

# Design of a Hands-on Course in Networked Control Systems

Josep M. Fuertes, Ricard Villà, Jordi Ayza, Pere Marés, Pau Martí,  
Manel Velasco, José Yépez, Gina Torres and Miquel Perelló

Automatic Control Department, Technical University of Catalonia  
Barcelona, Spain

Research Report: ESAIL-RR-12-01

January 2012

## Abstract

*This report presents a hands-on course in networked control systems (NCS) to be integrated in the education of embedded control systems engineers. The course activities have a strong practical component and most of them are applied exercises to be implemented in a NCS setup. The report contains four parts: a) a report that describes the experimental setup, proposing several activities that can be shaped into a course program according to the needs and diverse background of the targeted audience, b) a tentative program example for master students, c) a user manual to help setting up the hardware and software from a Live CD, and d) a quick guide to start working with the programming environment.*

**Keywords:** Networked control systems, education.

# Contents

<b>1</b>	<b>Research report</b>	<b>3</b>
1.1	Introduction . . . . .	3
1.2	Background on NCS . . . . .	4
1.3	Experimental set-up . . . . .	5
1.3.1	Introduction . . . . .	5
1.3.2	Plant . . . . .	5
1.3.3	Processing platform . . . . .	8
1.4	Software . . . . .	13
1.4.1	Main software in NCS nodes . . . . .	14
1.4.2	DCSMonitor . . . . .	20
1.4.3	Code availability . . . . .	23
1.5	Course activities . . . . .	23
1.6	Conclusions . . . . .	26
<b>2</b>	<b>Course program</b>	<b>31</b>
2.1	Introduction . . . . .	31
2.2	Simulation of NECS . . . . .	34
2.2.1	Control design . . . . .	34
2.2.2	Analysis of multitasking embedded control systems (ECS) . . . . .	40
2.2.3	Design of multitasking embedded control systems (ECS)	41
2.2.4	Analysis of networked control systems (NCS) . . . . .	42
2.2.5	Design of networked control systems (NCS) . . . . .	44
2.2.6	Analysis and design of event-driven control systems (EDCS) . . . . .	45
2.3	Implementation of NECS . . . . .	49
2.3.1	Introduction to the development framework . . . . .	49
2.3.2	Single control task system . . . . .	54
2.3.3	Two tasks system: standard controller and dummy task	56

2.3.4	Two tasks system: one-shot controller and dummy task	57
2.3.5	Networked control system . . . . .	59
2.3.6	Event-driven control . . . . .	66
<b>3</b>	<b>User manual: live CD</b>	<b>67</b>
3.1	Minimum requirements . . . . .	67
3.2	Booting the live CD . . . . .	68
3.3	Step by step Live CD guide . . . . .	69
3.3.1	Live CD structure . . . . .	69
3.3.2	Compiling DSPIC_SENSOR with Eclipse . . . . .	70
3.3.3	Programming DSPIC_SENSOR with MplabX . . . . .	73
3.3.4	Summary to program the other devices . . . . .	76
3.3.5	DCSMonitor . . . . .	78
3.4	Installing . . . . .	83
<b>4</b>	<b>Quick guide to start working</b>	<b>88</b>
4.1	Development Environment . . . . .	88
4.1.1	Mplab installation . . . . .	89
4.1.2	Eclipse installation . . . . .	93
4.1.3	Installing RTDruid plugin on Eclipse . . . . .	93
4.2	Led Blinking . . . . .	98
4.2.1	Eclipse environment . . . . .	98
4.2.2	Mplab X environment . . . . .	103

# Chapter 1

## Research report

### 1.1 Introduction

Networked control systems [1, 2], i.e. control loops closed over communication networks where sensors, controllers and actuators are physically distributed and exchange control data through a shared network, are gaining increased attention in many control application areas due to their cost-effectiveness, reduced weight and power requirements, simple installation and maintenance, and high reliability. At the same time, the underlying required control theory is starting to offer mature and methodological results, e.g. [3, 4].

In a parallel track, since the economic importance of embedded systems has grown exponentially as electronic components are in everyday use devices, embedded systems education is becoming an strategic asset. Hence, university curricula are being adapted accordingly to cover this domain [5]. In addition, noting that many embedded systems are control systems [6] and considering the importance of NCS in industrial processes, there is a growing demand of including NCS courses in the education of embedded systems engineers.

The traditional teaching approach to the diverse disciplines involved in NCS such as control systems, real-time computing and communication systems, can be often quite math-intensive and abstract, thus failing to introduce students to the realities of NCS implementation. Hence, laboratory activities are crucial to consolidate the diverse theoretical material. Following this trend, this paper presents a hands-on course in networked control systems to be integrated in the education of embedded control systems engineers. The course activities have a strong practical component and most of

them are applied exercises to be implemented in a NCS setup. This course can be taken as a complimentary material to other exiting courses in NCS and to other initiatives related to NCS education such as the "Networked Control Systems Repository" at <http://filer.case.edu/org/ncs/index.htm>, the NCS wiki page course at [http://www.cds.caltech.edu/~murray/wiki/index.php/EECI08:\\_Introduction\\_to\\_Networked\\_Control\\_Systems](http://www.cds.caltech.edu/~murray/wiki/index.php/EECI08:_Introduction_to_Networked_Control_Systems), or other similar efforts. The proposal in this paper extends to a networked setup a previously presented laboratory experiment targeting microprocessor-based real-time control systems [7].

After describing the basics on NCS (Section 1.2), the paper describes the experimental setup from a hardware (Section 1.3) and software point of view (Section 1.4), and then proposes several activities (Section 1.5) that can be shaped into a course program according to the needs and diverse background of the targeted audience. Section 1.6 concludes the paper.

## 1.2 Background on NCS

NCS take different forms, and two major types of control systems can be identified: shared-network control systems and remote control systems. The hands-on course proposal targets the first type, although several concepts can also be applied to the second type.

Hence, the course context is the NCS illustrated in Figure 1.1. Several control loops, each one formed by a sensor, a controller and an actuator implemented in physically separated nodes, share a single broadcast domain to exchange the control data required for each control loop operation. In addition, other nodes, represented by the load boxes, also use the network to exchange other non-control data.

For a given networked closed loop system, a control job will denote the required operations for each plant update. Hence, each control job would basically require sending the sensor reading in the sensor message after sampling the plant, and sending the control signal in the control message after computing its value in the controller node using the information contained in the incoming sensor message. Thus, in terms of bandwidth utilization, each control job simplifies to sending two messages, the sensor and the control message.

The most common design and implementation approach for NCS consists in the periodic execution of the control algorithm, which implies that messages from sensors to controllers and from controllers to actuators are periodic [8]. The course will cover these methods, but will also tackle other

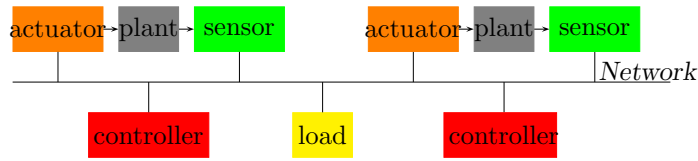


Figure 1.1: Networked control system scheme

approaches that go beyond the periodicity of the standard approach. Among them, two new tendencies for the analysis and design of NCS can be identified. The first one is to apply rate adaptation techniques where the period is selected according to the controlled system dynamics and/or to the bandwidth conditions, e.g. [9–11]. The main goal of these approaches is to improve the aggregated control performance for the set of control loops by efficiently using all the communication bandwidth. The second tendency is to apply event-based sampling techniques which produce non-periodic executions of the control algorithm, and therefore, non-periodic messaging in the network, e.g. [12–14]. The main goal of these approaches is to minimize the bandwidth utilization while still guaranteeing stability and acceptable control performance.

## 1.3 Experimental set-up

### 1.3.1 Introduction

The desired scheme for each experimental setup is illustrated in Figure 2.3. Hence, each student (or student group) will have a two-node NCS in which one node acts as a controller (left node) while the other node acts as a sensor and actuator (right node) and is attached to the plant. The reason for putting together the sensor and actuator in the same node is to save hardware resources.

The controlled plant and processing platform (hardware, real-time operating system, and network) have been carefully selected to have a friendly, flexible, and powerful experimental set-up.

### 1.3.2 Plant

Many standard basic and advanced controller design methods rely on the accuracy of the plant mathematical model. The more accurate the model, the more realistic the simulations, and the better the observation of the

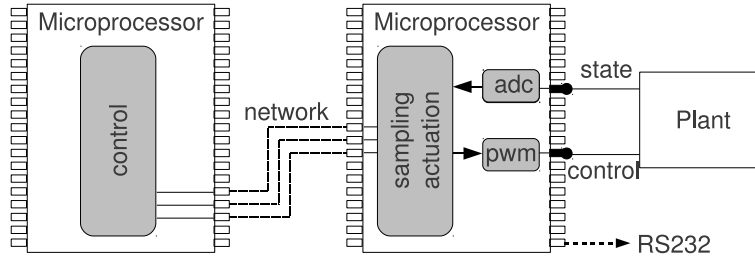


Figure 1.2: Experimental setup.

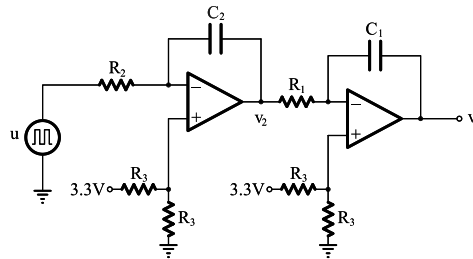


Figure 1.3: Plant: electronic double integrator (DI) circuit

effects of the controller on the plant. Hence, the plant was selected among those for which an accurate mathematical model could easily be derived.

Following the same reasons discussed in [7], a simple electronic circuit in the form of a double integrator (Figure 1.3) was selected<sup>1</sup>. The relative simplicity of its components together with the inherent unstable dynamics that makes the control more challenging have been the main reasons for its selection. Note however that experiments using other plants can be complementary to the approach presented here.

The selection of an electronic circuit as a plant has also an important advantage: depending on the specific circuit, it can be directly plugged into a micro-controller without using intermediate electronic component. That

<sup>1</sup>Note that in the integrator configuration, the operational amplifiers require positive and negative input voltages. Otherwise, they will quickly saturate. However, since the circuit is powered by the micro-controller, and thus no negative voltages are available, the 0V voltage ( $V_{ss}$ ) in the non-inverting input has been shifted from GND to half of the value of  $V_{cc}$  (3.3V) by using a voltage divider  $R_3$ . Therefore, the operational amplifier differential input voltage can take positives or negatives values.



Component	Value
$R_3$	$1k\Omega$
$R_1$	$100k\Omega$
$R_2$	$100k\Omega$
$C_1$	$470nF$
$C_2$	$470nF$

Table 1.1: Electronic components values

is, the transistor-transistor logic (TTL) level signals provided by the micro-controller can be enough to carry out the control. Note that this is not the case, for example, for many mechanical systems. Such a simplification in terms of hardware reduces the modeling effort to study the plant and no models for actuators or sensors are required.

The DI nominal electronic components are shown in table 1.1.

The operational amplifier in integration configuration [15] can be model by

$$V_{\text{out}} = \int_0^t -\frac{V_{\text{in}}}{RC} dt + V_{\text{initial}} \quad (1.1)$$

where  $V_{\text{initial}}$  is the output voltage of the integrator at time  $t = 0$ , and ideally  $V_{\text{initial}} = 0$ , and  $V_{\text{in}}$  and  $V_{\text{out}}$  are the input and output voltages of the integrator, respectively.

Taking into account (1.1), and the scheme shown in Figure 1.3, the double integrator plant dynamics can be modeled by

$$\begin{aligned} \frac{dv_2}{dt} &= \frac{-1}{R_2 C_2} u \\ \frac{dv_1}{dt} &= \frac{-1}{R_1 C_1} v_2 \end{aligned}$$

In state space form, the model is

$$\begin{aligned} \begin{bmatrix} \dot{v}_1 \\ \dot{v}_2 \end{bmatrix} &= \begin{bmatrix} 0 & \frac{-1}{R_1 C_1} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{-1}{R_2 C_2} \end{bmatrix} u \\ y &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \end{aligned}$$

Taking into account the tolerances in the electronics components (5% for resistors and 25% for capacitors), the model that best adapts to the real plant is given by the values listed in table 1.2.

Component	Value
$R_1$	$100k\Omega$
$R_2$	$100k\Omega$
$C_1$	$420nF$
$C_2$	$420nF$

Table 1.2: Validated values for the electronics components.

The model validation has been performed applying a standard control algorithm with a sampling period of  $h = 50\text{ms}$ , with reference changes, and comparing the theoretical results obtained from a Simulink<sup>®</sup> model with those obtained from the plant. Hence, with the validated values for the components, the model is given by

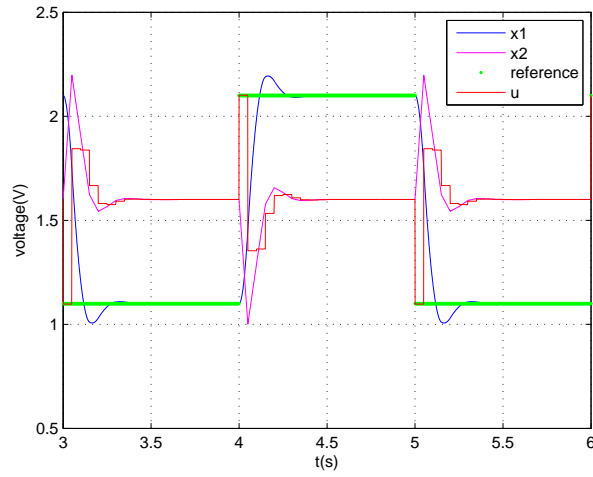
$$\begin{aligned} \dot{x} &= \begin{bmatrix} 0 & -23.809524 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ -23.809524 \end{bmatrix} u \\ y &= \begin{bmatrix} 1 & 0 \end{bmatrix} x \end{aligned} \quad (1.2)$$

Figure 1.4 shows the results of this validation. In particular, the controller gain designed via pole placement locating the desired closed loop poles at  $\lambda_{1,2} = -15 \pm 20i$  is  $K = \begin{bmatrix} 0.5029 & -0.9519 \end{bmatrix}$ . Since the voltage input of the operational amplifier is 1.6V (which is half  $V_{cc}$ : the measured  $V_{cc}$  is 3.2V although it is powered by 3.3V), the tracked reference signal has been established to be from 1.1V to 2.1V ( $\pm 0.5\text{V}$  around 1.6V). For the tracking, the feedforward matrix  $N_x$  is zero and  $N_x = \begin{bmatrix} 1 & 0 \end{bmatrix}$ .

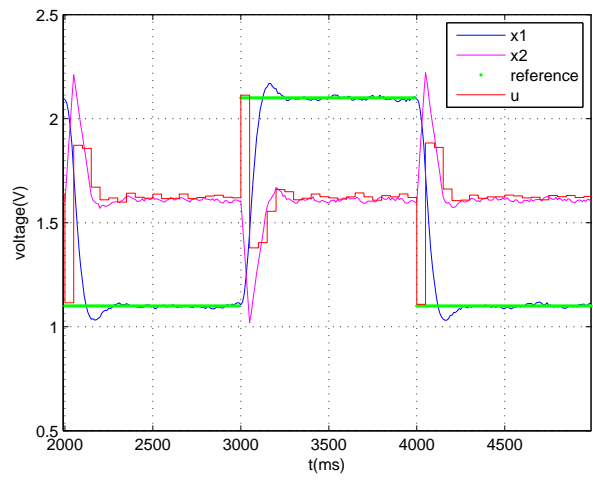
Note that the plant is unstable because the eigenvalues of the system matrix are  $\lambda_{1,2} = 0$ . The goal of the controller is to make the circuit output voltage ( $v_1$  in Figure 1.3) to track a reference signal by giving the appropriate voltage levels (control signals)  $u$ . Both states  $v_1$  and  $v_2$  can be read via the Analog-to-Digital-Converter (ADC) port of the micro-controller and  $u$  is applied to the plant through the Pulse-Width-Modulation (PWM) port.

### 1.3.3 Processing platform

The processing platform consists of the hardware platform, the real-time operating system and the network. As hardware platform, a micro-controller based architecture was selected because NCS are typically implemented using this type of hardware. As discussed in [7], for the processing platform adopting the Flex board [16] (in its full version, Figure 1.5) equipped with a Microchip dsPIC<sup>®</sup> DSC micro-controller dsPIC33FJ256MC710 represents



(a) Theoretical simulated plant response



(b) Experimental plant response

Figure 1.4: Model validation.

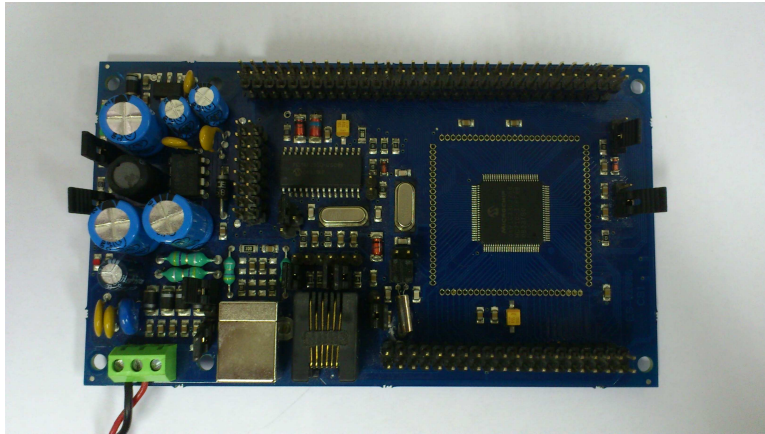


Figure 1.5: Full Flex board.

a good compromise between cost, processing power, and programming flexibility.

And regarding the real-time operating system, Erika Enterprise real-time kernel [16] was selected because it provides full support to the Flex board in terms of drivers, libraries, programming facilities, and sample applications, and it gives support for preemptive and non-preemptive multitasking, and implements several scheduling algorithms [17]. In addition, its API provides support for tasks, events, alarms, resources, application modes, semaphores, and error handling, permitting to enforce real-time constraints to application tasks to show students the effects of sampling periods, delays and jitter on control performance.

Regarding the network, the CAN (Controller Area Network, [18]) protocol that was originally designed for the automotive industry, but it has also become a popular bus in industrial automation as well as other applications has been selected. CAN provides the basis for many cost-effective distributed embedded applications, and its properties and functionality such as reliability, dependability, or clock synchronization, are being constantly enhanced.

Once the platform is ready, the networked setup for each student looks like as in Figure 2.3 (top). The bottom board acts as a (remote) controller, and communicates via CAN with the top board, that acts as a sampler and actuator. Note that the same hardware, micro-processor based control can also be tested. In this case, the bottom board is not used, and the top board performs all the activities: sampling, control algorithm computation

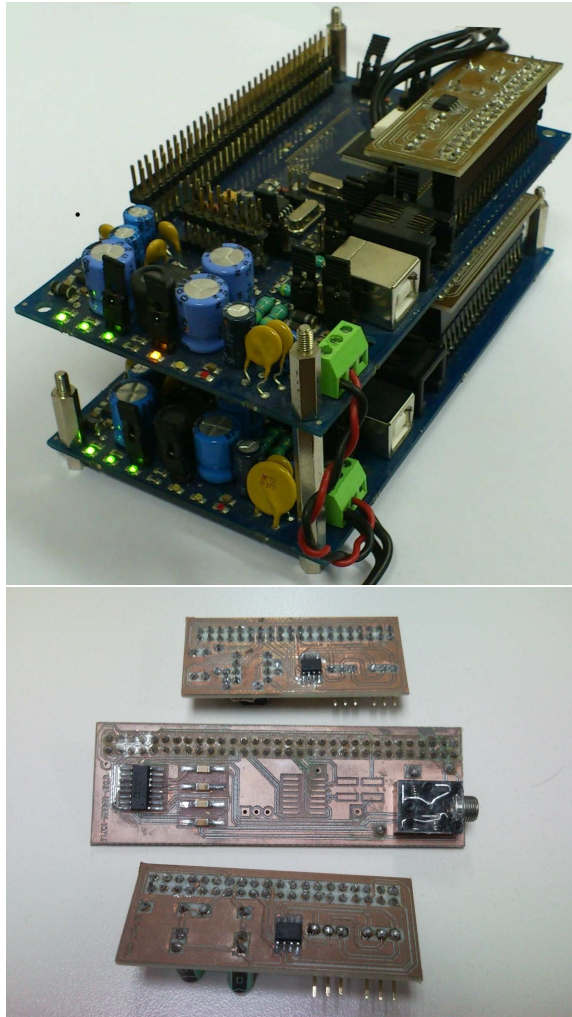


Figure 1.6: Controller-sensor/actuator hardware (top). Details of the CAN, DI and RS232 daughter boards (bottom)

and actuation. The daughter boards plugged into the top and bottom board, illustrated in Figure 1.6 (bottom), serve different objectives: the daughter board in the bottom is for CAN communication, the one in the middle is for RS232 communication (for monitoring) and the top daughter board is the DI circuit also enabled with CAN communication.

It is interesting to note that the modular architecture design permits to network all the different students setups in a single CAN network. In

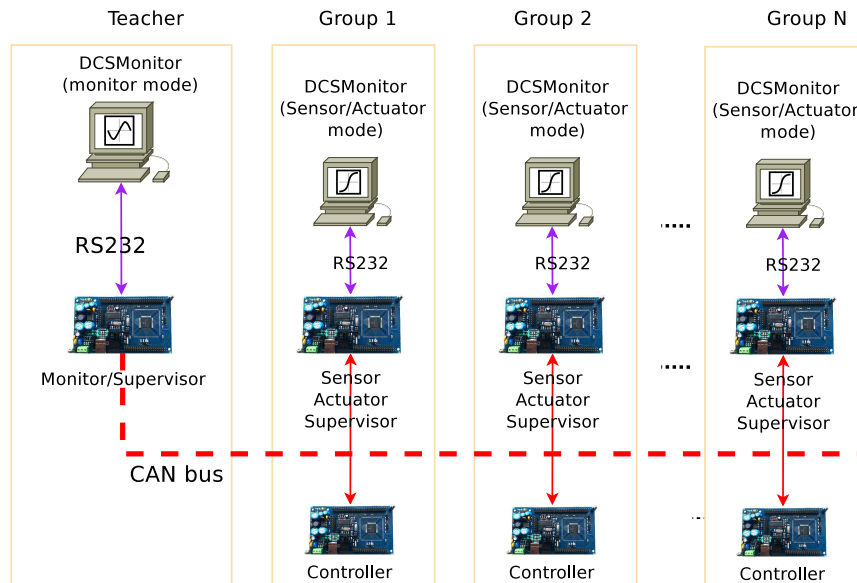


Figure 1.7: Scheme of the overall networked control system.

this way, a full networked control system can be built, where several nodes acting as controllers, sensors and actuators control different double integrator systems. This richer setup will permit to observe the interaction among different closed-loops in terms of bandwidth utilization and control performance.

In addition to all the pair of controller and sensor/actuator nodes that constitute several loops closed over the network, another node can be added to the network acting as a monitor/supervisor. The hardware of this node is again the full Flex board, equipped with the daughter board with CAN communication and the RS232 board used for debugging purposes. The role of this node is to gather information of the state of all the control loops, as well as to monitor and manage the network bandwidth. This node would be mainly used for the instructor/teacher, although it can be also used by students.

