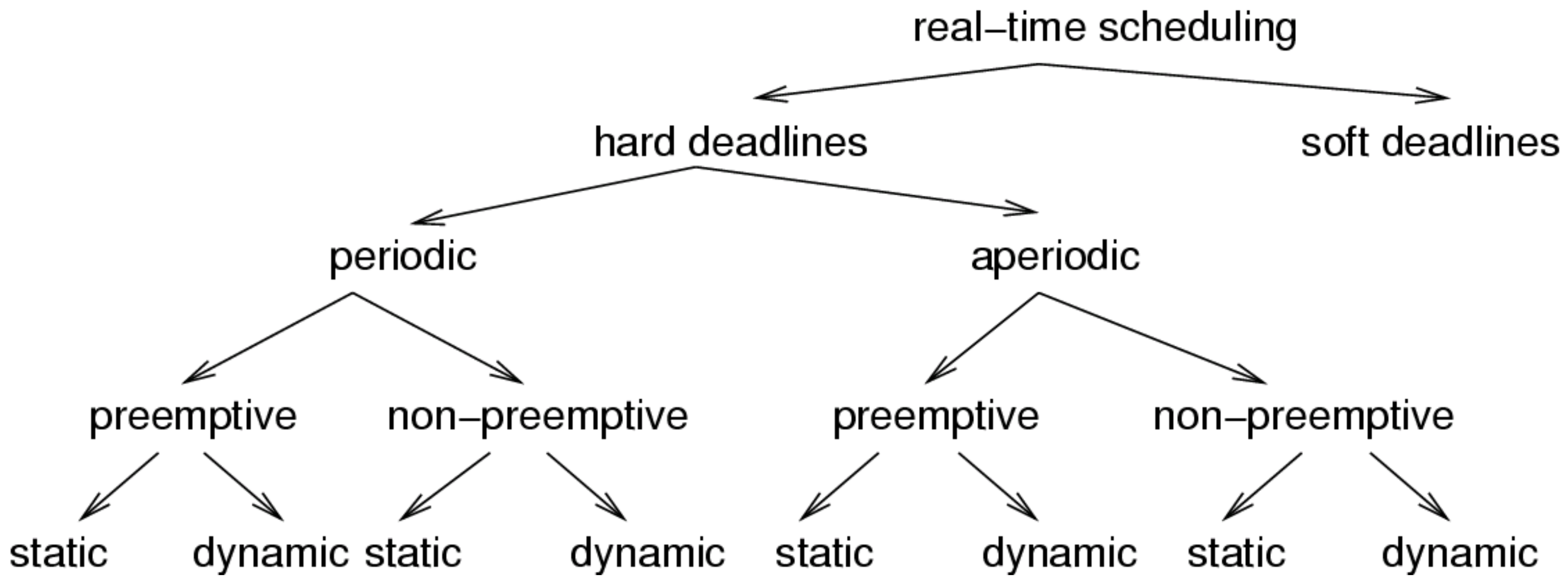
Scheduling Cost functions

- **Cost function:** Different algorithms aim at minimizing different functions.

- **Def.: Maximum lateness =**
  max$_{\textbf{all tasks}}$ (completion time – deadline)
  Is <0 if all tasks complete before deadline.



Scheduling deadlines



- **Def.:** A time-constraint (deadline) is called **hard** if not meeting that constraint could result in a catastrophe [Kopetz, 1997].
- All other time constraints are called **soft**.

Task periodicity

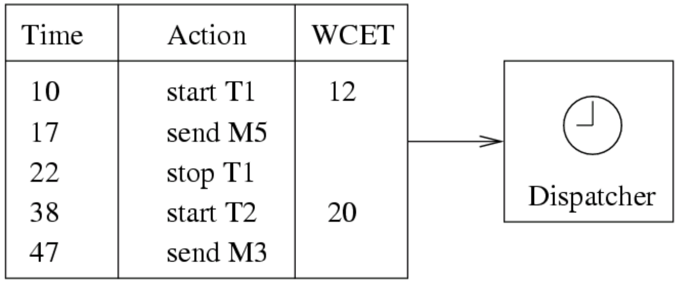
- **Def.:** Tasks which must be executed once every *p* units of time are called **periodic** tasks. *p* is called their period. Each execution of a periodic task is called a **job**.
- All other tasks are called **aperiodic**.
- **Def.:** Tasks requesting the processor at unpredictable times are called **sporadic**, if there is a minimum separation between the times at which they request the processor.
- **Dynamic/online scheduling:**
  Processor allocation decisions (scheduling) at run-time; based on the information about the tasks arrived so far.