The complete scheme showing  $N$  control loops together with the supervisor/monitor is illustrated in Figure 1.7. In the figure, each group is a pair of controller and sensor/actuator nodes that are connected using CAN (that would correspond to the setup shown in Figure 2.3 top) and it is the hardware setup for each student (or group of students). Then, all the groups can be networked between them, using also CAN, and the monitor/supervisor

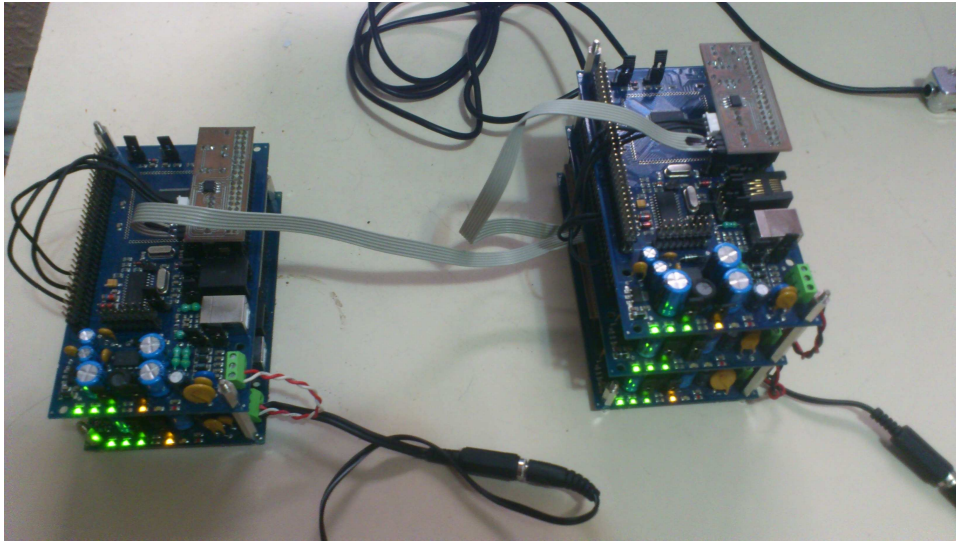


Figure 1.8: Example of networked control system.

node can also be attached to the network.

Figure 1.8 shows an example of the full NCS with two groups, together with the monitor. In this case, one of the groups has the controller-sensor/actuator hardware and the monitor hardware plugged together in a single tower (on the right).

## 1.4 Software

This section briefly describes the main software components that have been prepared for the experimental setup. Two types can be distinguished. First, the software that goes to each node (Flex board) and second, the software that can be run in an external PC for monitoring purposes, and that is called “DCSMonitor” in Figure 1.7. The node codes, the DCSMonitor, and other information related to this hands-on course are available at <http://code.google.com/p/pfc-platform-test/source/browse/>. Compared to the lab proposed in [7], where the plant dynamics were monitored from Matlab, the DCSMonitor provides a set of richer and flexible features to monitor the NCS. It has to be noted that the DCSMonitor and all the information that it monitors still can be used with the old Matlab monitoring system.

```

CPU mySystem {
  OS myOs {
    EE_OPT = "DEBUG"; CPU_DATA = PIC30 {
      APP_SRC = "setup.c"; APP_SRC = "e_can1.c";
      APP_SRC = "code.c";
      MULTI_STACK = FALSE; ICD2 = TRUE;};
    MCU_DATA = PIC30 {MODEL = PIC33FJ256MC710;};
    BOARD_DATA = EE_FLEX {USELEDS = TRUE;};
    KERNEL_TYPE = EDF { NESTED_IRQ = TRUE;
      TICK_TIME = "25ns";};};
    TASK TaskReferenceChange {
      REL_DEADLINE = "0.005s"; PRIORITY = 3;
      STACK = SHARED; SCHEDULE = FULL;};
    TASK TaskController {
      REL_DEADLINE = "0.05s"; PRIORITY = 3;
      STACK = SHARED; SCHEDULE = FULL;};
    COUNTER myCounter;
    ALARM AlarmReferenceChange {
      COUNTER = "myCounter";
      ACTION = ACTIVATETASK {
        TASK = "TaskReferenceChange"; };};
    ALARM AlarmController {
      COUNTER = "myCounter";
      ACTION = ACTIVATETASK {
        TASK = "TaskController"; };};
  };
};

```

Figure 1.9: Configuration file (conf.oil) for the controller node

#### 1.4.1 Main software in NCS nodes

The description of the Erika codes is ordered according to the three type of nodes: controller, sensor/actuator, monitor/supervisor. For each node, the kernel configuration file *conf.oil* specifies the main parameters for the dsPIC and real-time kernel, and the *code.c* contains the main functionality.

##### Controller node

The configuration file is shown in Figure 1.9. It specifies the C files that are used in this node, the scheduling algorithm (EDF, Earliest Deadline First), and it defines two tasks, *TaskReferenceChange* and *TaskController*, which are implemented in the *code.c* and associated to an alarm to control their periodicity of execution.



```

TASK(TaskReferenceChange){
    if (r == -0.5){r=0.5; LATBbits.LATB14 = 1;
    }else{ r=-0.5; LATBbits.LATB14 = 0;}
    Send_Controller_ref_message(&r);
}
TASK(TaskController){
    x0=(p_x0); //Get state x[0] from CAN msg
                //from rx_ecan1_message1[0]..[3] data field
    x1=(p_x1); //Get state x[1] from CAN msg
                //from rx_ecan1_message1[4]..[7] data field
    x_hat[0]=x0-r*Nx[0]; x_hat[1]=x1-r*Nx[1];
    u_ss=r*Nu;
    u=-k[0]*x_hat[0]-k[1]*x_hat[1]+u_ss;
    /* Check for saturation */
    if (u>v_max/2) u=v_max/2;
    if (u<-v_max/2) u=-v_max/2;
    Send_Controller2Actuator_message(&u);
}
int main(void){
    Sys_init();
    SetRelAlarm(AlarmReferenceChange,1000,1000);
    for (;;) return 0;
}

```

Figure 1.10: Main parts of the code.c for the controller node

The *code.c* file in the node has three main parts (Figure 1.10): the *main*, and the two task defined in the *conf.oil*. The *main* configures the **TaskReferenceChange** whose role is to create the reference signal to be tracked by the controller. The **TaskReferenceChange** code simply generates a square wave that switches from  $-0.5\text{v}$  to  $0.5\text{v}$  each second, and this information is sent over CAN for debugging purposes. The **TaskController** is activated by interrupt upon reception of the sensor message delivered over CAN. It gets the plant state for the message, implements a control law (in the figure a state-feedback controller with a tracking configuration), and sends the computed control signal over CAN in the control message. The *code.c* also manages the CAN interruption to receive the message sent by the sensor.

```

TASK(TaskSensor){
    LATBbits.LATB14 ^= 1; Read_State();
    Send_Sensor2Controller_message(&x[0]);
}
TASK(TaskSensor_supervision){
    LATBbits.LATB14 ^= 1; Read_State();
    Send_Sensor2Supervisor_message(&x[0]);
}
TASK(TaskActuator){
    u>(*p_u); // Gets control signal u from CAN msg
    PDC1=((*p_u)/v_max)*0x7fff+0x3FFF;
}
TASK(TaskSupervision){
    /* It sends data via RS-232 to the PC
}
int main(void){
    Sys_init();
    SetRelAlarm(AlarmSensor,1000,50);
    SetRelAlarm(AlarmSupervision, 1000, 10);
    SetRelAlarm(AlarmSensor_supervision, 1000, 10);
    for (;;) return 0;
}

```

Figure 1.11: Main parts of the code.c for the sensor/actuator node

### Sensor/Actuator node

The sensor/actuator node configuration file is very similar to the one in the controller node. However, it defines different tasks: `TaskSensor`, `TaskSensor_supervision`, `TaskActuator`, `TaskSupervision`, that are code in the corresponding *code.c* file.

Figure 1.11 shows the main parts of the *code.c* file. In the `main`, the periodicity for the sensor task is defined to 50ms and the periodicity for the supervision tasks are defined to 10ms. Note that the periodicity of the actuator task is not defined because it is executed upon reception of the control message send by the controller node. The `TaskSensor` reads the plant state and sends this value in the sensor message over CAN using the periodicity established in the controller design stage. The `TaskSensor_supervision` sends the plant state over CAN every 10ms but with very low priority, that will be used for monitoring purposes, and in particular, for being able to plot the plant dynamics in the DCSMonitor. Similarly, the `TaskSupervision`

```

TASK(TaskControllerMonitor){
    x0=(p_x0);//Get state x[0] from CAN msg
        //from rx_ecan1_message1[0]..[3] data field
    x1=(p_x1);//Get state x[1] from CAN msg, [4]..[7] data field
}
TASK(TaskSensor_supervision){
    x0=(p2_x0);//Get state x[0] from CAN msg
        //from rx_ecan1_message1[0]..[3] data field
    x1=(p2_x1);//Get state x[1] from CAN msg, [4]..[7] data field
}
TASK(TaskActuatorMonitor){
    u=(*p_u);
    x0=(p_x0);//Get state (control signal) x[0] from CAN msg
    x1=(p_x1);//Get state (control signal) x[1] from CAN msg
}
TASK(TaskSupervision){
/* It sends data via RS-232 to the PC
}
TASK(TaskToggleLed){
    LATBbits.LATB14 ^= 1;//Toggle orange led
}
TASK(TaskCANUseless){
    Send_CAN_useless();
}
int main(void){
Sys_init(); init_devices_list(); for (;;) return 0;
}

```

Figure 1.12: Main parts of the code.c for the monitor/supervisor node

sends a similar information through the RS232 port, also for monitoring purposes (for the DCSMonitor when is used by the same student group, or for using the old Matlab monitoring system). Finally, the TaskActuator, upon reception of the control message, takes the control signal from the message, and applies it to the plant using the PWM.

### Monitor/supervision

This node is the special purpose node that gathers information from all the control loops and network. It communicates with the DCSMonitor software that runs in an external PC, that will be described in next section.

The *conf.oil* for this node defines the TaskSupervision, TaskActuatorMonitor,

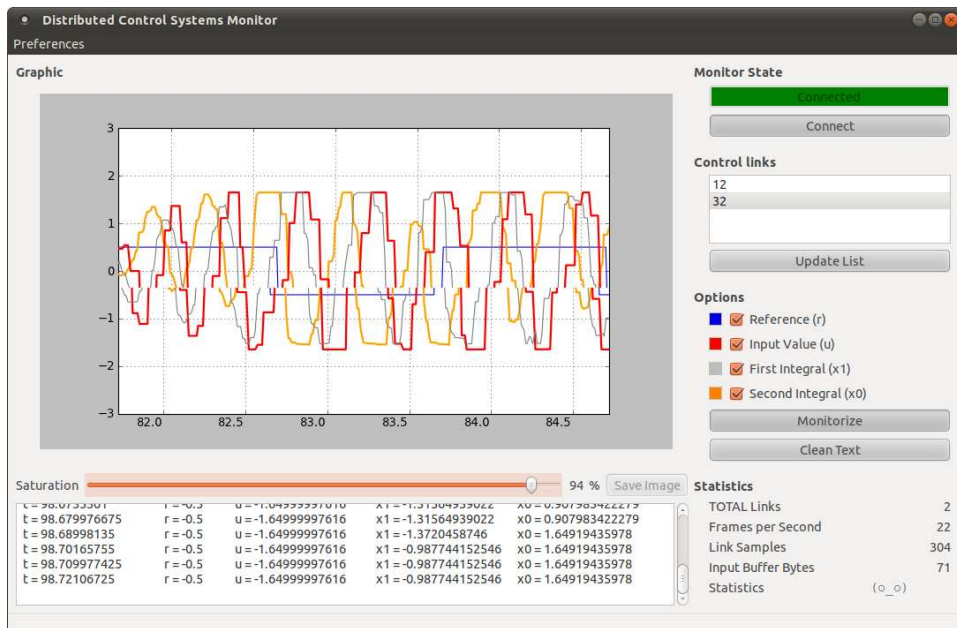


Figure 1.13: DCSMonitor.

TaskControllerMonitor, TaskSensor\_supervision, TaskToggleLed and the TaskCANUseless, which are coded in the *code.c* file.

The TaskSupervision sends data over the RS232 port with the same purpose that had in the sensor/actuator node. The TaskActuatorMonitor mainly obtains the control signal for a given control loop, while the tasks TaskControllerMonitor, TaskSensor\_supervision obtain the state of a given plant (one reading the state sent by the sensor for controlling purposes and the other reading the state sent by the sensor for monitoring purposes). The TaskToggleLed blinks a led, and the TaskCANUseless is used to regulate the bandwidth of the network by sending dummy messages.

Although not detailed here, the full *code.c* also implements the code that manages the interaction between the monitor/supervision node and the DCSMonitor software, which communicate over RS232.

### Nodes tasks summary

Table 1.3 summarizes the main tasks coded in each node, indicating to which node it belongs, how the task is activated, and its function.

<b>Task</b>	<b>Node</b>	<b>Activation</b>	<b>Description</b>
TaskReferenceChange	Contrl.	Periodic	generates the reference signal to be tracked by the controller
TaskController	Contrl.	on msg rx	computes the control signal and sends $u$ over CAN for closing the loop
TaskSensor	S/A	Sampling period	samples the plant and sends $x$ over CAN for closing the loop
TaskSensor_supervision	S/A	Monitoring period	samples the plant and sends $x$ over CAN for DCSMonitor monitoring
TaskActuator	S/A	on msg rx	applies the control signal $u$ to the plant
TaskSupervision	S/A	Monitoring period	sends loop state in a 23-byte frame via RS232 for “student” DCSMonitor monitoring
TaskSupervision	Monitor	Monitoring period	sends loop state in a 71-byte frame via RS232 for “teacher” DCSMonitor monitoring
TaskActuatorMonitor	Monitor	on msg rx	obtains the control signal from the incoming CAN msg sent by the controller
TaskControllerMonitor	Monitor	on msg rx	obtains the state from incoming CAN msg sent by the S/A
TaskSensor_supervision	Monitor	on msg rx	obtains the state from incoming monitoring CAN msg sent by the S/A
TaskToggleLed	Monitor	nop	not used
TaskCANUseless	Monitor	progressive	sends dummy CAN messages to load CAN with a progressive periodicity

Table 1.3: Tasks summary

Node	Value	Meaning
SIGNAL_STOP	0x01	It cancels current actions
SIGNAL_MONITOR	0x00	To monitor a particular control loop
SIGNAL_PERCENT	0x02	To generate artificial load in the bus
SIGNAL_DEVICES	0x04	To list all active control loops

Table 1.4: Commands from DCSMonitor to the monitor/supervisor node

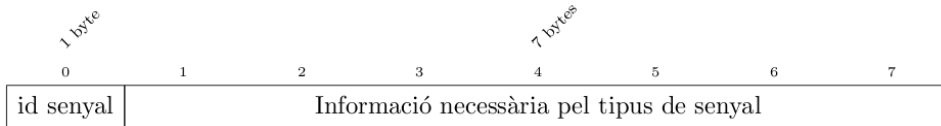


Figure 1.14: RS232 communication frame for sending commands from the DCSMonitor to the monitor/supervisor node.

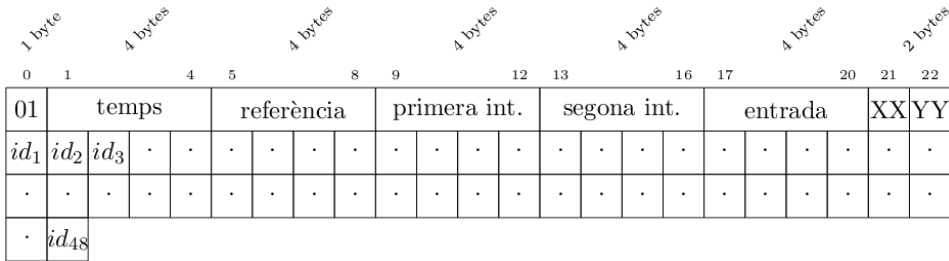


Figure 1.15: RS232 communication frame for sending commands from the monitor/supervisor node to the DCSMonitor.

### 1.4.2 DCSMonitor

The DCSMonitor is a monitoring program that can be used by each student group to monitor its control loop dynamics, but it has been mainly designed for the instructor/teacher to allow monitoring any of the group control loops, as well as to regulate the network bandwidth. It has been coded with Phyton.

Figure 1.13 provides a general view of the DCSMonitor. It can perform three main activities. First, it permits to monitor the number of control loops that are exchanging data over the network (Control links in the figure). It also displays the dynamics (reference signal, control signal and states) of a given control loop, either numerically or graphically. And finally, it has a slider bar that permits to regulate the number of dummy messages that are

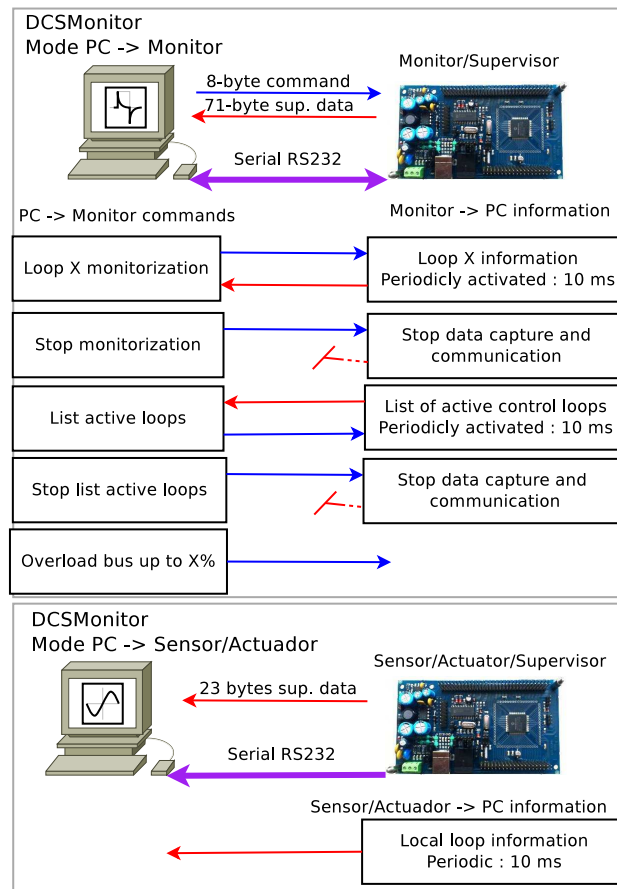


Figure 1.16: RS232 communication between the monitor/supervisor node and the DCSMonitor. Top: teacher setup. Bottom: student group setup.

sent over the network to create different bandwidth loads.

To allow this functionality, the communication between the monitor/supervisor node and the DCSMonitor software over RS232 was configured to be bidirectional. On one hand, the DCSMonitor sends control commands to the monitor/supervisor node using a simple 8-byte frame coded according to an identifier (Figure 1.14), whose values and meaning are summarized in table 1.4. For example, to monitor a particular loop, a frame with 0x00 in the id field will be sent, where in the last byte the particular control loop number will be identified. If later on a frame with 0x01 with empty data is sent, the monitoring of that particular loop will terminate.

On the other hand, the monitor/supervisor node sends to the DCSMon-

Msg	TX node	RX node	Description
Sensor-to-controller	S/A	Controller	sends the state for control loop operation. Also received by the Monitor
Controller-to-sensor	Controller	S/A	sends the control input for control loop operation. Also received by the Monitor
Reference	Controller	S/A	sends the reference signal for monitoring to both the S/A and Monitor nodes
Sensor-to-monitor	S/A	Monitor	sends the loop state for monitoring
Useless	Monitor	CAN	dummy message to regulate the CAN load

Table 1.5: CAN messages

itor a more complete 71-byte frame ((Figure 1.14)) in which the monitor/supervisor node communicates information (state of the plant and control signals) about the control loop under supervision, as well as the list of active control loops if required (up to 48). In the case that the DCSMonitor is used for a student group to monitor its own control loop dynamics, the information is sent by the sensor/actuator node of that particular control loop using a medium 23-byte frame that correspond to the first 23 bytes of the long frame. A scheme of this data exchange is shown in Figure 1.16, where the top sub-figure shows the communication when the DCSMonitor is used by the teacher while the bottom sub-figure shows the communication when the DCSMonitor is used by a group of students.

In order to allow the monitor/supervisor node to send accurate monitoring information to the DCSMonitor software, the CAN messages used in each control loop were coded following different guidelines. Table 1.5 summarizes the set of CAN messages used for control and monitoring purposes. Regarding the codification of the CAN messages, first, different levels of priority were required. In addition, it was required to be able to identify different control loops, and in each control loop, different message classes, and even subclasses. This has been achieved by coding each message identifier as shown in Figure 1.17.

The message type field permits to distinguish a) control messages (000) that are sent from any controller node to its sensor/actuator node and must be of higher priority (they carry a 4-byte control signal value in the data field), b) sensor messages (001) that are sent from any sensor/actuator node to its controller node (they carry a 8-byte two states values in the data field), and c) general purpose messages (010) that depending on the subclass spec-



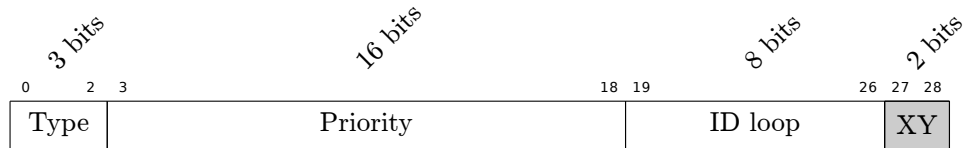


Figure 1.17: CAN message codification.

ifies reference/change messages (00) that are sent by the controller (they carry a 4-byte reference value in the data field), and supervisory messages (01) that are sent by the sensor (they carry a 4-byte control signal value in the data field). The latter class is non-critical communication, and therefore, they have the lowest priority although their periodicity is high in order to allow the DCSMonitor to plot an accurate graph of the systems dynamicis. Using this coding, and filling up each data frame with the appropriate information, the monitor/supervisor node can gather all the plants information to be send to the DCSMonitor via RS232 whenever required.

### 1.4.3 Code availability

The code for the DCSMonitor as well as for all nodes is available at <http://code.google.com/p/pfc-platform-test/source/browse/>. The source three looks like in Figure 1.18. The repository also contains diverse documentation (some in English but mainly in Catalan) related to the project.

## 1.5 Course activities

This section presents some of the activities to carry out in the hands on course using the presented setup. To start with, basic control theory may be needed to establish a common knowledge among all the students. It could include

- State-space system representation, in the continuous-time but more important in the discrete-time domain, and including delays
- Discrete-time systems analysis, including topics such as controllability and observability, as well as sampling period selection
- Control design by state feedback, stating with pole placement techniques and considering delays, together with observer and tracking structures

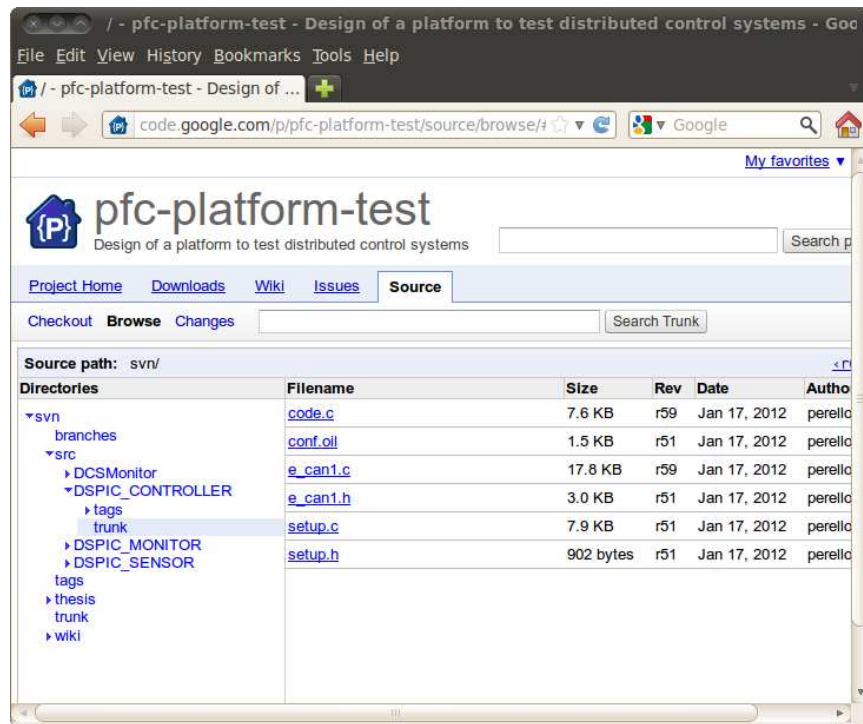


Figure 1.18: Source code.

From this basic theory, simulation and implementation exercises including not-NCS and NCS could be for example

1. **Open loop analysis (not-NCS):** Using for example Matlab/Simulink, to obtain the open loop system response of the double integrator circuit in front of a reference signal in the form of a square wave of amplitude from 1.5V to 2V and frequency 1Hz. And to perform a simple stability analysis by pole inspection. This can also be implemented in the hardware set-up, using only the sensor/actuator node acting also as a controller.
2. **Closed-loop design in the continuous-time domain (not-NCS):** To perform state feedback controller design via pole location in the continuous-time domain with tracking structure and assuming that both states can be measured, such that a stable circuit response is achieved and the control signal values range withing 3.38V and 0V. The first constraint is a control specification while the second one is a

hardware limitation.

3. **Closed-loop design in the discrete-time domain (not-NCS):** To carry out the state feedback controller design via pole location in the discrete-time domain with tracking structure and assuming that both states  $x_1$  and  $x_2$  are measured, such that the stable circuit response is achieved while the previous hardware constraint is still met. In this case, the sampling period and closed-loop pole locations for controller are can be obtained by standard rules-of-thumb. Alternatively, it can be specified a period of  $h = 0.05$  s, and to design a discrete state feedback controller placing the continuous closed loop poles at  $p_{1,2} = -5 \pm 20i$ , which meet the specifications. Again, this can also be implemented in the hardware set-up, using only the sensor/actuator node acting also as a controller. At this stage, different type of observers can be also designed, simulated and implemented.
4. **Delay effects in a single NCS (NCS):** To simulate for example with TrueTime (<http://www3.control.lth.se/truetime/>) the NCS for the double integrator system using the settings for period and desired poles as before, with sampling occurring in the sampler node, control algorithm executing in a controller node, and actuation taking place in the actuator node. As a network, use for example CAN, with different baud-rates and then to analyze, by observing different responses, how the communication delay affects the performance considering that the controller that applies is the one obtained previously. Note that in this case, the delay is constant. The simulation study can also be implemented by each group, locally, using the NCS setup, and monitoring the response in the DCSMonitor.
5. **Delay effects in a single NCS with different traffic loads (NCS):** To extend the previous simulation, adding more nodes sending additional traffic to the network and observe the effect of this new traffic on the performance of the double integrator control. Note that in this case, the delay varies. This simulation can be also implemented by each group, and in this case, the slider bar fixing the network load in the DCSMonitor can be used to create different network conditions.
6. **NCS simple controller design (NCS):** The degrading effects of delays can be treated using different approaches. A simple approach is to design the controller assuming an input delay in the system, which will result in extending the original state space system with a new state

variable that represents the previous control signal [19]. This imposes to place a third discrete-time pole for obtaining the state-feedback gain. It can be placed at 0. Then it can be noted that the application of this new controller is effective for a constant delay but not for the scenario of varying delay. The simulation can also be implemented in the setup. In addition, all the loops from all the students can be networked together, to see with a more realistic setting the effects of different traffic on control performance. The CAN bus baud-rate can also be used for experimental purposes.

7. **NCS advanced controller design (NCS):** In order to design a controller capable of dealing with varying time delays, several strategies are available, including control-centered approaches, see e.g. [1], or implementation-based approaches, such as [20]. One or more of them can be simulated and implemented.
8. **Rate adaptation techniques and event-driven NCS (NCS):** using the latest results on these areas (see section 1.2), existing results can also be simulated and implemented.

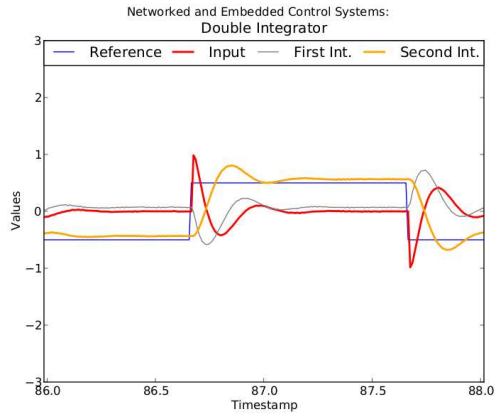
Note that the last exercises can have different levels of difficulty and may require non-basic control and real-time systems theory for its correct simulation and implementation.

Performing the previous activities students can implement a rich set of exercises. They will observe several DI dynamics according to the activity. Just for illustrative purposes, Figure 1.19 shows the DI response in the case of having different loads in the CAN network.

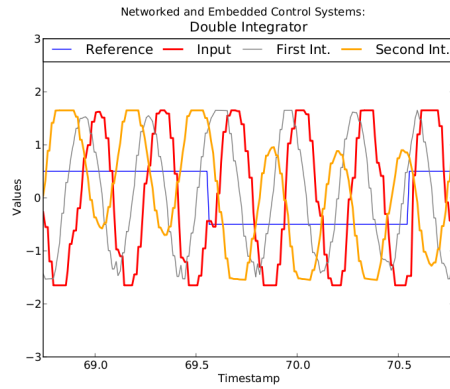
## 1.6 Conclusions

This paper has presented a hands-on course to be integrated in the education curriculum of embedded control systems engineers. The main focus of the course is NCS. The selection of the plant and processing platform has been discussed. Details of the code to be executed in each networked node have been presented. And a new software to be executed in an external PC for monitoring purposes has been explained. Finally, a set of course activities have been listed.

In summary, the proposed course poses several *real* challenges to the students that can be met by putting together interdisciplinary skills (electronics, real-time systems, control theory, programming) towards a single goal: building a networked control system.



(a) when bandwidth utilization is low



(b) when bandwidth utilization is high

Figure 1.19: DI dynamics.

# Bibliography

- [1] J.P. Hespanha, P. Naghshtabrizi and Yonggang Xu, “A Survey of Recent Results in Networked Control Systems,” *Proceedings of the IEEE* , vol.95, no.1, pp.138-162, Jan. 2007
- [2] R.A. Gupta and M.-Y. Chow , “Networked Control System: Overview and Research Trends,” *IEEE Transactions on Industrial Electronics* , vol.57, no.7, pp.2527-2535, July 2010
- [3] D. Nešić and D. Liberzon, “A unified framework for design and analysis of networked and quantized control systems,” *IEEE Transactions on Automatic Control*, Vol. 54, No. 4, pp. 732–747, April 2009.
- [4] W.P.M.H. Heemels, A.R. Teel, N. van de Wouw, D. Nešić , “Networked Control Systems With Communication Constraints: Tradeoffs Between Transmission Intervals, Delays and Performance,” *IEEE Transactions on Automatic Control*, vol.55, no.8, pp.1781-1796, Aug. 2010
- [5] D. J. Jackson and P. Caspi, “Embedded systems education: future directions, initiatives, and cooperation”, *SIGBED Rev.*, vol. 2, no. 4, pp. 1-4, Oct. 2005.
- [6] K.-E. Årzén, A. Blomdell, and B. Wittenmark, “Laboratories and real-time computing: integrating experiments into control courses,” *IEEE Control Systems Magazine*, vol. 25, no. 1, pp. 30-34, Feb. 2005.
- [7] P. Martí, M. Velasco, J.M. Fuertes, A. Camacho, G. Buttazzo, “Design of an Embedded Control System Laboratory Experiment,” *IEEE Transactions on Industrial Electronics*, vol.57, no.10, pp.3297-3307, Oct. 2010
- [8] D. Hristu-Varsakelis and W. S. Levine, *Handbook of Networked and Embedded Control Systems*, Birkhäuser Boston, June, 2008.

- [9] Reh binder, H. and Sanfridson, M., “Scheduling of a limited communication channel for optimal control,” *Automatica*, vol. 40, n. 3, pp. 491-500, March 2004.
- [10] Ben Gaid, M.E.M.; Cela, A.; Hamam, Y., “Optimal integrated control and scheduling of networked control systems with communication constraints: application to a car suspension system,” *IEEE Transactions on Control Systems Technology*, vol.14, no.4, pp. 776-787, July 2006.
- [11] P. Martí, A. Camacho, M. Velasco, M. El Mongi Ben Gaid, “Runtime Allocation of Optional Control Jobs to a Set of CAN-Based Networked Control Systems,” *IEEE Transactions on Industrial Informatics*, vol.6, no.4, pp.503-520, Nov. 2010
- [12] Hristu-Varsakelis, D., and Kumar, P.R., “Interrupt-based feedback control over a shared communication medium,” *41st IEEE Conference on Decision and Control*, Dec. 2002.
- [13] X. Wang and M. Lemmon, “Decentralized Event-triggering Broadcast over Networked Systems,” *Hybrid Systems: Computation and Control*, 2008.
- [14] A. Anta and P. Tabuada, “On the benefits of relaxing the periodicity assumption for networked control systems over CAN,” *Real Time Systems Symposium*, December 2009.
- [15] Wikipedia. Operational amplifier applications, 2012, [http://en.wikipedia.org/wiki/Operational\\_amplifier\\_applications](http://en.wikipedia.org/wiki/Operational_amplifier_applications), [Online; accessed 16-January-2012].
- [16] Evidence srl., <http://www.evidence.eu.com/>
- [17] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Norwell, MA, USA: Kluwer Academic Publishers, 1997.
- [18] CAN Specification version 2.0. Robert Bosch GmbH, Postfach 30 02 40, D-70442 Stuttgart, 1991.
- [19] K.J. Åström and B. Wittenmark, *Computer controlled systems*, Prentice Hall, 1997.

- [20] C. Lozoya, P. Martí, M. Velasco, J.M. Fuertes, “Analysis and design of networked control loops with synchronization at the actuation instants,” in 34th Annual Conference of IEEE Industrial Electronics, pp.2899-2904, 10-13 Nov. 2008



# Chapter 2

## Course program

### 2.1 Introduction

The objective of these labs is the analysis, design and implementation of networked and embedded control systems (NECS). Different steps must be covered:

- Control design: analysis and design of feedback controllers, in the continuous and discrete time domains, which can be mainly performed using a simulation tool such as Matlab/Simulink<sup>©</sup>.
- Analysis of multitasking NECS (1): analysis of the interactions that appear whenever several controllers have to be executed with limited computing resources, which can be mainly performed using a simulation tool such as TrueTime (<http://www.control.lth.se/truetime/>).
- Design of multitasking NECS (2): design of strategies that permit a *better* operation for the set of controllers concurrently executed with limited computing resources, which can be also performed for example with TrueTime
- Implementation of NECS: implementation of the strategies/designs obtained before.

The implementation platform will permit to control an electronic *Double Integrator (DI)* circuit from a micro-controller based architecture, as shown in Figures 2.5 and 2.8

In the first set-up, illustrated in Figure 2.5, several tasks will execute concurrently on top of the Erika real-time kernel. Therefore, the control

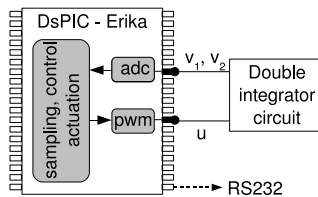


Figure 2.1: Microprocessor-based control.

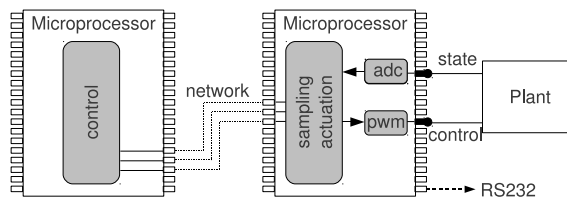


Figure 2.2: Network-based control.

tasks in charge of the double integrator circuit will compete with the rest of tasks for the CPU time.

The second set-up, illustrated in Figure 2.8, permits to close the loop over a network, which in this case is CAN (Controller Area Network). In this scenario, the limitation of resources is given by the communication bandwidth. Networking several loops in a single CAN network will allow the analysis of realistic networked control systems.

In both setups, the boards are equipped with a micro-controller dsPIC33FJ256MC710 (<http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en024663>) from Microchip. The boards are the Full Flex training board (<http://www.evidence.eu.com>) from Evidence. Three daughter boards with the DI circuit, CAN communications and a RS232 port has been plugged to the Flex boards. The dsPICs run the Erika, a multitasking real time kernel also from Evidence. The micro-controller and circuit interface is the sensor and actuator. The sensor is the analog/digital converter (ADC) to read the circuit output voltage  $V_1$ , and actuation is achieved through the pulse width modulator (PWM) by applying different voltage levels to the circuit input,  $u$ .

The control objective would be to have the circuit output voltage  $V_1$  (controlled variable) to track a reference signal while meeting for example given transient response specifications, which mandates to use tracking structures. The control will be achieved by sampling  $V_1$ , executing the con-

trol algorithm and applying the calculated control signal to the circuit via varying the circuit input voltage  $u$  (manipulated variable).

Once the platform is ready, the setup looks like as in Figure 2.3 (right). The bottom board acts as a remote controller, and communicates via CAN with the top board, that, in the networked scenario, acts as a sampler and actuator. In the mono-processor scenario, the bottom board is not used, and the top board performs all the activities: sampling, control algorithm computation and actuation. The daughter boards plugged into the top and bottom board, illustrated in Figure 2.3 (left), serve different objectives. The daughter board in the bottom board is for CAN communication. In the top board, one daughter board is the DI circuit and is enabled with CAN communication, and the other one is the RS232 board used for debugging purposes.

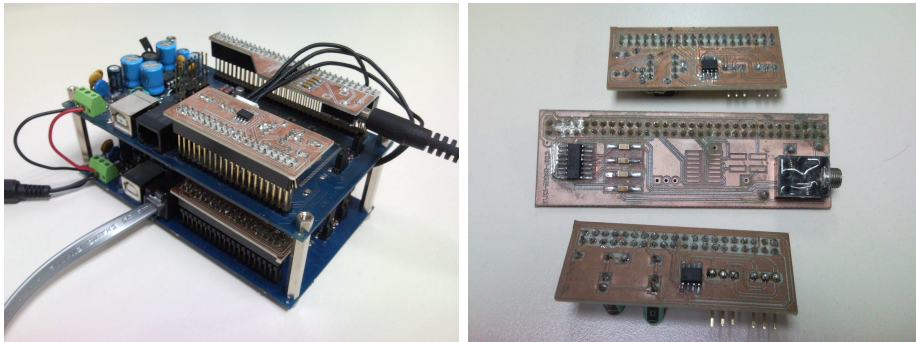


Figure 2.3: Full hardware scheme.

## 2.2 Simulation of NECS

### 2.2.1 Control design

#### Basics on control

Let

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{A}\mathbf{x}(t) + \mathbf{B}u(t) \\ y(t) = \mathbf{C}\mathbf{x}(t) \end{cases}$$

be the continuous time state-space representation of a system. With period  $h$ , the discrete-time state-space form is

$$\begin{cases} \mathbf{x}_{k+1} = \Phi(h)\mathbf{x}_k + \Gamma(h)u_k \\ y_k = \mathbf{C}\mathbf{x}_k \end{cases}$$

where  $\Phi(t)$  and  $\Gamma(t)$  can be obtained, with  $t = h$ , from

$$\begin{aligned} \Phi(t) &= e^{\mathbf{A}t} \\ \Gamma(t) &= \int_0^t e^{\mathbf{A}s}\mathbf{B}ds \end{aligned}$$

The sampling period determines the system dynamics. It should be chosen according to the desired continuous-time poles locations.

- if the fast closed-loop poles have to be complex (oscillatory response), the sampling period must be chosen such that between 10 and 20 samples ( $N$ ) must be taken for oscillation period of the desired response, that is

$$h = \frac{2\pi}{N\omega_n\sqrt{1-\xi^2}}$$

where  $\omega_n$  is the system natural frequency and  $\xi$  is the damping coefficient of the poles, which are given by

$$s = -\omega_n\xi \pm j\omega_n\sqrt{1-\xi^2} = -\omega_n\xi \pm \omega_n\sqrt{\xi^2-1}$$

which are the roots of the denominator of

$$\frac{\omega_n^2}{s^2 + 2\xi\omega_n s + \omega_n^2}$$

Recall that the oscillatory frequency is  $\omega_d = \omega_n\sqrt{1-\xi^2}$ , the percentage overshoot is  $SP = 100e^{\frac{-\xi\pi}{\sqrt{1-\xi^2}}}$ , the exponential involvements are  $1 \pm \frac{e^{-\xi\omega_n t}}{\sqrt{1-\xi^2}}$  and the settling time is  $t_s = -\frac{1}{\xi\omega_n} \ln(\% \sqrt{1-\xi^2})$  where % usually is 0.02 or 0.05

- if the fast poles are real, the sampling period must be chosen such that the number of samples taken during the rising time of the fast pole

$$\frac{1}{\tau s + 1}$$

should be between 4 and 10 ( $N$ ), that is

$$h < \frac{\tau}{N}$$

Care must be taken with systems of order bigger than two: slow poles determine the dynamics, but fast poles determine the selection of the sampling period.

Recall that controllability must be ensured by checking if the controllability matrix

$$\mathbf{W}_c = (\mathbf{\Gamma}, \mathbf{\Phi}\mathbf{\Gamma}, \mathbf{\Phi}^2\mathbf{\Gamma}, \dots, \mathbf{\Phi}^{n-1}\mathbf{\Gamma})$$

has rank  $n$ , and observability by checking if the observability matrix

$$\mathbf{W}_o = \begin{pmatrix} \mathbf{C} \\ \mathbf{C}\mathbf{\Phi} \\ \mathbf{C}\mathbf{\Phi}^2 \\ \vdots \\ \mathbf{C}\mathbf{\Phi}^{n-1} \end{pmatrix}$$

has also rank  $n$ .

Finally, for discrete-time controller design, the Ackerman formula is a standard procedure. Given

$$\begin{cases} \mathbf{x}_{k+1} = \mathbf{\Phi}\mathbf{x}_k + \mathbf{\Gamma}u_k \\ \mathbf{y}_k = \mathbf{C}\mathbf{x}_k + \mathbf{D}u_k \end{cases}$$

and the desired poles  $\lambda_1, \lambda_2, \dots, \lambda_n$ ,

$$p_d(z) = (z - \lambda_1)(z - \lambda_2) \cdots (z - \lambda_n) = z^n + b_1z^{n-1} + \cdots + b_{n-1}z + b_n$$

the controller gains  $K$  are given by the Ackerman formula

$$\mathbf{K} = (0 \ 0 \ 0 \ \cdots \ 1) (\mathbf{\Gamma} \ \mathbf{\Phi}\mathbf{\Gamma} \ \mathbf{\Phi}^2\mathbf{\Gamma} \ \cdots \ \mathbf{\Phi}^{n-1}\mathbf{\Gamma})^{-1} p_d(\mathbf{\Phi})$$

where

$$p_d(\mathbf{\Phi}) = \mathbf{\Phi}^n + b_1\mathbf{\Phi}^{n-1} + \cdots + b_{n-1}\mathbf{\Phi} + b_n$$

producing a control input of the form

$$u = -\mathbf{K}\mathbf{x}$$

**Matlab hint:**

- ackerman is applied as  $\mathbf{K}=\text{acker}(\Phi, \Gamma, \mathbf{dp})$ , where  $\mathbf{dp}$  is the list of desired discrete-time closed loop poles  $\lambda_i$ , which usually are obtained from the continuous ones through  $\lambda_i = e^{s_i h}$ .
- For continuous-time design, Ackerman is applied as  $\mathbf{K}=\text{acker}(A, B, \mathbf{cp})$ , where  $\mathbf{cp}$  is the list of desired continuous-time closed-loop poles  $s_i$ .

**Plant model**

Before any controller design, the circuit model must be obtained. To do so, the circuit can be analyzed by nodes, as shown in figure 2.4

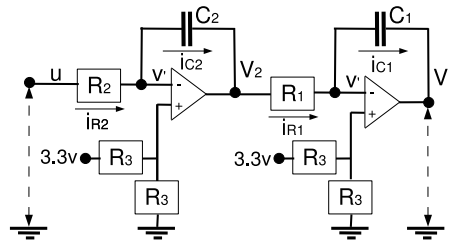


Figure 2.4: Circuit analysis.

By knowing that the control input is  $u$  and the circuit output is  $y = V_1$ , the circuit dynamics is given by

$$u - V' = R_2 i_{R2} \tag{2.1}$$

$$V_2 - V' = R_1 i_{R1} \tag{2.2}$$

$$i_{C2} = C_2 \frac{d(V' - V_2)}{dt} \tag{2.3}$$

$$i_{C1} = C_1 \frac{d(V' - V_1)}{dt} \tag{2.4}$$

$$i_{R2} = i_{C2} \tag{2.5}$$

$$i_{R1} = i_{C1} \tag{2.6}$$

From (2.1), (2.2), (2.3) and (2.4), and considering that without voltage dividers  $V' = 0$ , equations (2.5) and (2.6) become

$$\dot{V}_2 = -\frac{u}{R_2 C_2} \quad (2.7)$$

$$\dot{V}_1 = -\frac{V_2}{R_1 C_1} \quad (2.8)$$

Taking as state variables  $x_1 = V_1$  and  $x_2 = V_2$ , the model is state-space form is

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{R_1 C_1} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{1}{R_2 C_2} \end{bmatrix} u \quad (2.9)$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (2.10)$$

And considering the component values  $R_1 = R_2 = 100K\Omega$  and  $C_1 = C_2 = 420nF$ , the model is

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & -23.8095 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -23.8095 \end{bmatrix} u \quad (2.11)$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (2.12)$$

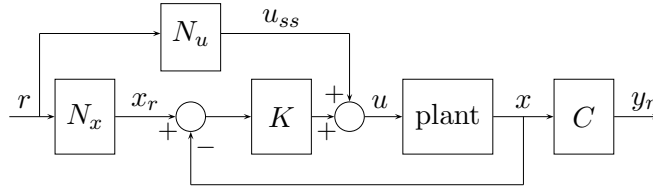
Considering the voltage divider, that is, a constant voltage of 1.65v in the positive input of the operational amplifiers, the model remains the same but the “zero” has changed to 1.65v. Then, with the same model, it must be considered that  $V_2 = x_2 + 1.65v$ , and that the real control input at the board is  $u + 1.65v$ .

### Work to be done

1. **Open loop analysis:** Using Matlab/Simulink, obtain the open loop system response in front of a reference signal in the form of a square wave of amplitude from 1.5V to 2V and frequency 1Hz. And perform a simple stability analysis by pole inspection.
2. **Closed-loop design (continuous-time domain):** Perform state feedback controller design via pole location in the continuous-time domain with tracking structure and assuming that both states can be measured, such that
  - a stable circuit response is achieved

- the control signal values range withing 3.38V and 0V

The first constraint is a control specification while the second one is a hardware limitation. Obtain also the closed-loop response. As a note, the tracking structure is given by



where  $N_u$  is the matrix for the feedforward signal to eliminate steady-state errors,  $N_x$  is the matrix that transforms the reference  $r$  into a reference state,  $K$  is the state feedback gain and  $C$  is the plant output matrix. Recall that  $N_x$  and  $N_u$  can be computed in the continuous-time domain as

$$\begin{pmatrix} \mathbf{N}_x \\ \mathbf{N}_u \end{pmatrix} = \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & 0 \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{0} \\ 1 \end{pmatrix}$$

while in the discrete-time domain are computed as

$$\begin{pmatrix} \mathbf{N}_x \\ \mathbf{N}_u \end{pmatrix} = \begin{pmatrix} \Phi - \mathbf{I} & \Gamma \\ \mathbf{C} & 0 \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{0} \\ 1 \end{pmatrix}$$

3. **Closed-loop design (discrete-time domain):** Carry out the state feedback controller design via pole location in the discrete-time domain with tracking structure and assuming that both states  $x_1$  and  $x_2$  are measured, such that the stable circuit response is achieved while the previous hardware constraint is still met.