Time-triggered systems

In an entirely time-triggered system, the temporal control structure of all tasks is established ***a priori*** *by off-line support-tools. This temporal control structure is encoded in a* ***Task-Descriptor List (TDL)*** *that contains the cyclic schedule for all activities of the node.*

The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz].

- It can be easily checked if timing constraints are met.
  The disadvantage is that the response to sporadic events may be poor.

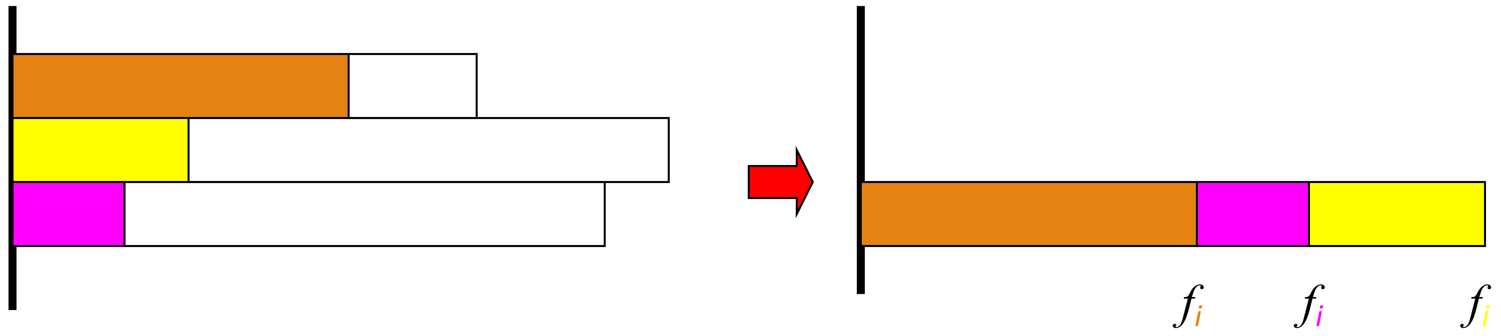


Time	Action	WCET
10	start T1	12
17	send M5	
22	stop T1	
38	start T2	20
47	send M3	

Scheduling approaches

- **Earliest Due Date** (EDD): Execute task with earliest due date (deadline) first.

Uniprocessor with equal arrival times

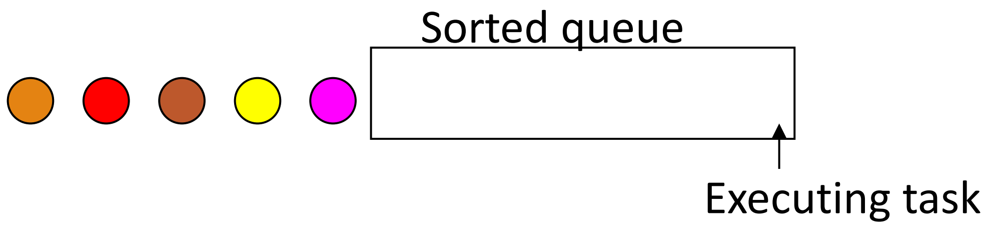


EDD requires all tasks to be sorted by their (absolute) deadlines. Hence, its complexity is $O(n\log(n))$.

Preemption is not useful.

Scheduling approaches

- **Earliest deadline first** (EDF) algorithm:
  - Each time a new ready task arrives:
  - It is inserted into a queue of ready tasks, sorted by their **absolute** deadlines. Task at head of queue is executed.
  - If a newly arrived task is inserted at the head of the queue, the currently executing task is preempted.
- Straightforward approach with sorted lists (full comparison with existing tasks for each arriving task) requires run-time $O(n^2)$; (less with binary search or bucket arrays).



Scheduling approaches

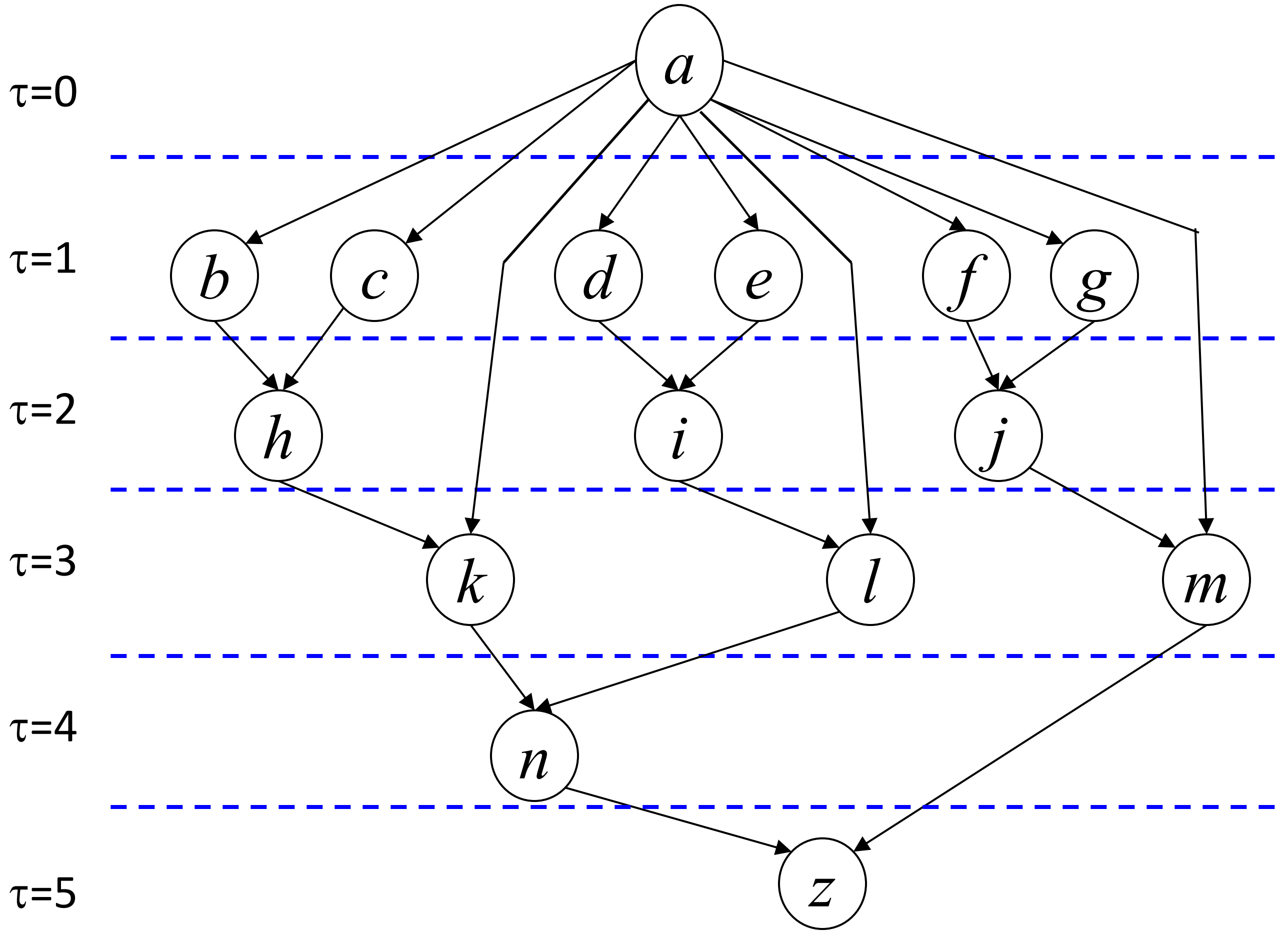
As soon as possible (ASAP) scheduling

ASAP: All tasks are scheduled as early as possible

Loop over (integer) time steps:

- Compute the set of unscheduled tasks for which all predecessors have finished their computation
- Schedule these tasks to start at the current time step.

As soon as possible (ASAP) scheduling example



Formal verification

- Formal verification = formally proving a system correct, using the language of mathematics.
- Formal model required. Obtaining this cannot be automated.
- Model available ☞ try to prove properties.
- Even a formally verified system can fail (e.g. if assumptions are not met).
- Classification by the type of logics.

- **Ideally:** Formally verified tools transforming specifications into implementations (“*correctness by construction*“).
- **In practice:** Non-verified tools and manual design steps ☞ validation of each and every design required Unfortunately has to be done at intermediate steps and not just for the final design ☞ Major effort required.