In this case, the sampling period and closed-loop pole locations for controller are: period of  $h = 0.05$  s, and the discrete state feedback controller is designed to place the continuous closed loop poles at  $p_{1,2} = -5 \pm 20i$ .

**Observer:** As a note, if only the first state is measured, an observer must be designed. For example, a full predictor observer can be used,



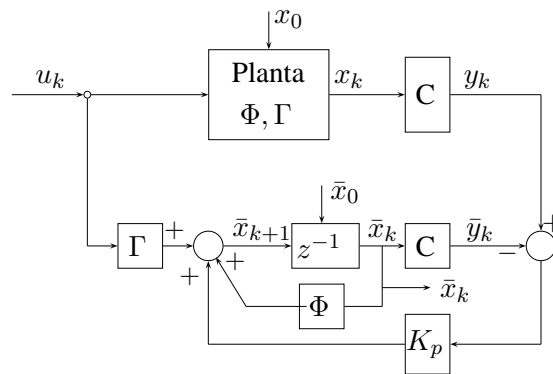
whose equation is

$$\bar{\mathbf{x}}_{k+1} = \Phi \bar{\mathbf{x}}_k + \Gamma \mathbf{u}_k + \mathbf{K}_p (y_k - \mathbf{C} \bar{\mathbf{x}}_k)$$

where  $\bar{\mathbf{x}}_k$  is the predicted state, and  $\mathbf{K}_p$  is the observer gain of the form

$$\mathbf{K}_p = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{pmatrix}$$

which can be computed directly in Matlab by typing `Kp=acker(Phi',C',p)'`. The observer block diagram is



### **2.2.2 Analysis of multitasking embedded control systems (ECS)**

In many high-tech systems, the micro-controller and/or network is used not only for the control computations, but also for interrupt handling, error management, monitoring, etc. And it is known that in a multitasking real-time control systems, jitters, i.e. timing interferences on control tasks due to the concurrent execution of other tasks, deteriorate control loops performance. The objective of this stage is to observe these degrading effects.

#### **Work to be done**

Using Truetime, simulate the multitasking micro-processor based system. In particular, simulate a two tasks systems. One task is controller with a period (and relative deadline) of 0.05 s (as indicated in the previous section) and execution time of 0.01 s, plus a dummy task with a period (and relative deadline) and execution time whose function is to create jitters. Play with different timing constraints for the dummy task in order to generate different jitter pattern on the control task. Observe and discuss the DI closed-loop responses for the different settings of the dummy task.

### 2.2.3 Design of multitasking embedded control systems (ECS)

The jitter problem can be treated using different approaches. One of them is to adopt the *one shot task model*.

The basic idea is to synchronize the operations within each control loop at the actuation instants. In this way, the time elapsed between consecutive actuation instants, named  $t_{k-1}$  and  $t_k$ , is exactly equal to the sampling period,  $h$ . Within this time interval, the system state is sampled, named  $x_{s,k}$ , and the sampling time recorded,  $t_{s,k} \in (t_{k-1}, t_k)$ . The difference between this time and the next actuation time

$$\tau_k = t_k - t_{s,k} \quad (2.13)$$

is used to estimate the state at the actuation instant as

$$\hat{x}_k = \Phi(\tau_k)x_{s,k} + \Gamma(\tau_k)u_{k-1} \quad (2.14)$$

where  $\Phi(t) = e^{At}$  and  $\Gamma(t) = \int_0^t e^{As}dsB$ , being  $A$  and  $B$  the system and input matrices, and  $u_{k-1}$  the previous control signal. Then, making use of  $\hat{x}_k$ , the control command is computed using the original control gain  $K$  as

$$u_k = K\hat{x}_k. \quad (2.15)$$

The control command  $u_k$  is held until the next actuation instant. A control strategy using (2.13)-(2.15) relies on the time reference given by the actuation instants, if  $u_k$  is applied to the plant by hardware interrupts, for example. In addition, samples are not required to be periodic because  $\tau_k$  in (2.13) can vary at each closed-loop operation.

#### Work to be done

Using Truetime, simulate the multitasking real-time system developed in the previous section with the controller implementing the one-shot task model. Hence, the multitasking system will run the control task and the dummy task. And it should be observed that when the control task implements the one-shot task model, the degradation introduced by the jitters caused by the dummy task disappear.

### 2.2.4 Analysis of networked control systems (NCS)

When control loops are closed over a communication network, time delays appear within each loop operation.

#### Basics on control with time delays

Let

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}u(t - \tau) \\ y(t) = \mathbf{C}\mathbf{x}(t) \end{cases}$$

be the continuous time state-space representation of a system with a time delay  $\tau$ . With period  $h$ , with  $\tau \leq h$ , the discrete-time state-space form is

$$\begin{cases} \mathbf{x}_{k+1} = \Phi(h)\mathbf{x}_k + \Phi(h - \tau)\Gamma(\tau)u_{k-1} + \Gamma(h - \tau)u_k \\ y_k = \mathbf{C}\mathbf{x}_k \end{cases} \quad (2.16)$$

where  $\Phi(t)$  and  $\Gamma(t)$  can be obtained, with  $t = h$  or  $t = h - \tau$  or  $t = \tau$ , as before from

$$\begin{aligned} \Phi(t) &= e^{\mathbf{A}t} \\ \Gamma(t) &= \int_0^t e^{\mathbf{A}s}\mathbf{B}ds \end{aligned}$$

The state-space representation of (2.16) is

$$\begin{pmatrix} \mathbf{x}_{k+1} \\ z_{k+1} \end{pmatrix} = \begin{pmatrix} \Phi(h) & \Phi(h - \tau)\Gamma(\tau) \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{x}_k \\ z_k \end{pmatrix} + \begin{pmatrix} \Gamma(h - \tau) \\ 1 \end{pmatrix} u_k \quad (2.17)$$

where  $z_k = u_{k-1}$  represents the previous control signal.

From (2.17), controllability and observability can be analyzed as explained. And control design via pole location can also be done using the ackermans formula, but taking into account that an extra desired discrete time closed-loop pole has to be specified. Usually, this extra pole, in the discrete time domain, is chosen to be 0.

#### Work to be done

Using Truetime, simulate several networked control systems. In particular,

- simulate the NCS for the double integrator system using the settings for period and desired poles as before, with sampling occurring in the sampler node, control algorithm executing in a controller node, and

actuation taking place in the actuator node. As a network, use for example CAN, with different baudrates. Analyze, by observing different responses, how the communication delay affects the performance considering that the controller that applies is the one obtained previously in the multitasking system (dimension 2). Note that in this case, the delay is constant.

- Extend the previous simulation, adding more nodes sending additional traffic to the network and observe the effect of this new traffic on the performance of the double integrator control. Note that in this case, the delay varies.

### 2.2.5 Design of networked control systems (NCS)

The effect of delays can be treated using different approaches.

#### **Work to be done**

Do the same two simulated experiments performed in the previous section but using a controller that accounts for delays, that is, with an extra gain (dimension 3). Remember to place the third discrete-time pole at 0. The application of this new controller is effective for both constant and varying delay scenarios?

### 2.2.6 Analysis and design of event-driven control systems (EDCS)

In networked and embedded control systems, the most classic method used to read the input data of a controller is the *periodic sampling*. However the enforcement of a fixed separation between two consecutive activations can be inappropriate depending on the controller status. In some critical condition the controller may wish to sample more frequently. On the other hand, sometimes the samples could be less dense, allowing to save more computational resources.

The utility in breaking the constraint of periodic sampling is more relevant in severely limited computational resources such as in networked and embedded control systems. In event-driven control, the controller is activated upon some condition on the system status and not periodically. The condition, called *event condition* or *execution rule*, mandates to take a new control action when the measurement signal has deviated sufficiently from the desired set-point.

The plant to be controlled is modeled by the following continuous-time linear control system

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx\end{aligned}\tag{2.18}$$

where  $x \in \mathbb{R}^{n \times 1}$  represents the system state,  $u \in \mathbb{R}^{m \times 1}$  is the input,  $y \in \mathbb{R}$  denotes the system output, and  $A \in \mathbb{R}^{n \times n}$ ,  $B \in \mathbb{R}^{n \times m}$ , and  $C \in \mathbb{R}^{1 \times n}$  describe the evolution of the system. Given the feedback matrix  $K \in \mathbb{R}^{m \times n}$ , let

$$u_i = Kx(a_i) = Kx_i\tag{2.19}$$

be the control updates computed from a sample at time  $a_i$  by a linear feedback controller designed in the continuous-time domain. Notice that we set  $x_i = x(a_i)$  to denote in short the system state at the sampling time  $a_i$ .

In periodic sampling we have  $a_{i+1} = a_i + h$ , where  $h$  is the period of the controller. However in event-driven sampling the activations of the controller occurs at a sequence

$$\{a_i\}_{i \in \mathbb{N}}\tag{2.20}$$

that is not necessarily periodic. Between two consecutive control updates, the control signal is held constant, meaning that

$$\forall i \in \mathbb{N} \quad \forall t \in [a_i, a_{i+1}[ \quad u(t) = u_i.\tag{2.21}$$

The event conditions specify the rule that triggers the execution of controller jobs. Generally speaking, the execution rule ensures that the system

state does not deviate significantly from the desired/expected value, i.e. a given error is tolerated from the sampled state.

Let

$$e_i(t) = x(t) - x_i \quad (2.22)$$

be the error evolution between consecutive samples with  $t \in [a_i, a_{i+1}[$ .

For several types of event-driven control approaches, event conditions can be generalized by introducing a generic function  $f : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$  that measures the magnitude of the error. Once such a function is provided, the condition that must be ensured during the controller life time is

$$f(e_i(t), x_i) \leq \eta \quad (2.23)$$

where  $\eta \in ]0, 1]$  is the error tolerance.

Since we require that the condition of (2.23) holds during the complete lifetime of the controller, it is quite natural to set the next sampling instant  $a_{i+1}$  equal to the first instant when the state  $x_{i+1} = x(a_{i+1})$  reaches the boundary of (2.23). Hence,

$$a_{i+1} = \min\{t \in \mathbb{R} : t \geq a_i, f(x(t) - x_i, x_i) = \eta\} \quad (2.24)$$

The application of event-driven sampling requires dedicated hardware logics to check the system trajectory, and check whether the control job must be activated according to the specific event-condition. However, if the next activation time can be computed at each job execution, the dedicated hardware is no longer required. Jobs will follow a self-triggered model because each job will determine the release of its next job.

Determining the next activation  $a_{i+1}$  from (2.24) requires to find the zeros of a generic function  $f$ . That is, given the sampled state  $x_i$  at the instant  $a_i$ , then  $a_{i+1}$  is given by the smallest value that solves

$$f(x(t) - x_i, x_i) = \eta \quad (2.25)$$

Finding the minimum  $t$  that solves (2.25) is a challenging task because having  $x(t)$  implies that matrix exponential will appear as input arguments of  $f$ , making the equation transcendent, and preventing so to have an analytical solution.

Fortunately, finding a generic approximate solution is possible. For any linear system (2.18)–(2.19) with execution rule given by (2.23), and for any given state  $x_i = x(a_i)$ , the next sampling instant  $a_{i+1}$  is the smallest positive zero of the function on  $t$

$$f(\Psi(t - a_i)(A - BK)x_i, x_i) - \eta = 0, \quad (2.26)$$



where  $\Psi : \mathbb{R} \rightarrow \mathbb{R}^{n \times n}$  is defined as

$$\Psi(t) = \int_0^t e^{As} ds. \quad (2.27)$$

Notice that finding the zeros of (2.26) is still challenging. However, since  $\Psi(t)$  can be efficiently approximated by the following power series

$$\Psi(t) = \sum_{k=1}^{\infty} \frac{A^{k-1} t^k}{k!} \quad (2.28)$$

then we can simplify the problem by finding the smallest zero  $t$  of (2.26), replacing  $\Psi(t)$  by the  $n^{\text{th}}$  order Taylor series approximation.

The previous result gives the first steps toward an explicit approximate solution for finding the next activation time given a generic  $f$ . Now we examine a special cases of  $f$ .

A typical choice for  $f$  is a quadratic function that is defined as follows

$$f(e, x) = \frac{e^T Q_1 e}{x^T Q_2 x}, \quad (2.29)$$

where  $Q_1, Q_2 \in \mathbb{R}^{n \times n}$  are weighting matrices. In this case it is possible to determine an explicit solution of the next sampling instant  $a_{i+1}$  that solves (2.24).

Given the sampled state  $x_i$  at the instant  $a_i$ , when the error is quadratic for the problem of (2.24), then  $a_{i+1}$  is given by the smallest value that solves the following equation

$$\frac{(x(t) - x_i)^T Q_1 (x(t) - x_i)}{x_i^T Q_2 x_i} = \eta \quad (2.30)$$

Fortunately the following results allows to find efficiently this value. For any linear system (2.18)–(2.19) with execution rule of the form (2.29), and for any given state  $x_i$ , the next sampling instant is the smallest positive zero of the function on  $t$

$$[\Psi(t - a_i)(A - BK)x_i]^T Q_1 [\Psi(t - a_i)(A - BK)x_i] - \eta(x_i^T Q_2 x_i) = 0, \quad (2.31)$$

where  $\Psi : \mathbb{R} \rightarrow \mathbb{R}^{n \times n}$  is defined in (2.27).

Note that finding the zeros of (2.31) is still a challenging task. However, by using an  $n^{\text{th}}$ -order approximation of  $\Psi$  in (2.31), equation (2.31) becomes a polynomial on  $t$  of degree  $2n$ , and the  $a_{i+1}$  will be the smallest positive root of the polynomial.

For example, using this procedure, and considering that the event condition is given by

$$[x_{k+1} - x_k]^T M_1 [x_{k+1} - x_k] = \eta x_k^T M_2 x_k \quad (2.32)$$

with user-defined matrices  $M_1$  and  $M_2$ , the next activation time is the positive  $t$  of

$$t = \frac{\sqrt{-4[(A - BK)x_k]^T [(A - BK)x_k] (-\eta)x_k^T x_k}}{2[(A - BK)x_k]^T [(A - BK)x_k]}. \quad (2.33)$$

### Work to be done

- Simulate in matlab the event-driven control of the double integrator plant using the event-condition (2.32) with matrices  $M_1$  and  $M_2$  being the identity,  $\eta = 0.1$ , and  $K = [0.5 \quad -1.35]$ . Carry out the simulation emulating dedicated hardware for detecting the event condition. Use different values for the weighting matrices  $M_i$ , for  $\eta$ , and different control gains  $K$  to observe the system response and the pattern of activation times.
- Perform the same study as before but using the self-triggered approach, that is, using (2.33).

## 2.3 Implementation of NECS

The implementation will cover several phases and will consist in implementing the controller in different scenarios:

- single task system: only the standard controller, using the data obtained in the discrete-time control design in stage 2.2.1.
- two tasks multitasking system: standard controller and dummy task, as in stage 2.2.2.
- two tasks multitasking system: controller and dummy task, with the one-shot task model implemented in the controller, as in stage 2.2.3.
- networked control system with the standard controller subject to delays as in stage 2.2.4
- networked control system with the controller designed from the extended model, as in stage 2.2.5.
- event-driven control, as in stage 2.2.6.

### 2.3.1 Introduction to the development framework

The goal of this work is to know the development framework.

#### Work to be done

1. Execute the program that intermittently activates the orange led. To do so, go through the following instructions
  - Execute RT-Druid (Eclipse)
    - in File – > New – > RT-Druid Oil and C/C++ Project, enable “Create a Project” using one of these templates, and open “pic30”, open “FLEX”, and select “EDF: Periodic task with period 0.5s”. In the first little window, click “Next”, give it a name such as “example”, click “Next” and finally “Finish”.
    - Then it appears in the “project explorer” window the “example” project with the .oil (kernel configuration file) and the .c (code file, where we have the actual program code)

- To compile, it must be remembered that it is always required to modify and save either the .oil or the .c. Then, to compile, go to Project – > Build Project. If everything is correct, it will appear in the terminal “Compilation terminated successfully!”. And in the ”project explorer” a new ”Debug” folder must appear, where the pic30.cof file is located, which is the binary file to be downloaded to the board using MPLAB.
- Execute MPLAB
  - in File – > Import: go to the workspace, and inside of the RT-Druid project just created, and inside of the Debug folder, select the .cof.
  - To download the file to the dsPIC, click on the second yellow icon with a down arrow.
  - To execute the file, click on the icon with an ascending step.
  - To stop it, click on the icon with a descending step.

The led should start blinking.

2. To start working with the *DI* circuit, the first thing to do is the open loop control. This means that it will be necessary to apply a reference signal in the form of a square wave of amplitude from 1V to 2.5V and frequency  $1Hz$  to the plant input. This will be done using the PWM. The plant output can be read using the ADC, and the data is sent to Matlab using RS232 for better displaying the response. To do so, the following steps must be performed:
  - 11sdcDI.zip: this file must be unzipped in ”...\Evidence\examples\pic30”. In the new created folder, ”...\examples\pic30\11sdcDI”, 5 files can be found:
    - template.xml: it’s a meta-data file, that is not really used.
    - RS232\_mfile.m: it’s the Matlab file that will get the data sent from the board through the RS232 port. It works as an ”oscilloscope”.
    - conf.oil: kernel configuration file. It includes details of the kernel configuration as well as the definition of the tasks that the kernel will execute. In particular it defines:
      - \* TASK TaskReferenceChange: it generates the reference signal previously described
      - \* TASK TaskController: it will code the control algorithm

\* TASK Send: it sends data through RS232.

In addition, it defines which *alarms* are associated to each task. Alarms will then be used in the *code.c* to activate the associated tasks (in the *main*). Their definition is as follows:

```
...
ALARM AlarmReferenceChange {
    COUNTER = "myCounter";
    ACTION = ACTIVATETASK { TASK = "TaskReferenceChange"; };
};
ALARM AlarmController {
    COUNTER = "myCounter";
    ACTION = ACTIVATETASK { TASK = "TaskController"; };
};
ALARM AlarmSupervision {
    COUNTER = "myCounter";
    ACTION = ACTIVATETASK { TASK = "TaskSupervision"; };
};
```

If more tasks were to be defined, they should be specified in this configuration file using the same structure.

- setup.h: it includes those libraries required for the circuit control, as well as other definitions.
- code.c: actual program code that includes:

\* int main(void): main program where tasks periodicity is configured in milliseconds (with the last parameter of function *SetRelAlarm*) as follows

```
SetRelAlarm(AlarmReferenceChange, 1000, 1000);
// reference changes every 1000ms=1s

SetRelAlarm(AlarmController, 1000, 1000);
//Controller activates every 1000ms

SetRelAlarm(AlarmSupervision, 1000, 10);
//Data is sent to the PC every 10ms
```

Note that the two timing constraints refer to the initial offset and the period. For example, for the *SetRelAlarm(AlarmSupervision, 1000, 10)*, the TaskSupervision will start executing after 1s (first parameter, 1000), with a periodicity of 10ms (second parameter, 10).

\* TASK(TaskReferenceChange): it generates the reference signal as follows

```
TASK(TaskReferenceChange)
{
    if (r == -0.5)
    {
        r=0.5;
        LATBbits.LATB14 = 1;//Orange led switch off
    }else{
        r=-0.5;
        LATBbits.LATB14 = 0;//Orange led switch on
    }
}
```

\* void Read\_State(void): function that is used every time the plant output/s must be read (it reads the ADCs).

\* TASK(TaskController): tasks that codes the control algorithm. In open loop, it has the following code

```
TASK(TaskController)
{
    /* sampling */
    Read_State(); // returns x[0] and x[1]
    /* control law */
    u=r;
    /* Check for saturation */
    if (u>v_max/2) u=v_max/2;
    if (u<-v_max/2) u=-v_max/2;
    /* PDC1 sets the PWM duty cycle */
    PDC1=(u/v_max)*0x7fff+0x3FFF;
}
```

These *main* and the *task controller* will be the parts requiring more work in future sections, as well as coding new tasks such as a *TaskDummy* to create jitter or thinking on a possible new task, *TaskApplyControl*, to implement the one-shot task model.

To open this project, in RT-Druid, go to File→New→RT-Druid .., and enabling “Create a project..”, inside of “pic30”, a new folder named “11sdcDI” can be found. Inside of this folder the project named *Standard Control* is found. It performs the open loop control.

- Meanwhile, open Matlab (7), and specify "... \Evidence\examples\pic30\11sdcDI" as a working directory. Open the *RS232\_mfile.m* file, which will be use to acquire the data from the board. In this file care must be taken with the serial port "s=serial('COM1');". It has to be specified correctly.

To execute the open loop controller found in "code.c" together with the Matlab file, the following must be done. In RT-Druid, the project must be compiled, the binary transferred to MPLAB, in order to download it to the dsPIC.

After executing both the binary file in the Flex and the matlab file, the following response should be obtained

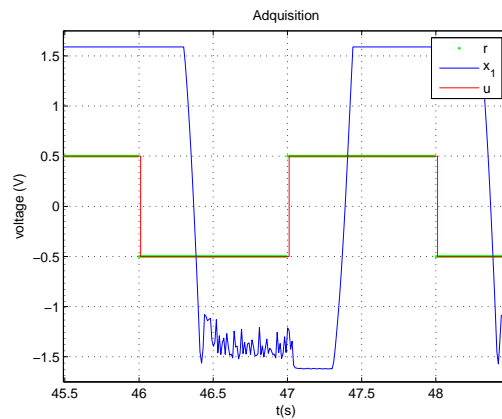


Figure 2.5: Open-loop response.

In this figure, the reference signal ( $r$ , green) and the control signal ( $u$ , red) are the same because of the *TaskController* code that specifies  $u = r$ . The DI response, in blue, is unstable as expected. Note that it saturates.

### 2.3.2 Single control task system

The work to be done is to implement the closed-loop control as designed in in stage 2.2.1. It can be done using the two circuit output variables, or the observer. Recall using the tracking structure. Both the fast and slow controller can be implemented (one after the other, starting with the slow one).

#### Work to be done

To implement the closed-loop control, the following items must be considered:

- The starting code is the “code.c” used in the previous section, to be used also with the previous m-file.
- For implementing the close-loop control, the “code.c” must be modified. Depending on whether the control is performed using the two measured state variables or an observer, the work to be carried out will slightly vary. In any case, it will be necessary
  - To define matrices  $N_x$ ,  $N_u$  and the controller gain  $K$  at the beginning of the program, near the other variables. If observer is used, its gain must be also defined.
  - In the “Read.State” function, only the required variables must be read. If the control is performed using the two state variables,  $x[0]$  and  $x[1]$  must be read. If observer is used,  $x[1]$  can be ignored.
  - In the “TaskController” task, after reading the reference, the control signal  $u$  must be properly calculated. In particular, in the task code shown next, the question marks mark the place where the control algorithm must be coded.

```
* Controller Task */
TASK(TaskController)
{
    /* sampling */
    Read_State(); // returns x[0] and x[1]
    /* Implemetation of the contoller */
    ?????????????? // Observer design
}
```



```

u=????; //Control law including state feedback and tracking
/* Check for saturation */
if (u>v_max/2) u=v_max/2;
if (u<-v_max/2) u=-v_max/2;
/* PDC1 sets the PWM duty cycle */
PDC1=(u/v_max)*0x7fff+0x3FFF; //PDC1 is the register witch sets
                               //the PWM duty cycle
}

```

- In the “main”, note that the controller period is specified in “SetRelAlarm(AlarmController, 1000, 1000)” (which in the actual code 1000 stands for 1000ms). The first 1000 stands for 1000 ms, and it specifies the time when the first execution of the controller will take place. From there, every 1000 ms will execute (second parameter).

After the code is compiled, the m-file in Matlab must be executed together with the binary file in the board. The execution should produce something like

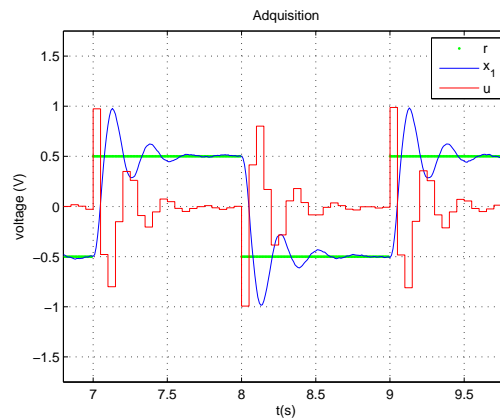


Figure 2.6: Closed-loop response.

### **2.3.3 Two tasks system: standard controller and dummy task**

The objective of this stage is to implement a multitasking system (as in stage 2.2.2) consisting of the standard controller (with a period and relative deadline as specified earlier) and the dummy task (with period and relative deadline as obtained in stage 2.2.2). The goal is to observe the degrading effects of jitters in control performance

#### **Work to be done**

The new implementation requires specifying a dummy task by modifying the kernel *oil* file in terms of defining the new task and the associated alarm. Also, the new task has to be coded. Forcing an artificial execution time can be achieved by a “silly” *for*. The *main* code has to be also modified to configure the new alarm associated to the new task.

### 2.3.4 Two tasks system: one-shot controller and dummy task

The objective of this stage is apply the one-shot task technique for removing the degrading effects of jitters, as in stage 2.2.3.

#### Work to be done

Implementation of the one-shot task model in the controller controller. There are several strategies for implementing this technique. A suggested strategy is to perform the actuation (application of the control signal to the plant) using a task, namely *TaskApplyControl*, different than the controller task (*TaskController*) that computes the actual value for the control signal.

The *TaskApplyControl* can have the same period as the control tasks, for example, 50ms, but an offset of say 45 ms and a deadline of 1ms. The *TaskApplyControl* deadline must be set in the definition of the task in the *conf.oil* while the offset and period are defined in the *main* of *code.c*, when activating the task. For example, if the *TaskController* starts its first execution at 500 ms with a period of 50 ms (see *SetRelAlarm(AlarmController, 500, 50);*), then, the new task should start its first execution at 545 ms, with also a period of 50 ms. This can be done by specifying

```
SetRelAlarm(AlarmApplyControl, 545, 50);
```

if *AlarmApplyControl* is the alarm associated to the task *TaskApplyControl*.

Note that in the *TaskController*, the instruction

```
PDC1=(u/v_max)*0x7fff+0x3FFF; //PDC1 is the register witch sets
//the PWM duty cycle
```

applies the computed control signal  $u$  to the plant. Hence, this instruction would be the one that should be placed in the new *TaskApplyControl*.

The implementation of the one-shot model requires in the *TaskController* obtaining the sampling time ( $t_{s,k}$ ) and the time when the actuation should be applied ( $t_k$ ) in order to compute (2.13). The sampling time can be recorded by checking, each time *ReadState()* is activated, the value of the variable *mytime* (defined as *static unsigned int*), i.e.

```
t_ks=mytime;
```

that is defined and updated every millisecond in the *setup.h* and that keeps the system time in milliseconds. The time of the actuation, if using the *TaskApplyControl*, can be obtained by the following instruction

```
GetAlarm(AlarmApplyControl,&next_act);  
t_k=(int)next_act;
```

where *next\_act* should be defined as *static TickType next\_act=0*, and *t\_k* should be defined as *static unsigned int t\_k=0*. The instruction *GetAlarm* returns in milliseconds the absolute time at which the next job of the task associated to the alarm will be released (and in our case executed).

Finally, the one-shot implementation in the *TaskController*, knowing the time interval from sampling to actuation (2.13), must predict the state at the actuation time (2.14), and compute the control signal with the predicted state (2.15). The latter can be easily done, however, predicting the state at the actuation time requires some thinking.

### 2.3.5 Networked control system

Similar to the work done in in stages 2.2.4 and 2.2.5, the goal of this stage is to successfully implement the networked control of the double integrator system over the CAN network. To start the networked control, the full tower as illustrated in figure 2.7 must be ready.

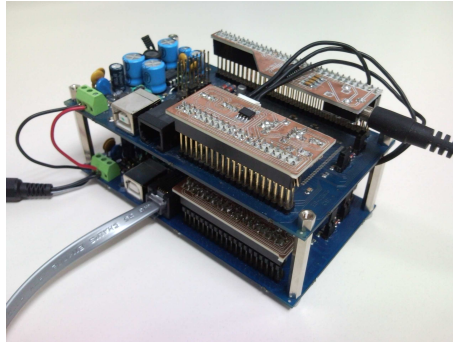


Figure 2.7: Hardware for networked control.

#### Work to be done

1. For the networked control, two projects must be created in RT-Druid. One project will produce the binary file to be downloaded to the bottom board acting as a *controller node* while the other project will produce the binary file for the top board acting as a *sensor/actuator node*.
  - 11sdcNDIc.zip and 11sdcNDIsa.zip: these files must be unzipped in "...\\Evidence\\examples\\pic30". Each file creates a new folder, named *11sdcNDIc* and *11sdcNDIsa*. The first one contains the software for the controller node and the second one for the sensor/actuator node. In the sensor/actuator folder "...\\examples\\pic30\\11sdcNDIsa", 10 files can be found:
    - template.xml: it's a meta-data file, that is not really used.
    - uart\_dma.c and uart\_dma.h: code and definition files for dma operation. These file have to be kept as they are without modifications.
    - e\_can1.c and e\_can1.h: code and definition files for CAN operation. These files must be kept as they are without modifications.

- RS232\_mfile.m: it's the Matlab file that will get the data sent from the board through the RS232 port. It works as an "oscilloscope".
- conf.oil: kernel configuration file. It includes details of the kernel configuration as well as the definition of the tasks that the kernel will execute. In particular it defines:
  - \* TASK TaskSensor: it samples the plant and sends this value via CAN
  - \* TASK TaskActuator: it applies the control signal received through CAN
  - \* TASK Send: it sends data through RS232.

In addition, it defines which *alarms* are associated to each task. Alarms will then be used in the *code.c* to activate the associated tasks (in the *main*). Their definition is as follows:

```

...
ALARM AlarmSensor {
COUNTER = "myCounter";
ACTION = ACTIVATETASK { TASK = "TaskSensor"; };
};
ALARM AlarmActuator {
COUNTER = "myCounter";
ACTION = ACTIVATETASK { TASK = "TaskActuator"; };
};
ALARM AlarmSupervision {
COUNTER = "myCounter";
ACTION = ACTIVATETASK { TASK = "TaskSupervision"; };
};

```

If more tasks were to be defined, they should be specified in this configuration file using the same structure.

- setup.h and setup.c: they include code and definitions of those libraries required for the circuit control, as well as other definitions.
- code.c: actual program code that includes:
  - \* int main(void): main program where tasks periodicity is configured in milliseconds (with the last parameter of function *SetRelAlarm*) as follows
 

```

SetRelAlarm(AlarmSensor,1000,50);
//Sensor activates every 50ms

```

```
SetRelAlarm(AlarmSupervision, 1000, 10);
//Data is sent to the PC every 10ms
```

Note that the *TaskActuator* is activated upon reception of a CAN incoming message.

\* void Read\_State(void): function that is used every time the plant output/s must be read (it reads the ADCs).

\* TASK(TaskSensor): it samples the state and sends this message (see *void Send\_Sensor2Controller\_message(float \*data)* over CAN within a message with identifier PLANT (plus the two states in the data field), where PLANT=1.

```
TASK(TaskSensor)
{
LATBbits.LATB14 ^= 1;//Toggle orange led
Read_State();
Send_Sensor2Controller_message(&x[0]); //identifier=ID_PLANT
}
```

\* TASK(TaskActuator): it is executed upon interruption of a CAN incoming message (see *ISR2(\_C1Interrupt)*) and it applies the control signal to the plant

```
TASK(TaskActuator)
{
u>(*p_u);
PDC1=((*(p_u))/v_max)*0x7fff+0x3FFF;
}
```

In the sensor/actuator folder "...\\examples\\pic30\\11sdcNDIc", 7 files can be found:

- template.xml: it's a meta-data file, that is not really used.
- e\_can1.c and e\_can1.h: code and definition files for CAN operation. These files must be kept as they are without modifications.
- conf.oil: kernel configuration file. It includes details of the kernel configuration as well as the definition of the tasks that the kernel will execute. In particular it defines:
  - \* TASK TaskReferenceChange: it provides the reference signal to be tracked
  - \* TASK TaskController: it applies the control signal received through CAN

In addition, it defines which *alarms* are associated to each task. Alarms will then be used in the *code.c* to activate the associated tasks (in the *main*). Their definition is as follows:

```

...
ALARM AlarmReferenceChange {
COUNTER = "myCounter";
ACTION = ACTIVATETASK { TASK = "TaskReferenceChange"; };
};
ALARM AlarmController {
COUNTER = "myCounter";
ACTION = ACTIVATETASK { TASK = "TaskController"; };
};

```

- setup.h and setup.c: they include code and definitions of those libraries required for the circuit control, as well as other definitions.
- code.c: actual program code that includes:

- \* int main(void): main program where tasks periodicity is configured in milliseconds (with the last parameter of function *SetRelAlarm*) as follows

```
SetRelAlarm(AlarmReferenceChange,1000,1000);
```

Note that the *TaskController* is activated upon reception of a CAN incoming message.

- \* TASK(TaskReferenceChange): it acts as before, but then, the reference is sent over CAN with a message with identifier PLANT+2 and data field containing the reference.

```

TASK(TaskReferenceChange)
{
if (r == -0.5)
{
r=0.5;
LATBbits.LATB14 = 1;//Orange led switch on
}else{
r=-0.5;
LATBbits.LATB14 = 0;//Orange led switch off
}
Send_Controller_ref_message(&r);//identifier=ID_PLANT+2
}

```

This information is only sent for debugging purposes, and it is used in the sensor/actuator node by the *TASK*



(*TaskSupervision*) to output at the right time the reference change over RS232.

- \* TASK(TaskController): it is executed upon interruption of a CAN incoming message (see *ISR2(\_C1Interrupt)*) and it computes the control signal to be sent again over CAN to the S/A node (see *void Send\_Controller2Actuator\_message(float \*data)*)

```
TASK(TaskController)
{
x0=(p_x0); //Get state x[0] from rx_egan1_message1[0]..[3] data fi
x1=(p_x1); //Get state x[1] from rx_egan1_message1[4]..[7] data fi
u=r;
/* Check for saturation */
if (u>v_max/2) u=v_max/2;
if (u<-v_max/2) u=-v_max/2;
Send_Controller2Actuator_message(&u); //identifier=ID_PLANT+1
}
```

The outgoing message as as identifier PLANT+1 and the data field contains the control signal.

To open these projects, in RT-Druid, go to File→New→RT-Druid ..., and enabling “Create a project...”, inside of “pic30”, a new folder named “11sdcNDIa” or “11sdcNDIc” can be found. Inside of these folders the projects named *Networked Control (S/A)* or *Networked Control (A)* are found.

- Meanwhile, open Matlab (7), and specify “... \Evidence\examples\pic30\11sdcNDIa” as a working directory. Open the *RS232\_mfile.m* file, which will be use to acquire the data from the board. In this file care must be taken with the serial port “s=serial('COM1');”. It has to be specified correctly.

Compiling both projects, downloading them to each board, and executing them, in the matlab oscilloscope it should appear the open-loop response, as in Figure 2.5. Recall that all the control data sent over the network is coded as follows, knowing that PLANT=1:

- sensor-to-controller: id PLANT (two states)
- controller-to-actuator: id PLANT+1 (control signal)
- reference update: id PLANT+2 (reference)

- Implementation of the closed loop control. Choose an strategy to implement the networked control. It can be done using the “standard” controller (as in 2.2.4) or using the “extended” controller (as in 2.2.5). In both cases, the closed loop response should be similar to the response shown in Figure 2.6, but probably with more oscillations, like in Figure 2.8. These oscillation are due to the transmission delays of the control data over the CAN network. One way to reduce them is to change the CAN baudrate. This can be done by changing in both files *e\_can1.h* the baudrate:

```
#define BAUDRATE_eCAN1 50000UL
//#define BAUDRATE_eCAN1 100000UL
//#define BAUDRATE_eCAN1 250000UL
//#define BAUDRATE_eCAN1 500000UL
//#define BAUDRATE_eCAN1 1000000UL
```

In both it is set to  $50Kbps$ . If the baudrate increases, the transmission delay will decrease, and less oscillation will appear in the networked closed-loop response.

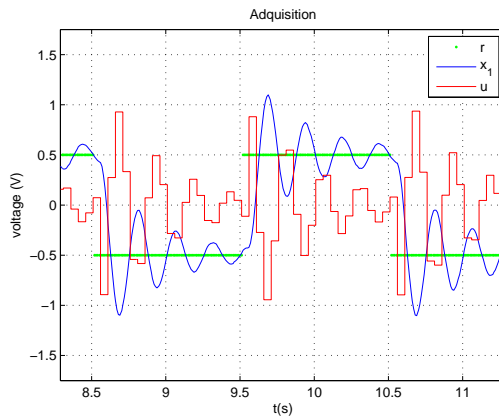


Figure 2.8: Networked closed-loop response.

- In order to study the effect of longer delays introduced by the network traffic, all closed loop systems can be networked in a single CAN network. To do so,
  - change plant identifier (PLANT),

- plug terminator,

and observe that the more loops in the network, the worse are the plant responses of those loops using higher identifiers. Recall that in CAN, the lower the identifier, the higher is the priority that applies in the bit-wise arbitration when collisions occur.

### 2.3.6 Event-driven control

In this stage, an event-driven control strategy is to be implemented in the microprocessor or network based control, as already proposed in 2.2.6.

#### Work to be done

The implementation of an event-driven controller requires re-setting tasks timing constraints at run-time. For example, in the microprocessor based control, this can be done in the *TaskController* as follows:

```
TASK(TaskController)
{
CancelAlarm(AlarmController);
    // it cancels the current period
    r=reference;
    ....
    // computation of the control signal and application
    // via PDC1 as usual
    ....
    Event_time=Calculate_Next_Activation_Time();
    //This function should estimate when the current state will
    //impact on the boundary
    SetRelAlarm(AlarmController, Event_time, 0);
    // it sets as a new release time the Event_time
}
```

where *EE\_UINT16 Event\_time* is in milliseconds. Note that computing the function *Calculate\_Next\_Activation\_Time()* is not straightforward.

1. Implement the event-driven control.
2. Play with different boundaries, etc, to see the impact on the closed loop response.
3. Modify the *TASK (TaskSupervision)* to be able to plot in matlab the sequence of sampling intervals generated by the event-driven strategy. This may require modifying the *RS232\_mfile.m* as well as other parts of the Erika code.

## Chapter 3

# User manual: live CD

The Live CD for this course is designed to be executed without needing to prepare a compilation and assembly environment.

The environment is ready so you can start the Live CD from the boot of the computer and you can work on it.

It is true that the speed of the environment executing into CD may be slower than installed system, but this gives us a portability and speed in preparing the environment.

However the same Live CD has the option to install in the computer, sharing it with another operating system if necessary and selecting which one want to start.

Also there is some guides of using the Live CD and the program DC-SMonitor in video format in the website of the project "Design of a platform to test distributed control systems" (<http://code.google.com/p/pfc-platform-test/><http://code.google.com/p/pfc-platform-test/>) in the video section (<http://code.google.com/p/pfc-platform-test/wiki/VideoExamples><http://code.google.com/p/pfc-platform-test/wiki/VideoExamples>).

### 3.1 Minimum requirements

There are the minimum requirements to start our laboratory Live CD.

- Processor: Intel x86 or compatible, with  $\geq 200$ MHz
- RAM memory:  $\geq 256$  MB.
- DVD reader unit.
- BIOS: it needs to have the option of start in the DVD unit.

- Graphical card: standard, compatible with SVGA.

## 3.2 Booting the live CD

First of all we need to configure the BIOS to boot into the CD. In most cases it is possible to modify pressing the key *F2* or *Sup* several times when the system is starting (in some computers can be other keys). Once we are in the BIOS we need to search the section *Boot* and then select in the first place the CD reader unit, and let the hard disc to second position.

Once we have configured this option, we only have to introduce the live CD to the reader, save the changes and reboot the computer.

When the LiveCD starts it prompts the next text (figure 3.1), we need to wait a little since the next menu appear.

```
ISOLINUX 3.63 Debian-2008-07-15 Copyright (C) 1994-2008 H. Peter Anvin
boot: _
```

Figure 3.1: Live CD booting.

In this menu we can start directly the live CD (first option) or install it into the hard disk (third option)

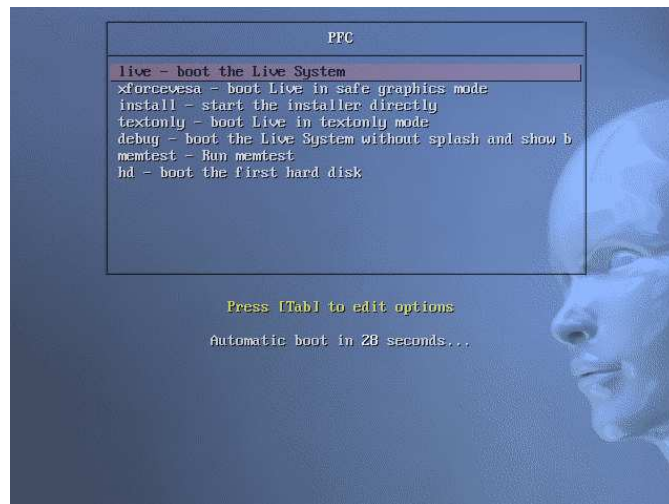


Figure 3.2: Live CD menu.

If you want to boot from the hard to follow step by step guide section 3.3.

User	student
Password	student

Table 3.1: Live CD user and password

MPLAB X IDE beta7.12	link to MplabX program, we will use this to program FLEX board
Eclipse	link to Eclipse program, we will use this to compile the RTOS Erika codes for the FLEX board
Programs	a folder with all source code of the programs used in this laboratory.
Guides	a series of guides in how to use this laboratory.

Table 3.2: Desktop files

If you want to install the system on your computer select the third option and follow the steps in the Installing section.

### 3.3 Step by step Live CD guide

When we choose the boot option we will see the Ubuntu loading logo:



Figure 3.3: Ubuntu loading logo.

At last we will see a user and password login.

#### 3.3.1 Live CD structure

Once we are logged in the system, we will see an Ubuntu desktop (figure 3.5), with the next files:

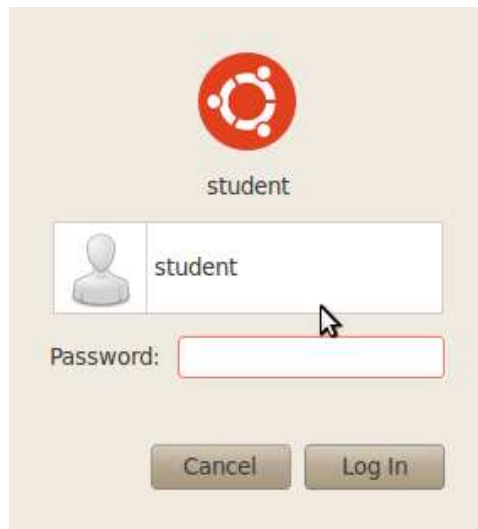


Figure 3.4: Live CD login.

Now we can enter to the directory `/home/student/workspace/Programs/` and we will see all necessary codes compressed in zip format (figure ??).

We must decompress the sources that we will need to be compiled and programmed.

### 3.3.2 Compiling DSPIC\_SENSOR with Eclipse

Once we know the structure of all important folders, we may open the Eclipse program. The first time we open this program, it will ask us which folder we want to be the working directory. In our case we will let this by default (check the square if you don't want to do this every time you open Eclipse) (figure 3.7).

In the main of Eclipse program we can create a new RTDruid project, to do that go to *File* → *New* → *NewProject*, and select *Evidence* → *RT – DruidOilandC/C++Project* (figure 3.8).

Next uncheck the square *Use default location*, and we must go to search the Sensor code in this folder : `/home/student/workspace/Programs/DSPIC_SENSOR/trunk` An then lets put a name to the project (figure 3.9).

Once we have the new project we can compile it clicking in *Project* → *Clean*, with this action we will clean the foulder firstly, and check the option *Start build immediately* to compile the code before we clean the folder (figure 3.10).



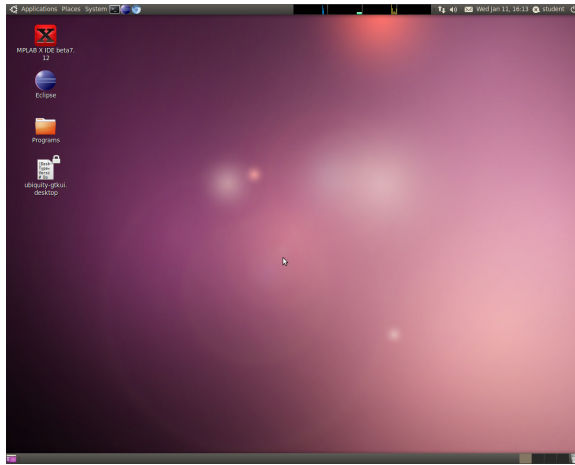


Figure 3.5: Live CD desktop.

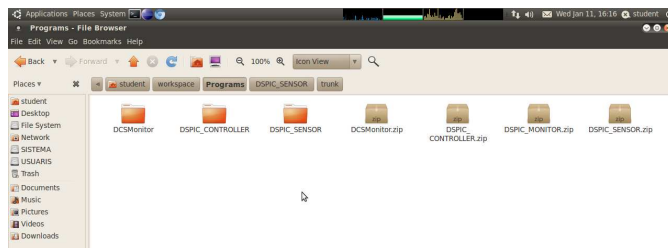


Figure 3.6: Laboratory programs compressed code.

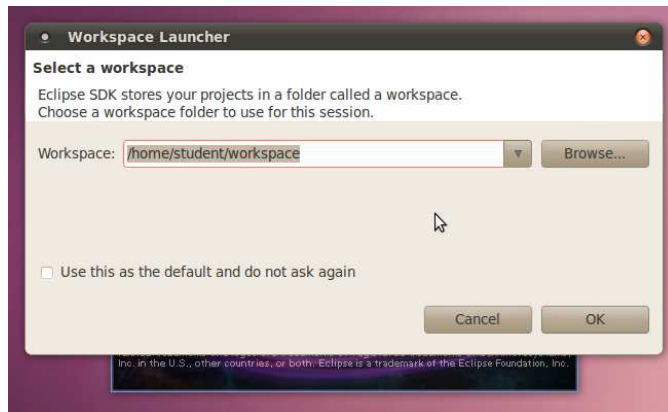


Figure 3.7: Eclipse working directory.