Validation and Evaluation

Validation is the process of checking whether or not a certain (possibly partial) design is appropriate for its purpose, meets all constraints and will perform as expected (yes/no decision).

Validation with mathematical rigor is called *(formal) verification*.

Evaluation is the process of computing quantitative information of some key characteristics of a certain (possibly partial) design.

Validation Challenges

Dependability and correctness are central concerns in embedded systems design.

A program may have the possibility of deadlock, for example, but nonetheless run correctly for years without the deadlock ever appearing.

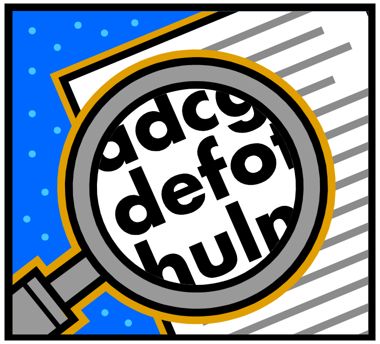
Due to physical aspect of the DUT, not all scenarios are testable, as well as it might be expensive (cost, time).

Programmers have to be very cautious, but reasoning about the programs is sufficiently difficult that programming errors are likely to persist.

Performance Evaluation

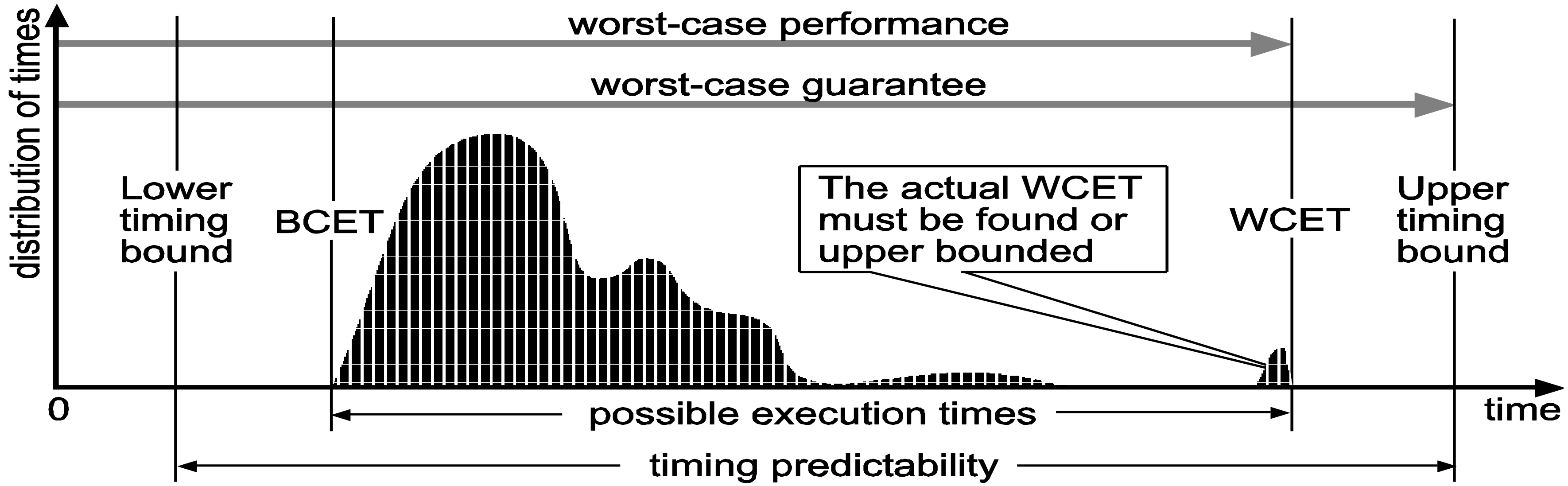
- **Estimated performance values:**
  Difficult to generate sufficiently precise estimates;
  Balance between run-time and precision

- **Accurate performance values:**
  As precise as the input data is.



We need to compute **average** and **worst case** execution times

Worst/best case execution times



WCET$_{EST}$

Requirements on WCET estimates:

- *Safeness:* WCET $\leq$ WCET$_{EST}$!
- *Tightness:* WCET$_{EST}$ – WCET $\rightarrow$ minimal

Chapter Summary

Prototyping
- IDE
- Simulation
- Testing
- Debugging

Functional Verification
- Considerations for Control Systems

Timing Verification
- Considerations for Scheduling

Formal Verification

Validation