DCSMonitor	This is the program that runs on your computer to see the control. You need to communicate via RS232 with a FLEX board with program dsPIC_SENSOR installed.
DSPIC_SENSOR	This program is for one FLEX board and it will modify the values of the double integrator and read his output value. The device programmed with this code is named Sensor/Actuator. Because it reads the output value of the integrator (this is the task of a Sensor) and writes the input value to the DI (this is a task of a Actuator).
DSPIC_CONTROLLER	This program is for one FLEX board and calculates the control value for the DI and send it to the actuator using the CAN bus.
DSPIC_MONITOR	This program is for another FLEX board, and it can monitory all the CAN bus.

Table 3.3: Files in  $\sim$  /workspace/Programs/

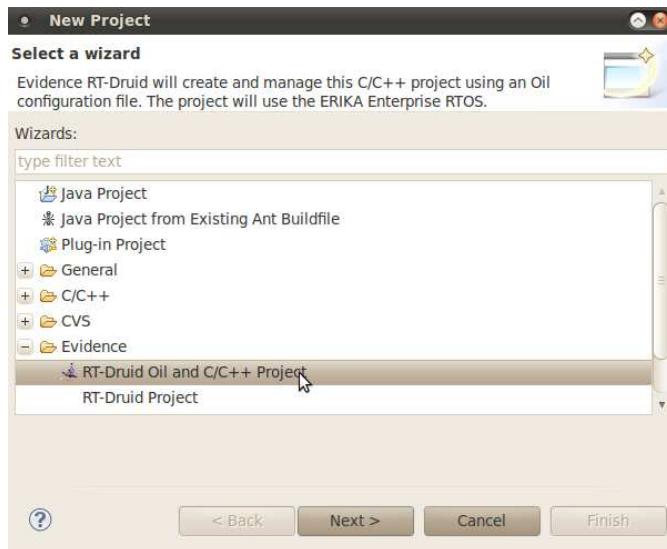


Figure 3.8: Creating a new RT-Druid project in Eclipse.

It will starts to compile and before this in the text box at bottom of the

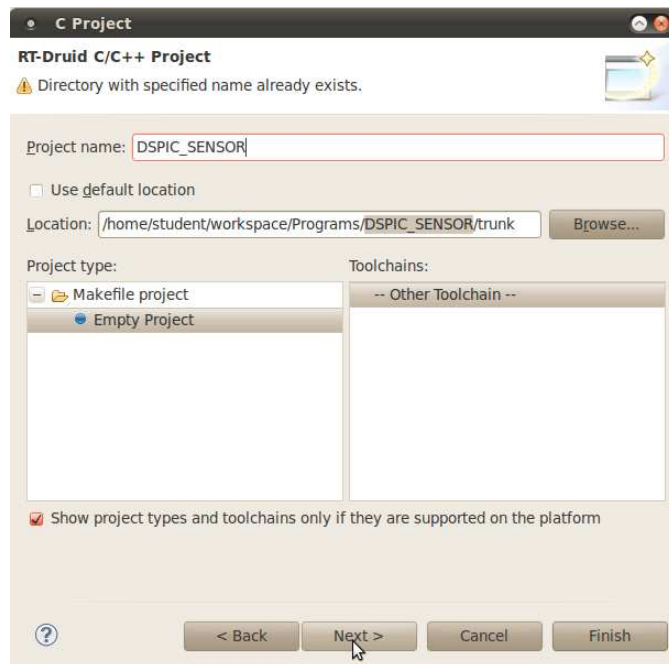


Figure 3.9: Selecting project folder in Eclipse.

program we will see the message:

*Compilation terminated successfully!*

### 3.3.3 Programming DSPIC\_SENSOR with MplabX

Once we compiled the program, let's open MplabX program (figure 3.11)

We will create a new project importing an .elf file generated by Eclipse:

*File* → *ImportHex(Prebuilt)Project*

Then we must search the file generated with Eclipse (figure 3.12):

*/home/student/workspace/Programs/DSPIC\_SENSOR/trunk/Debug/pic30.elf*

Next step is to select the device to be programmed (figure 3.13), in our case is :

- Family: DSPIC33
- Device: dsPIC33FJ256MC710

Now is time to select which programmer we want to use, in our case we use the mplab ICD3 programmer to program the FLEX boards (figure

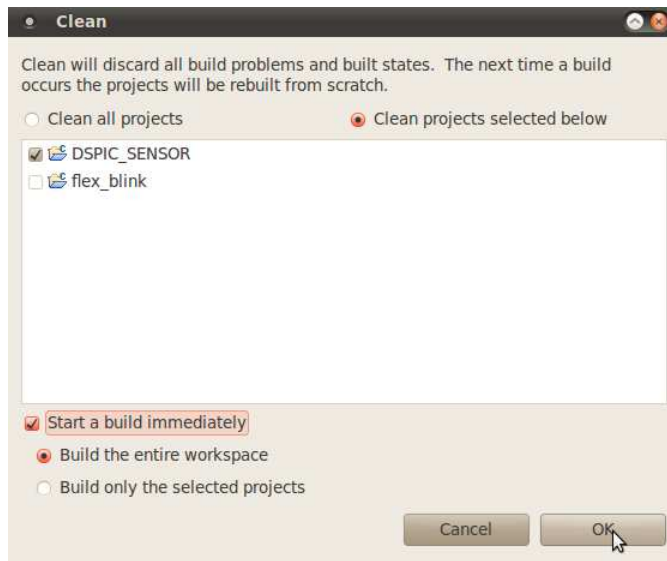


Figure 3.10: Cleaning and compiling in Eclipse.



Figure 3.11: MplabX loading logo.

3.15). In this moment we must connect the programmer mplab ICD3 to the FLEX board to be programmed (figure 3.14):

In case of select another programmer we need to know what the colour means, look into the next table 4.1.

At last we only have to select a name for this project (figure 3.16), and *it is very important to change the directory out of Debug* (figure 3.17). If



Figure 3.12: Selecting pre-compiled file in MplabX.



Figure 3.13: Selecting device in MplabX.

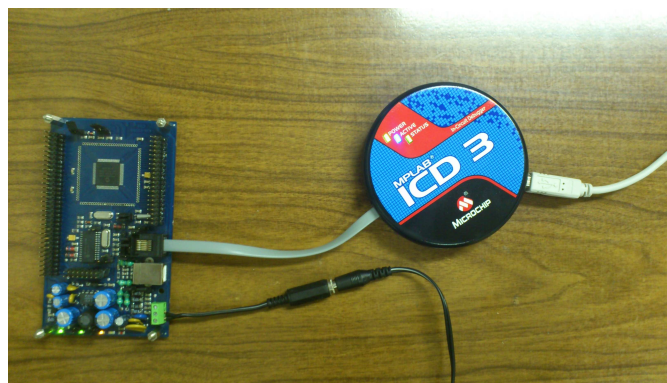


Figure 3.14: Mplab ICD3 and FLEX board connected.

we leave this by default when we modify the source code with Eclipse and make a *Clean* we would have problems with this project in MplabX.

Finally MplabX will show up all details of this new project, so we accept and would have all the environment prepared to program the microcontroller. To program the FLEX board, we must click at button *Make and Program Device* up to the right , once MplabX programmed the device will show this message (figure 3.18).

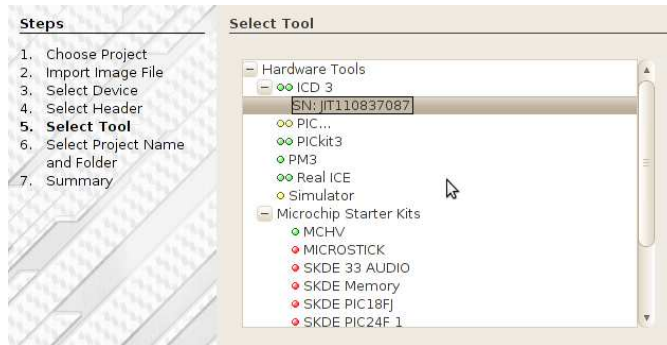


Figure 3.15: Choosing programmer in MplabX.

Green	This colour means that the programmer is fully functional in this version. .
Yellow	This colour means that the programmer is partially functional in this version. In this case it is possible to select, but it may not work.
Red	This colour means that the programmer is not functional in this version. In this case is not possible to select it.

Table 3.4: Programmers compatibility in MplabX

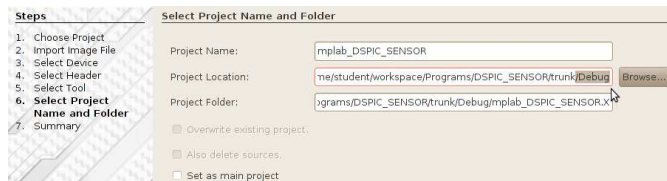


Figure 3.16: Selecting the name of the project in MplabX.

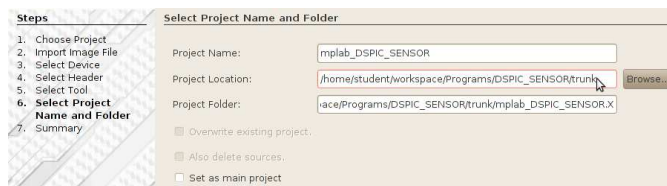
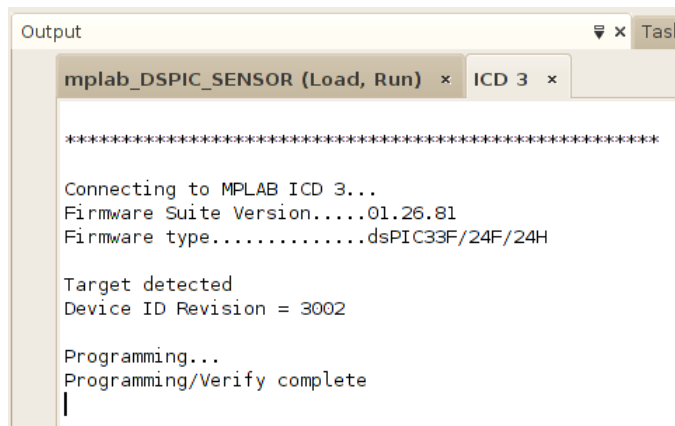


Figure 3.17: Location of the project in MplabX.

### 3.3.4 Summary to program the other devices

The steps to compile and program the other devices (with DSPIC\_CONTROLLER and/or DSPIC\_MONITOR) are the same that we did to program the DSPIC\_SENSOR.



```
Output
mplab_DSPIC_SENSOR (Load, Run) x ICD 3 x
*****
Connecting to MPLAB ICD 3...
Firmware Suite Version.....01.26.81
Firmware type.....dsPIC33F/24F/24H

Target detected
Device ID Revision = 3002

Programming...
Programming/Verify complete
|
```

Figure 3.18: Microcontroller programmed correctly in MplabX.

This section is a summary of all steps to program any device:

1. Decompress codes needed from */home/student/workspace/Programs/*
2. Open the Eclipse program
  - (a) Create a new RT-Druid project
  - (b) Select the folder where the project is, and we must put a name to the project.
  - (c) We must to create the project
  - (d) Clean and compile
3. Open MplabX program
  - (a) Create a new project with the pre-compiled code generated by Eclipse
  - (b) Search the file .elf (in the project folder into Debug)
  - (c) We must select the microcontroller DSPIC33 and dsPIC33FJ256MC710
  - (d) Connect the ICD3 to the computer and the FLEX board
  - (e) Select the ICD3 programmer serial number
  - (f) Put a name to the project and **delete the Debug part of the location**
  - (g) Accept and program

### 3.3.5 DCSMonitor

DCSMonitor is a program used to see the control being executed in one plant. To execute this program you must go to desktop and click in DCSMonitor (figure 3.19).

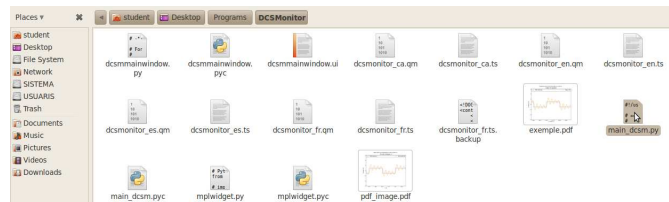


Figure 3.19: Directory with DCSMonitor code.

Once we opened the program we will see this window 3.20. In this program we can view our control in a graph at middle left of the window. To the right there is a button to connect to serial port. Just below if we are connected to a Monitor device we will see all plants in the CAN bus. And below of this you can select which plots you want to see. Then there is a button to start receiving data of a plant, and another to clean the text. Finally we can see statistical values.

Firstly we need to specify in preferences which device we want to connect, and we can choose the language (figure 3.21).

Once we have selected the language and which device we want to connect we would push connect button, if it is fine we would see connected in green color 3.22.

Then if we are connected to a Monitor device go to section 3.3.5.

Otherwise if you are connected to a Sensor/Actuator device go to section 3.3.5.

#### Connected to Sensor/Actuator device

In this execution mode is only enabled monitoring option, clean text and graphic export, so you may start clicking monitor button, and if you have the Sensor/Actuator connected to serial port we would see the real time graph. And in the text box we would see all received data. Like the figure 3.23.

If we want to export the graph firstly we need to stop the monitoring (see the section 3.3.5).



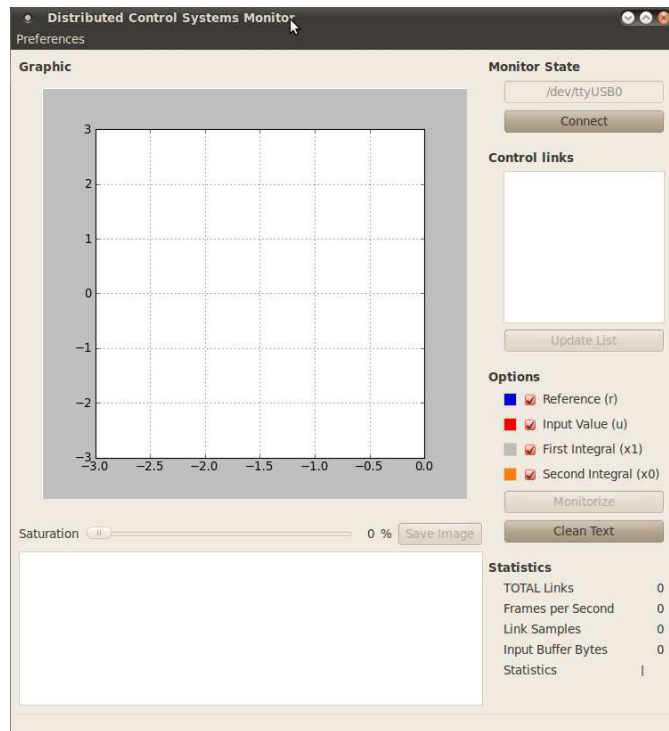


Figure 3.20: DCSMonitor main window.



Figure 3.21: Preferences.

### Connected to Monitor device

In this execution mode we have all the options enabled. Firstly we must click the button *List devices* and then we would see a list with all id plants connected to CAN bus (there is an example 3.24).

Once we had a list of plants, it is recommended to stop updating devices list, because computer and Monitor device will work faster.



Figure 3.22: DCSMonitor connecting to serial port.

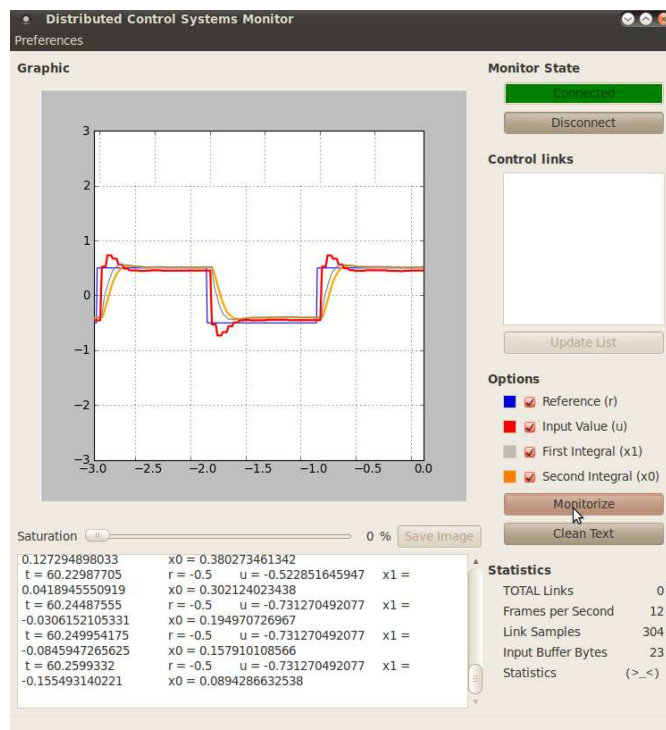


Figure 3.23: DCSMonitor program connected to a Sensor/Actuator device when we are monitoring we would see a real time graphic and data in the text box.

After that we must select any id plant of the list and click in Monitorize button to start receiving his values. Then we would see a real time graphic and their values in text box below (figure 3.25). We can enable or disable



Figure 3.24: Requesting for id plants in the CAN bus to DCSMonitor device.

any of the plots checking it in the options section at the right (figure 3.26).

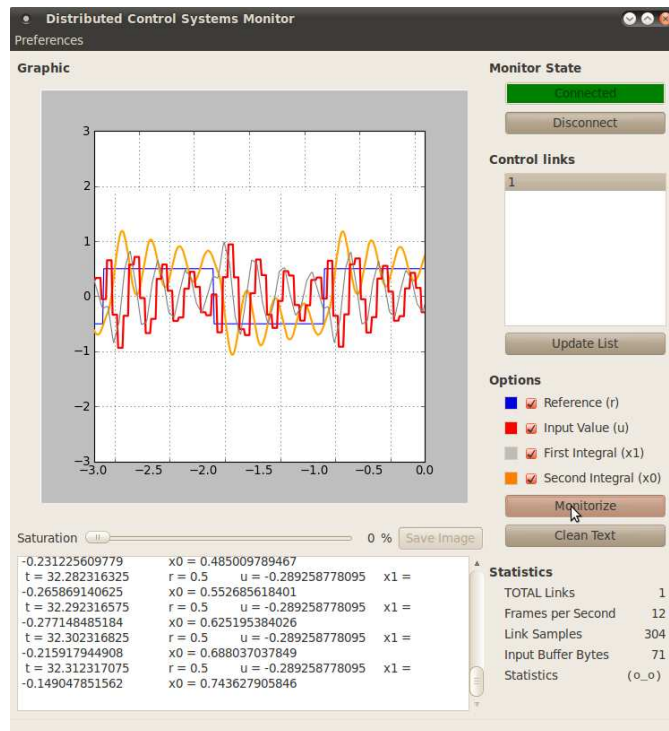


Figure 3.25: All plots in the graphic.

There is an other option in this execution mode, we can send a message to the Monitor device to start sending messages into CAN bus, it is possible dragging the Saturation bar (figure 3.27) since the wanted value. This will

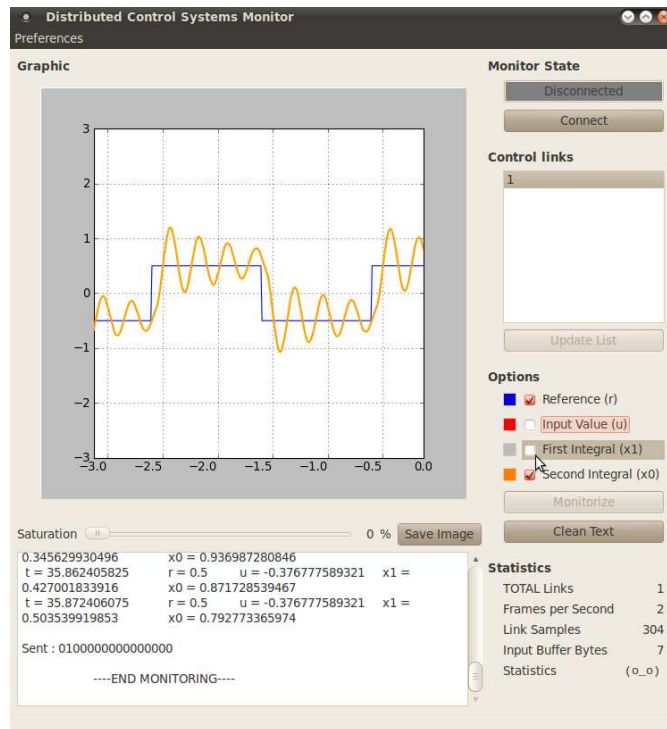


Figure 3.26: Only reference and second integral in the graph.

provoke monitor device to send maximal priorit messages into the CAN bus.

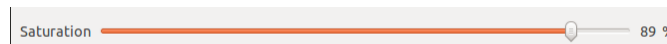


Figure 3.27: Saturation bar.

### Graph exporting

In all execution modes we can take an instance of the graph. To do that we must stop the monitoring in the wanted instance, then check the plots to be showed in the Options panel, and finally push the button Export graph figure 3.28). You only need to select the folder and assign a name to the file, with preferred extension (.pdf, .eps, .ps, .png, and more) (there is an example exporting an image to the desktop in figure 3.29).

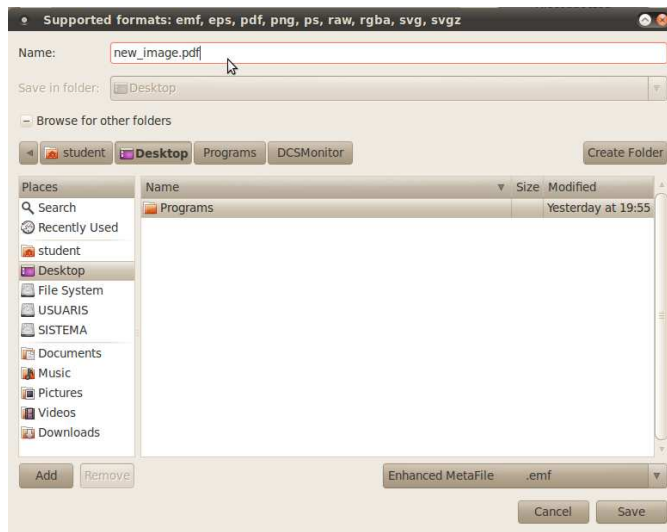


Figure 3.28: Choosing a name for the exported graph.

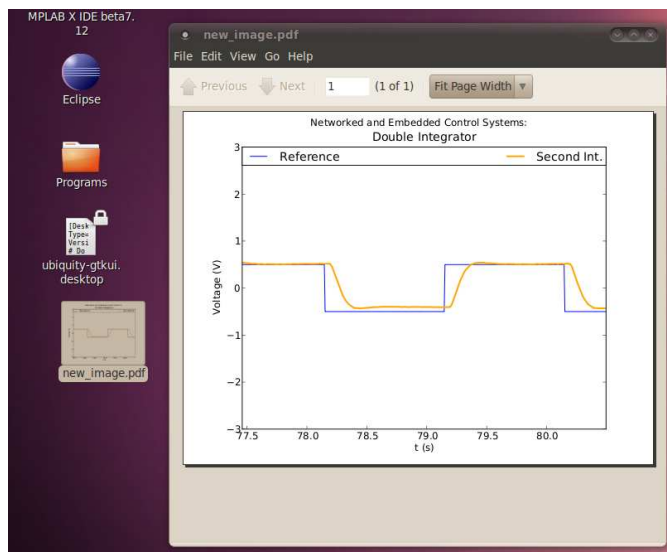


Figure 3.29: Example of exported graphic.

### 3.4 Installing

It is possible to install this Live CD into the computer side by side with another Operating System or alone. You must know that installing an op-

erating system may erase your important data, so you must make this at your own risk.

There is also the option to create a virtual machine (with VirtualBox or VMWare program) and thus work from our current operating system, and running a virtual machine with the laboratory.

Then you can see a step by step guide on how to install the live CD.

In the Live CD main menu (figure 3.30), select the option *Install*.

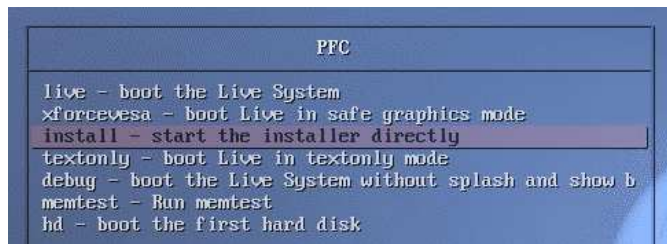


Figure 3.30: Live CD main menu, installing on computer.

Once we selected install we will see Ubuntu logo (figure 3.31) and then the CD will prepare the installation environment.



Figure 3.31: Ubuntu loading log.

Once the environment is ready, then we may choose the language, and after that click on next.

Then select your time zone to synchronise the computers time. And after that select your keyboard distribution.

At this point we need to choose where install the Live CD (figure 3.32).

The options are:

- Install LiveCD next to other operating system (recommended option)
- Install LiveCD in the entire disc.
- Install LiveCD partitioning the disc manually (advanced).

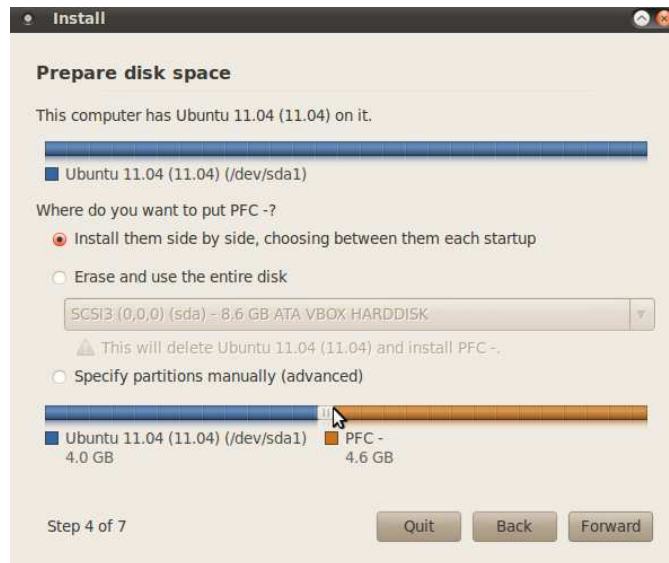


Figure 3.32: Hard disc partitioning.

If we don't know which option to choose, we must select the first option.

Once we finish this step, we would see an advertisement indicating the disc modifications. This step is not reversible, then accept in our own risk.

Now we must wait until it finishes partitioning. Once the disc is parted we must select a user and password (figure 3.33). This LiveCD was created to be the user and password *student* then you need to specify this string.

Once we specified all the options, we would see a summary. When we accept this it will start installing the system (this may take some time depending on the machine).

When the bar gets to 100% we will see an advertisement of completed installation, then click on *reboot now*, extract the disc and push enter.

When the computer starts we must select the new operating system Ubuntu. And then introduce the user and password *student* (figure 3.34).

Now we will have the environment ready to try the different device programs (LiveCD desktop 3.35).

If the LiveCD is installed in a VirtualBox we can install the package *Guest Additions*. This package will give us a lot of facilities to interact with the virtual machine.

To install this package you must start the virtual machine, once you will in the desktop select the option in the window *Devices* -> *Install Guest Additions*, this action will create a new virtual disc in your virtual desktop, then you



Figure 3.33: User and password configuration: we must specify user and password *student*.

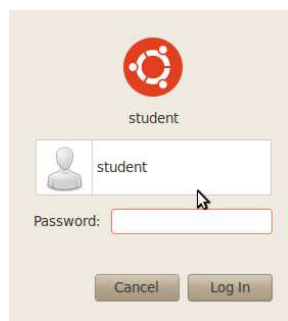


Figure 3.34: Login.

must open it and follow his instructions.

There is a known problem if your host is an Ubuntu machine, after you installed the Guest Additions it is possible that it do not work properly. Then in the virtual machine you must do this steps in a terminal:

1. `sudo apt-get update`
2. `sudo apt-get install build-essential linux-headers- $\$uname -r$`



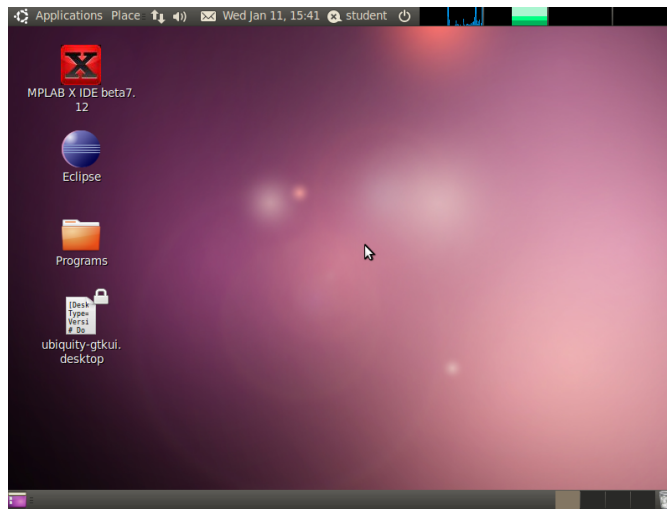


Figure 3.35: Live CD desktop in new installation

3. `sudo apt-get install virtualbox-ose-guest-x11`

# Chapter 4

## Quick guide to start working

### 4.1 Development Environment

To program the Flex boards for conducting laboratory you need to have development environment ready with the following programs:

- Eclipse (EE\_160) : programming environment, renamed Erika Enterprise (EE) RTDruid.
- Mplab (mplabx\_ide\_beta\_7) : program to burn binaries to dsPIC.
- Matlab (matlab 7.8) : program to establish communication between computer and dsPIC.
- Python (Python 2.7) : python shell.
  - PySerial : Serial communication package.
  - Matplotlib : Package to draw graphic plots.

All this programs are free of charge (Matlab is an exception), and all of this can be downloaded in his official web-pages:

- Eclipse  
<http://erika.tuxfamily.org/erika-for-multiple-devices.html>
- Mplabx  
[http://ww1.microchip.com/downloads/mplab/X\\_Beta/installer.html](http://ww1.microchip.com/downloads/mplab/X_Beta/installer.html)

- Python  
<http://www.python.org/download/>
- PySerial  
<http://pypi.python.org/pypi/pyserial>
- Matplotlib  
<http://matplotlib.sourceforge.net/users/installing.html>
- PyQt4  
<http://www.riverbankcomputing.co.uk/software/pyqt/download>

**Python packages** `sudo apt-get install gnuplot` `sudo apt-get install python-matplotlib` `sudo apt-get install python-scitools` `sudo apt-get install python-qt4`

#### 4.1.1 Mplab installation

MplabX needs Java to be run, in most cases (in our case running Ubuntu) this is pre-installed, but in other cases it is necessary to download and install from his official web-page:

<http://www.java.com/es/>

If you need to install it in Ubuntu, you can do it directly from terminal with the next commands:

```
sudo apt-get update
sudo apt-get install default-jre
```

Once Java is installed, go to MplabX web-page (figure 4.1) and select :

- MPLAB IDE X Beta
- MPLAB C30 Lite Compiler for dsPIC DSCs and PIC24 MCUs

[http://ww1.microchip.com/downloads/mplab/X\\_Beta/installer.html](http://ww1.microchip.com/downloads/mplab/X_Beta/installer.html)

Once downloaded, open a terminal to install them (Figura 4.2):

Access the directory where the files have been downloaded and verify that indeed have been downloaded:

```
cd ~/Downloads/
ls
```

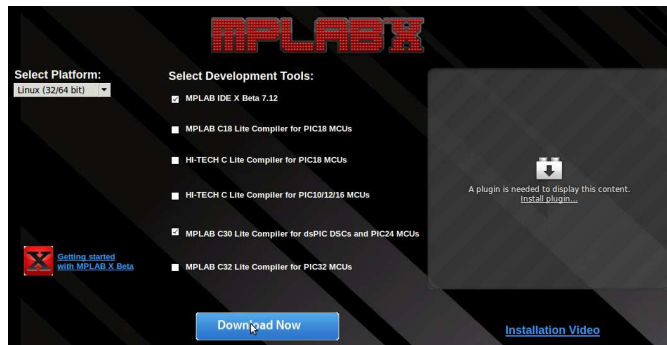


Figure 4.1: MplabX Official page. In this page you can download last version of this program.

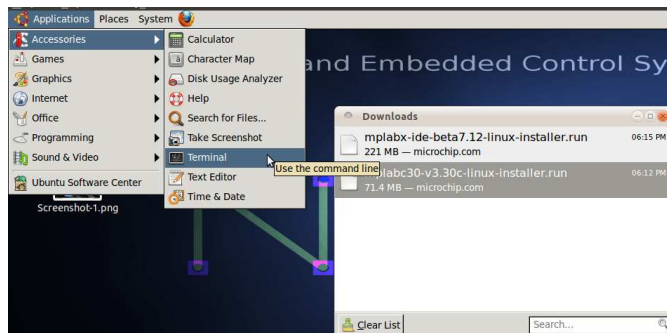


Figure 4.2: Opening a terminal: to open a terminal go to Applications -> Accessories -> Terminal.

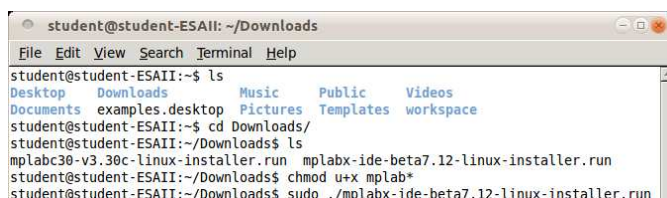


Figure 4.3: Installing MplabX IDE

We give permission to run mplabx-ide and install MplabX IDE (Figura 4.3):

```

chmod u+x mplabx-ide-beta7.12-linux-installer.run
sudo ./mplabx-ide-beta7.12-linux-installer.run

```

Then we'll just accept all the conditions of use (Figura 4.4) and specify the installation directory (Figura 4.5), we can let this by default to `‘/opt/microchip/mplabx‘`.

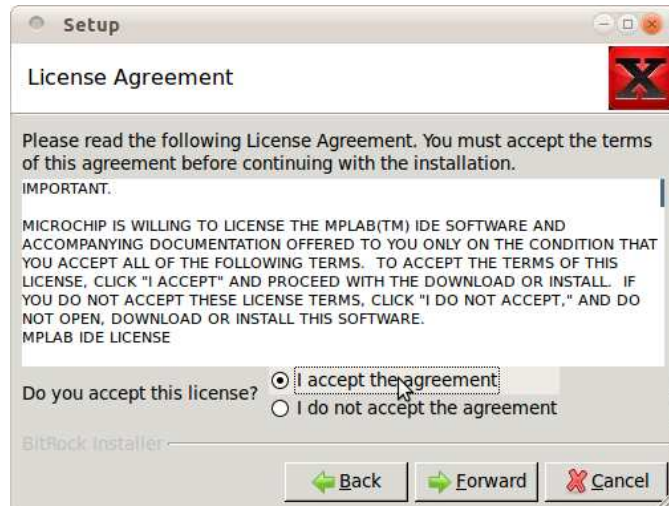


Figure 4.4: Use conditions on MplabX IDE: we need to accept this terms and conditions of MplabX

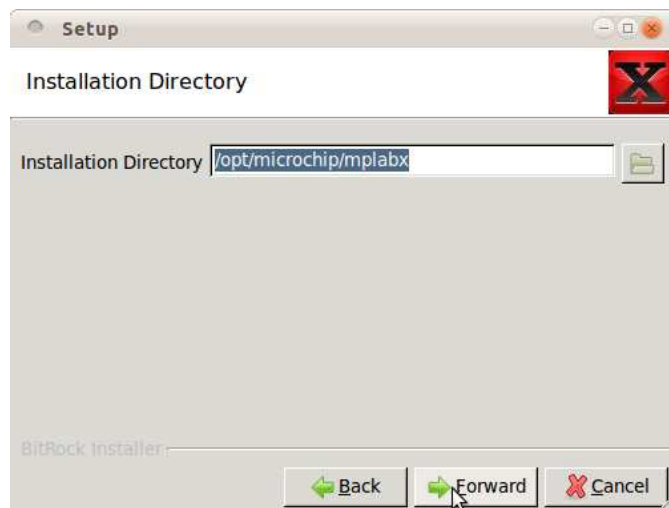


Figure 4.5: MplabX installation directory: we can let this by default

After that we need to reboot (Figura 4.6), but firstly install the other

package.

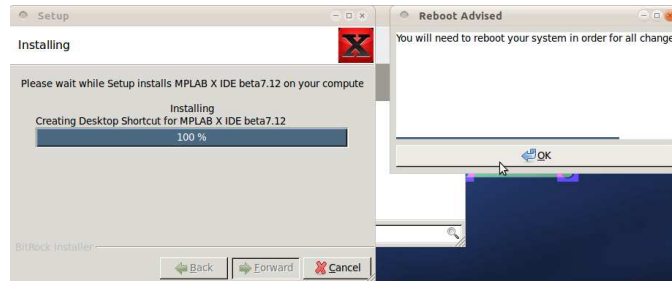


Figure 4.6: MplabX IDE installations end: it says that we need to reboot.

Once we have installed MplabX IDE, lets go to install the other package:

```
chmod u+x mplabc30-v3.30c-linux-installer.run  
sudo ./mplabc30-v3.30c-linux-installer.run
```

Likely the other installation we need to accept terms and conditions (Figura 4.7) and let installation directory by default to ‘/opt/microchip/mplabc30/v3.30c’

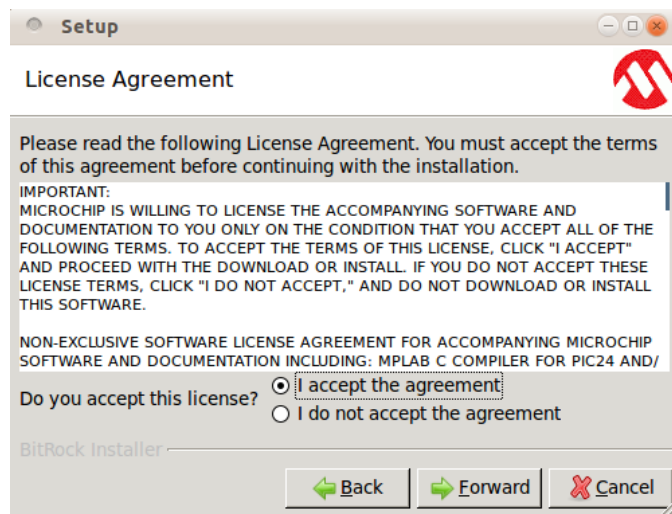


Figure 4.7: MplabX C30 compiler’s terms: we accept this terms.

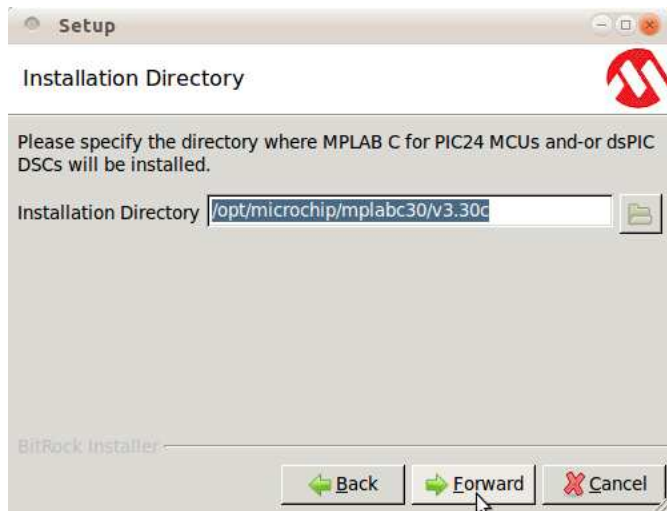


Figure 4.8: MplabX C30 compiler's installation directory: it is recommended to let by default.

### 4.1.2 Eclipse installation

Here we are going to install Eclipse following the steps of his web-page: <http://www.eclipse.org/>

There is in Linux repositories. Lets do this in a terminal:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install eclipse
```

### 4.1.3 Installing RTDruid plugin on Eclipse

Once we installed Eclipse, we need this plugin to program the RTOS Erika. Lets do this steps:

Go to:

- Help -> Install new Software...

We need to add a new entry (Figura 4.10) clicking in:

- Add...

In next window (Figura 4.11) fill entries with next data:

- Name ->RTDruid

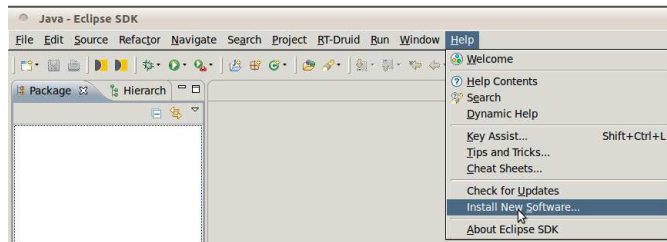


Figure 4.9: Installing Eclipse's plugin: to install new plugins in Eclipse go to Help -> Install new Software...

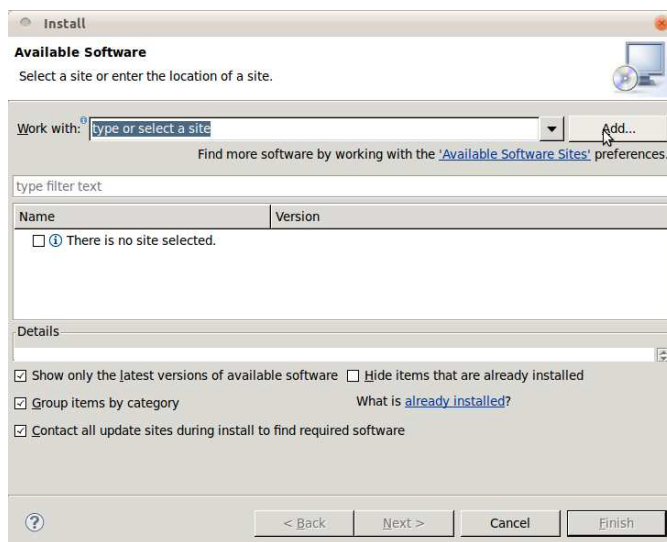


Figure 4.10: Window to add new plugins: in this window we can add new Eclipse's plugins.

- Location -> [http://download.tuxfamily.org/erika/webdownload/rtdruid\\_160\\_nb/](http://download.tuxfamily.org/erika/webdownload/rtdruid_160_nb/)

Then it must be to appear the packages that there is in this url (Figura 4.12), we need to check all of them.

Then will appear a summary of the packages to be installed, lets accept terms and conditions (Figura 4.13)

Be aware on installation because there is a moment that will popup a hidden advisement about the trust of the package, and we need to accept it (Figure 4.14).

Finally we must to reboot the computer for the changes to take effect.



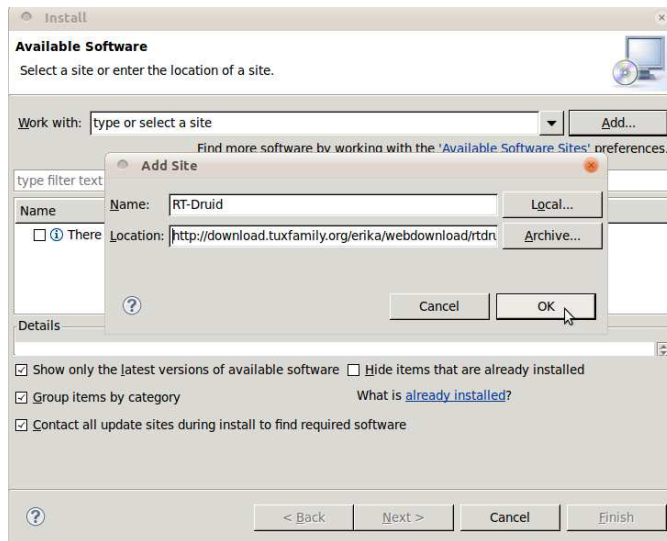


Figure 4.11: Adding new plugin's url: here we can add urls to Eclipse's plugins.

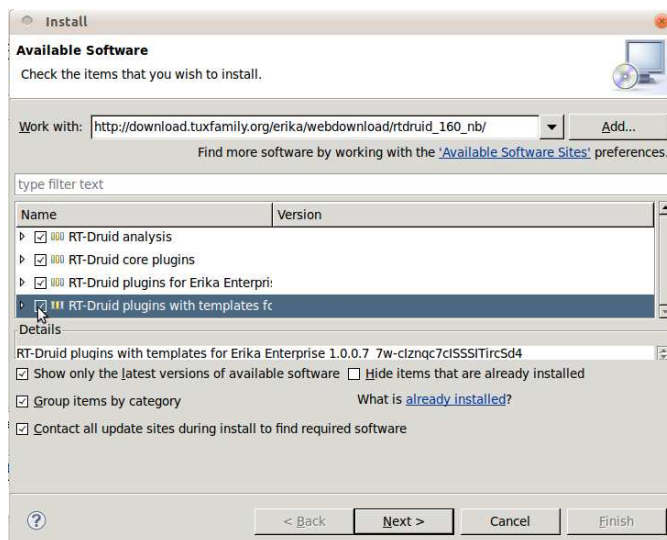


Figure 4.12: Available plugins: there is a list of plugins available in the url.

Then reboot it (Figure 4.15).

It is necessary to specify the path of the compiler to RT-Druid to compile and generate .elf files. Then specify this going to:

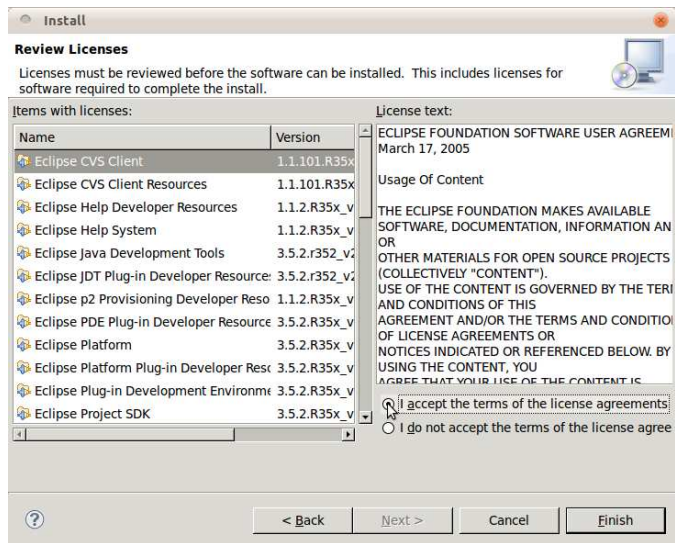


Figure 4.13: RTDruid terms and conditions: we need to accept this.

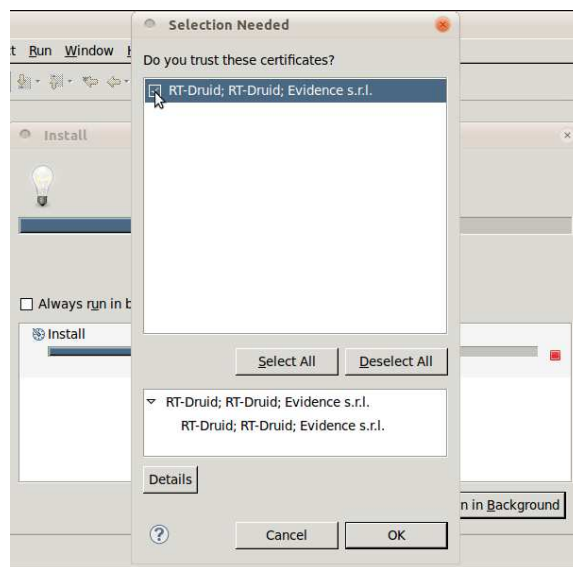


Figure 4.14: Trust of the package RTDruid: we need to accept the trust of RTDruid.

- Window -> Preferences

In the preferences go to:

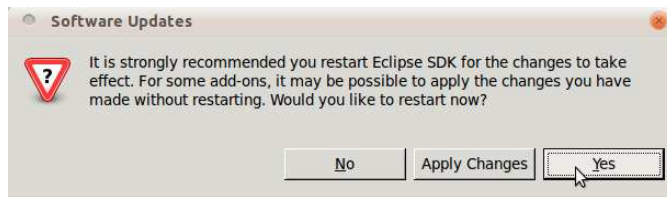


Figure 4.15: Reboot Eclipse: we need to reboot Eclipse.

- RTDruid -> Oil -> dsPic

And we must check if the path is correct (Figure 4.16) (Be careful because its possible that you have another Mplab version, but this is an example of how it must to seem:

- Gcc path /opt/microchip/mplabc30/X.XXy
- Asm path /opt/microchip/mplabx/asm30

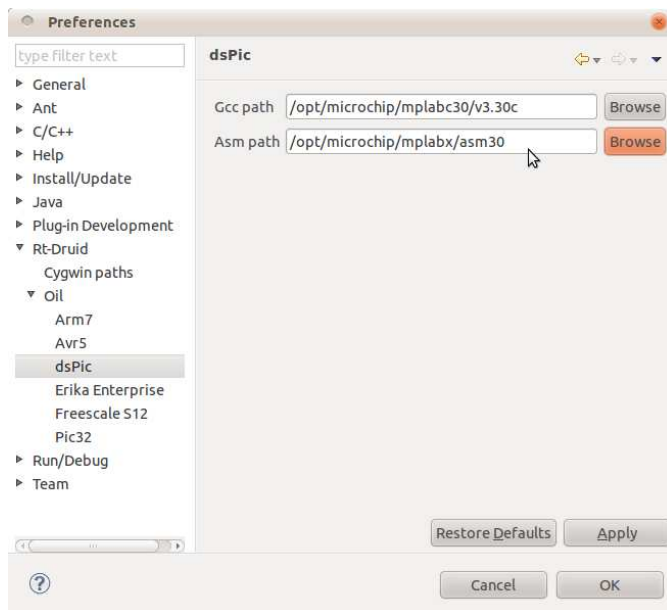


Figure 4.16: RTDruid compiler's path: here we must introduce the path of MplabX installation.

With this steps we will have the environment to program the microcon-  
trollers with the Real-Time Operating Systems Erika

## 4.2 Led Blinking

Here we can see the necessary steps to create a new project with a template, and then compile and program the microcontroller.

### 4.2.1 Eclipse environment

Firstly we need to open Eclipse, this is an programming IDE, and in this case we will use the RTDruid plugin of Evidence. This plugin will allow to program RTOS Erika.

Once the program is opened, lets create a new project (Figure 4.17)

- File -> New -> Project...

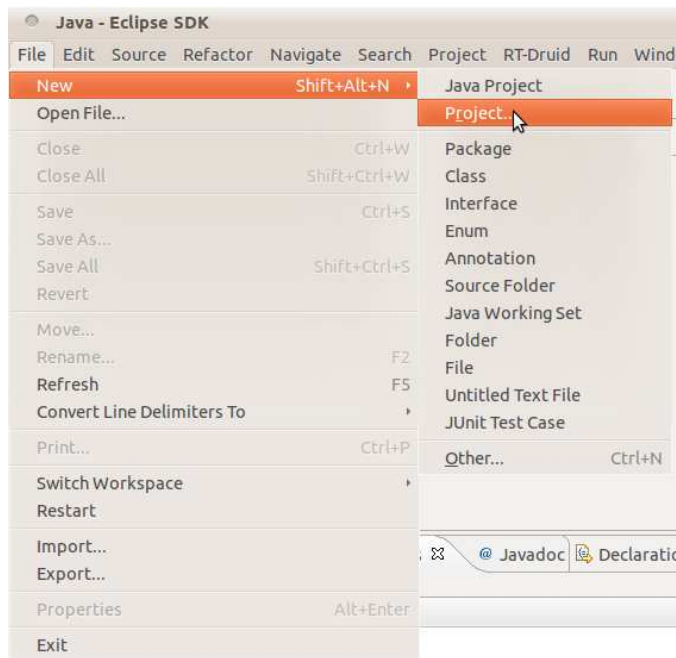


Figure 4.17: RTDruid new project.

We must to select this kind of project (Figura 4.18):

- Evidence -> RTDruid Oil and C/C++ Project

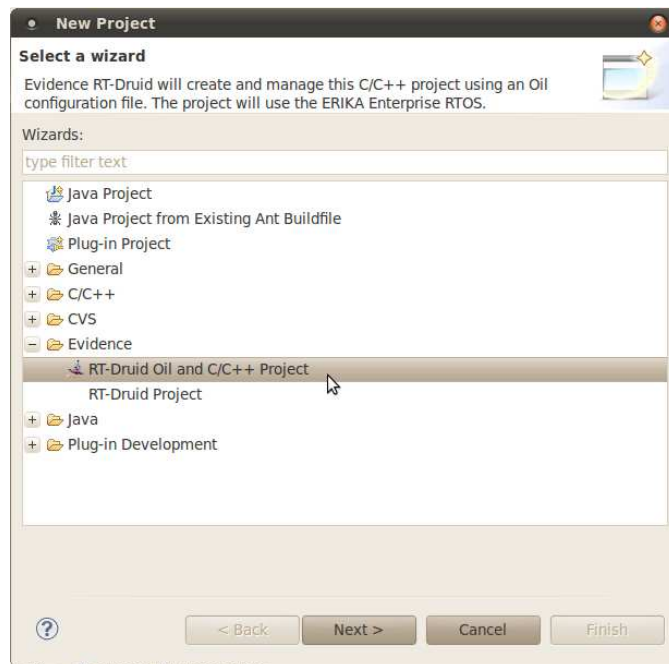


Figure 4.18: Selecting type of RTDruid project.

Then we must assign a name to this project (Figura 4.19) and let the other options by default (you could change the directory, but we recommend to let by default to *workspace*).

In this step we can select a template for the project (Figura 4.20), this will give us all the basic structure to program Erika RTOS. So lets do the following steps:

Check the box *Create a project using one of these templates* and select *pic30* (because in this case we want to program a dsPIC33) and here select any of this templates *templates*, in our case:

- pic30 -> FLEX -> EDF: Periodic task with period

Once we push the Finish button we will have all the environment ready to compile de program.

In case of option *Build Automatically* is activated, Eclipse will compile automatically the first time. It is needed to deactivate this option now. So lets check if the option *Build Automatically* is unchecked (Figure 4.22):

- Project -> Build Automatically

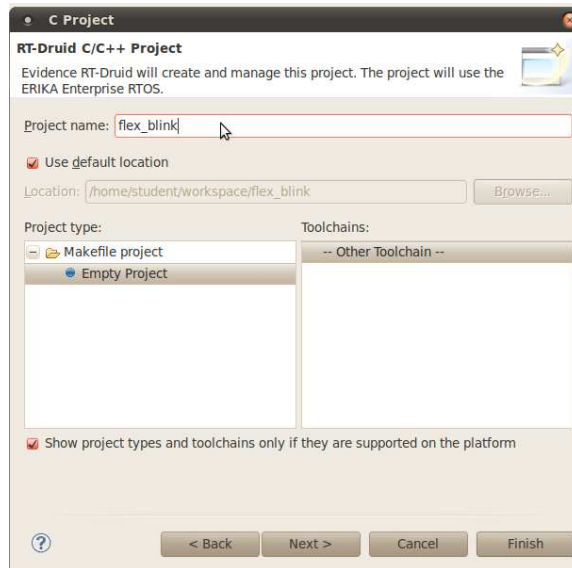


Figure 4.19: Assigning name to a RTDruid new project.

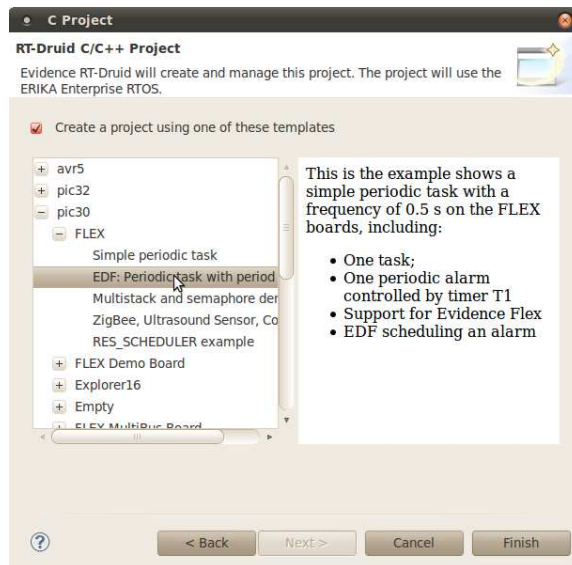


Figure 4.20: Templates RTDruid: in this section we may select any of this templates. This only need to be compiled and programmed to the FLEX board.

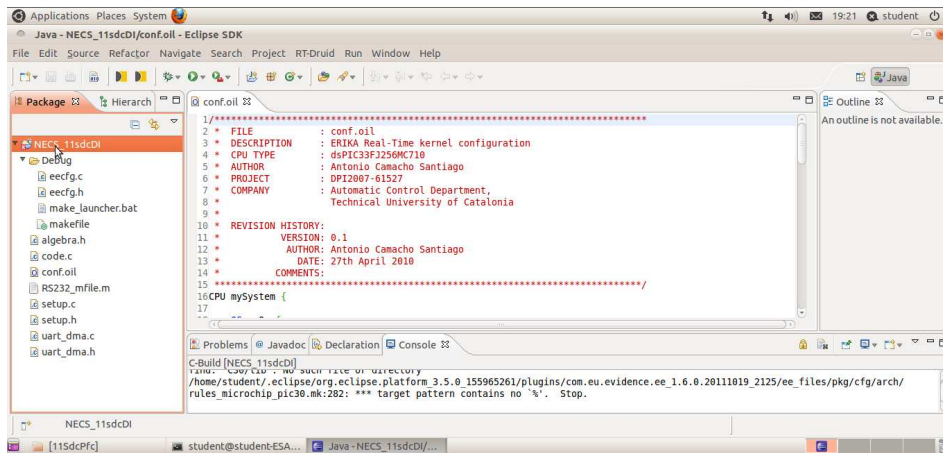


Figure 4.21: Eclipse environment: ready to compile.

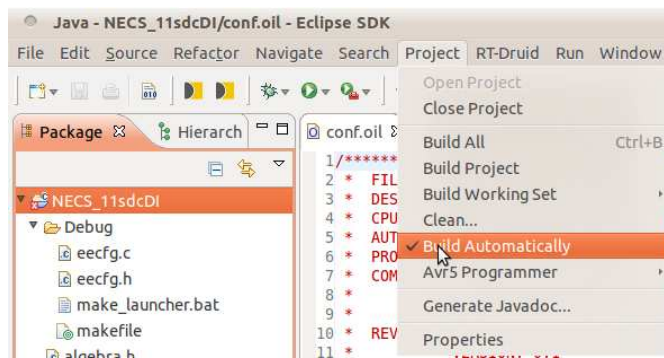


Figure 4.22: Eclipse environment: build

In this moment we can modify the code. When we want to compile the code we could **firstly save the project** (if you do not save the project before compiling, the code compiled may not be the last), once we saved the project go to:

- Project -> Clean

We want to clean to be sure that compiled code is the last saved, and let the other options by default (Figure 4.23) (this options will clean the project and compile it), or select *Clean projects selected below* and *Build only the selected projects* to clean and compile only the selected project.

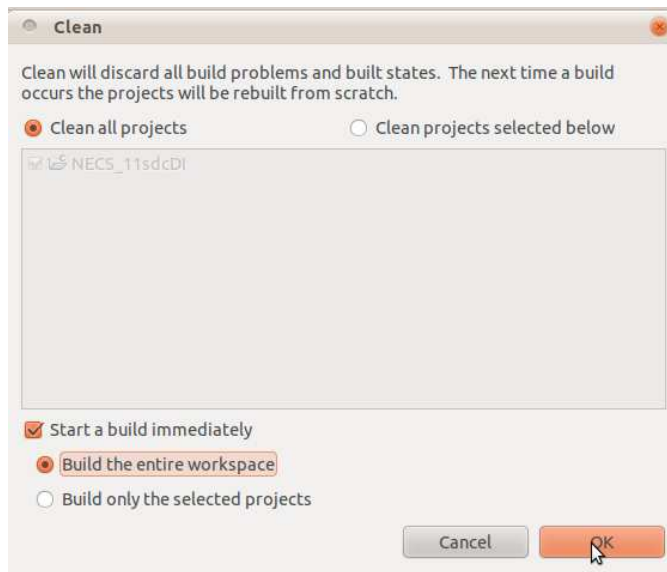


Figure 4.23: Compiling in Eclipse: before compile it is recommended to clean the project.

Once the compilation is complete it will show this message *Compilation terminated successfully!* (Figure 4.24), in this step we would have a new pic30.elf file in Debug directory. This is the file to be saved in the microcontroller with MplabX.

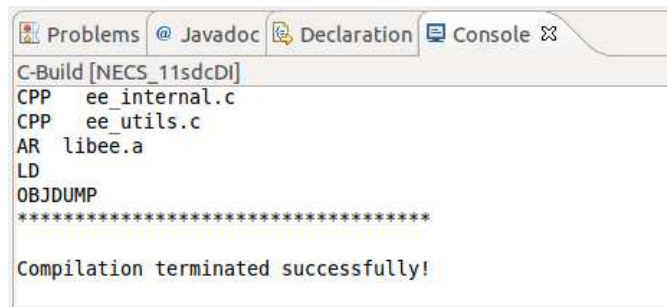


Figure 4.24: Eclipse compiled successfully message.

It is possible that in the compilation step RTDruid do not find compiler's path. In this case check you would check if the installation step 4.1.3 is correct (Figure 4.16).



## 4.2.2 Mplab X environment

Once we have created the pic30.elf file, is time to open the program MplabX. With this program we would program the microcontroller.

Then lets open the program, and lets to create a new project (Figure 4.25):

- File -> New Project...



Figure 4.25: MplabX new project.

Now the program let us to choose the project kind. In our case we have precompiled file and then we want to select our file (Figure 4.26).

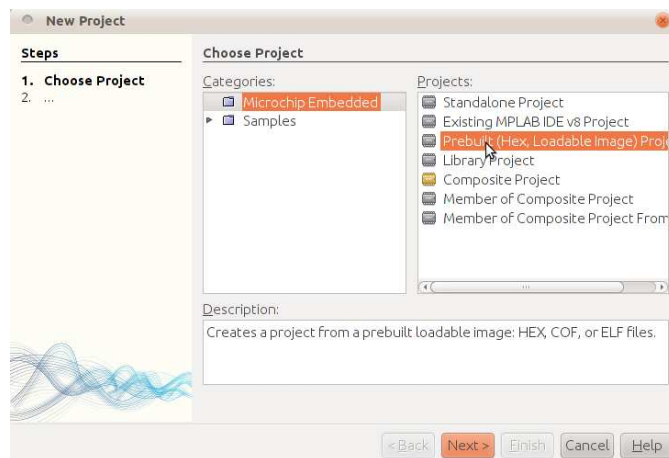


Figure 4.26: MplabX precompiled project.

- Categories : Microchip Embedded
- Projects: Prebuilt (Hex, Loadable image) Project

In next section we need to select our precompiled file, so lets select it (Figure 4.27). If you done all this guide steps, the path must be like this:

- *home/student/workspace/flex\_blink/Debug/pic30.elf*

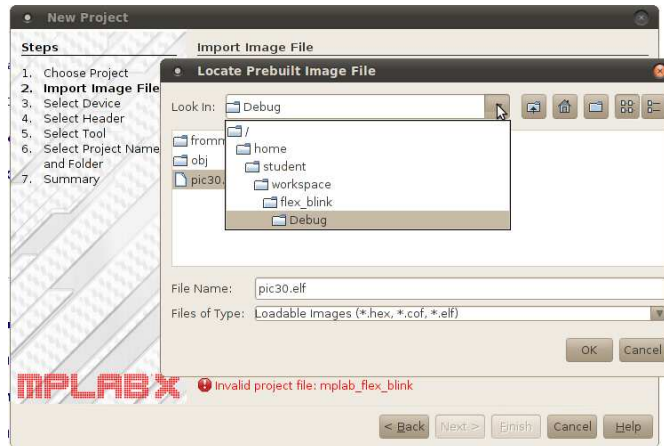


Figure 4.27: Selecting precompiled file in MplabX.

In this step we need to select witch microcontroller we want to program (Figura 4.28), in our case it is:



Figure 4.28: Selecting device in MplabX.

- Family: DSPIC33

- Device: dsPIC33FJ256MC710

Now is the time to select the programmer, in our case to program the FLEX boards we need the Mplab ICD3 programmer (Figura 4.30). We need to connect the programmer to the FLEX board in this moment (Figura 4.29):

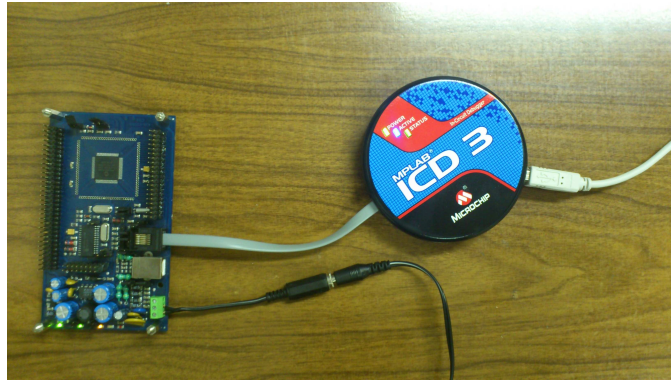


Figure 4.29: Flex and Mplab ICD3: ready to be programmed..

In case of choose another programmer, we need to know what means the the colour in the list:

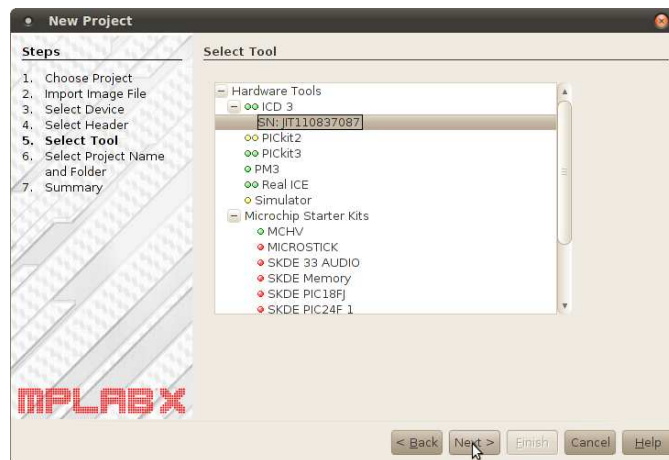


Figure 4.30: List of programmers in MplabX: the colours of the programmers have a meaning that can be consulted at table 4.1.

At last we only have to select a name for this project (Figure 4.31), and *it is very important to change the directory out of Debug* (Figure 4.32). If

Green	This colour means that the programmer is fully functional in this version. .
Yellow	This colour means that the programmer is partially functional in this version. In this case it is possible to select, but it may not work.
Red	This colour means that the programmer is not functional in this version. In this case is not possible to select it.

Table 4.1: Programmers compatibility in MplabX

we leave this by default when we modify the source code with Eclipse and make a *Clean* we would have problems with this project in MplabX.

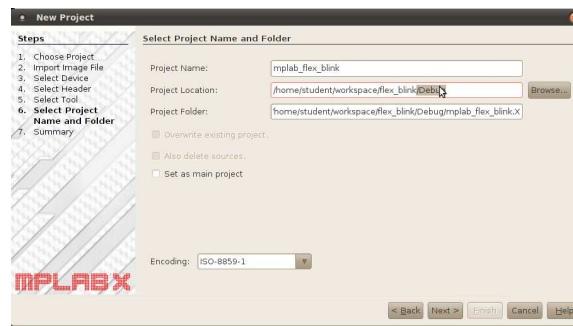


Figure 4.31: Selecting a name for the project in MplabX.

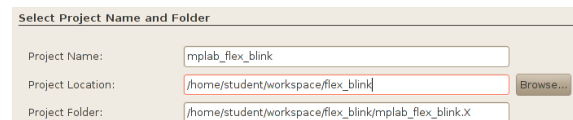


Figure 4.32: Taking out of the directory *Debug*: if we leave this by default when we modify the source code with Eclipse and make a *Clean* we would have problems with this project in MplabX.

Finally MplabX will show up all details of this new project, so we accept and would have all the environment prepared to program the microcontroller.

To program the FLEX board, we must click at button *Make and Program Device* up to the right (Figura 4.33).

Once we have programmed the board, the first orange led of the left would start blinking (Figura 4.34).

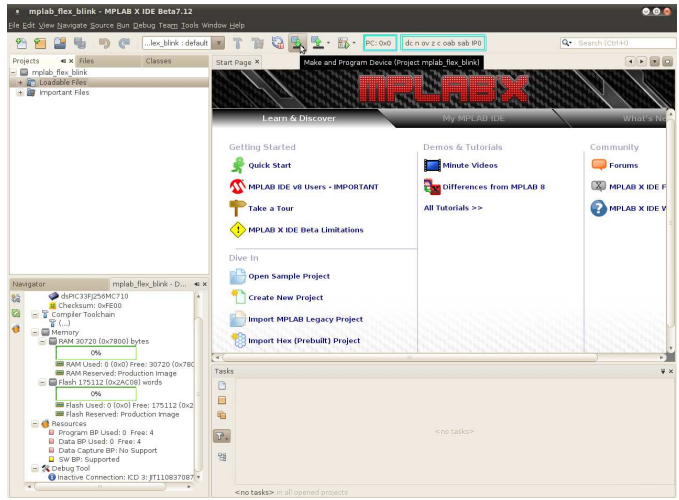


Figure 4.33: Programming FLEX board with MplabX: to program any controller we need to click this button.

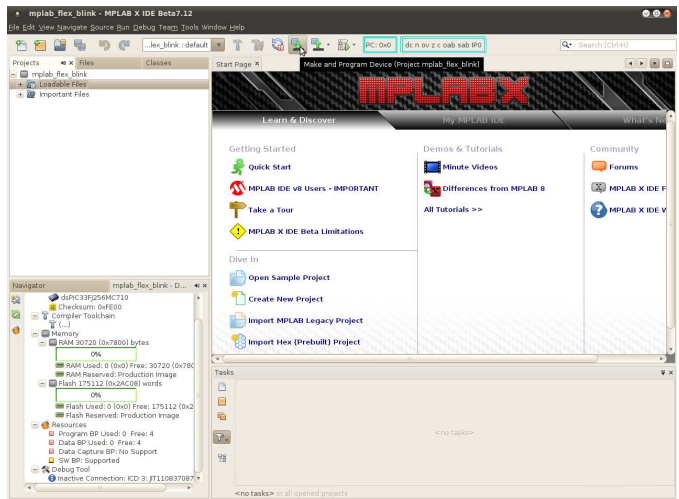


Figure 4.34: FLEX board running program *Blink*: with this program the first led will blink every half second.