# A Domain-specific Language for Real-time Dynamical Systems Emulation on a Microcontroller

**Francisco-David HERNANDEZ[1]*, Domingo CORTES[1],**
**Marco A. RAMIREZ-SALINAS[2], Jorge RESA[1]**

[1] Instituto Politécnico Nacional, ESIME Culhuacán, Av. Santa Ana 1000 Ciudad de México, 04440, México
fcod.hdez@gmail.com (**Corresponding author*), domingo.cortes@gmail.com, jrtipn@gmail.com

[2] Instituto Politécnico Nacional, CIC, Av. Juan de Dios Bátiz S/N, Ciudad de México, 07738, México
mars@cic.ipn.mx

**Abstract:** Translating a control law to code so that it can be executed in real time by a microcontroller is time-consuming and requires knowledge in diverse areas. There are powerful tools like Matlab and DSpace, that can ease the process, however, these tools are expensive and hide the way the translation is actually made. These two factors greatly diminish the use of these tools in education and small business. This paper presents SystDynam, a high-level language designed for describing static and dynamical systems and hence, controllers. The language was purposely created to be easy to process in order to obtain a C code by using free software tools. Therefore, a senior student or a control engineer with a short training in language processors can understand how the translation is made. The necessary code for translation is described here and is freely available. Having the controller described by C code, it can be compiled to be executed as the main task in a real-time operating systems, thereby obtaining the real-time controller. The complete process can also be used for emulating dynamical systems, thereby enabling the use of hardware in the loop simulations and low-cost rapid prototyping and providing an auxiliary tool for teaching some engineering courses.

**Keywords:** Control applications, Controllers, Rapid prototyping, Embedded systems, Control languages.

## 1. Introduction

Moving from a controller design to its real-time implementation is time-consuming and requires knowledge in various areas. To implement a real time controller, it has to be programmed using C language, compiled and executed in a micro-controller (MCU). For a complex control strategy, this task can be time-consuming and error-prone. Fixing errors in the program takes additional time and effort (How, 2019). These factors slow down small companies in developing products that involve controllers. In education environments cause students and professors prefer to make a computer simulation instead of a laboratory experiment or a physical application for testing control strategies (How, 2018).

For decades, many efforts have been made in order to ease the process of translating a control law into its real-time implementation (Hull, et al., 2004). There are software systems like DSpace (Chini, et al., 2017) MATLAB (Banerjee, et al., 2004; Zarrad, et al., 2019) and LabVIEW (Beck, et al., 2006; Chacon, et al., 2015), aimed to assisting and simplifying the different stages of a control application. These are very powerful tools, convenient for a well-established business. However, they have certain characteristics that inhibit their wider use, particularly in education environments and small technology-oriented companies: a) They are closed systems that can hide valuable information about how the translation is made. b) They are expensive.

In an educational setting such characteristics inhibit collaboration among professors and students, and reduce the flexibility that is necessary for experimentation. Furthermore, a hidden process may not facilitate the students` understanding of the process.

Controllers for dynamical systems are dynamical themselves. The static cases can be seen as particular instances of dynamical systems with no states. Hence to implement real-time controllers is tantamount to making a digital system to emulate a dynamical system. Hereafter real-time controller implementation and emulation of dynamical systems by digital systems are treated interchangeably.

The previously mentioned existing tools are based on a domain-specific language in which the dynamical equations are easily expressed (e.g. Simulink in case of MATLAB) and on a processor of such language whose purpose is to generate code which can be executed by a piece of hardware. The hardware can be manufactured by the provider company (e.g. LabVIEW, DSpace) or by a third party (e.g. MATLAB).

https://doi.org/10.24846/v28i4y201909 ICI Bucharest © Copyright 2012-2019. All rights reserved

To create a low-cost alternative, the following are necessary:

a) A language for expressing dynamical systems.

b) Processor for such a language.

c) A piece of hardware to execute the generated code.

There are several languages for expressing dynamical systems (see for example (Elmqvist, 1977; Gumzej, 2007; Ermentrout, 2012). However, it would be convenient to have a language which is rich enough to express a wide variety of dynamical systems and simple enough to be easily processed so that a senior student or control engineer can understand the process. A language processor is a software that processes a program in a certain language and generates a code in another (low-level) language. Typically, a language processor involves a lexical analyser and a syntactic analyser also knows as parser (Sujeeth et al., 2014). Free software tools meant to facilitate the development of language processors have been around for some time (Donnelly et al., 2019; Paxson et al., 2015).

For controller implementation the code generated by the language processor must be executed in real time by a digital system with proper interruption system, digital to analogue converter (DAC), analogue to digital converter (ADC), etc. (Nilsson, et al., 1998). Until recently, most digital boards based on micro-controllers, lacked the resources for running a real-time operating system (RTOS) so that a code could be loaded and executed in real time. However, now there exist boards which are based on 32 bit micro-controllers (32-bit MCU) that can run RTOS, make float point operations by hardware and include analog-to-digital and digital-to-analog converters. In addition, these board include hardware that is useful for controller implementation like pulse with modulation (PWM) modules, communication protocols and general-purpose input-output ports. Most of these micro-controllers are directly programmed in C (Gehani & Ramamritham, 1991).

As a consequence, a controller implementation can now be made, at a low cost, using three components: a) a language for expressing dynamical systems; b) a language processor for such a language generates the actual C code of the controller; c) a set of procedures through which the C code of the controller becomes the main task to be executed by the RTOS running in the 32-bit MCU. In this way the effort to test, modify and experiment with a control strategy can be reduced considerably.

The development of these three components is described in this paper. First, SystDynam a high-level language meant to describe dynamical systems (and hence controllers) is presented. Then the language processor to automatically generate the C code that describe the dynamical system is explained. Finally, the hardware necessary to execute the obtained C code is discussed.

The SystDynam language is flexible enough to describe a wide variety of dynamical systems but at the same time is easy to process. A senior student or a control engineer with a short training in language processors is able to understand how the translation is made and modify that language, if it is necessary.

With the language and its processor here-presented is easy to emulate a great variety of dynamical systems and controllers using a 32-bit MCU board. Thereby enabling a broad range of experiments, for example instead of using a high-cost dynamical system for experimentation, it could be emulated using a 32-bit MCU.

The paper is organized as follows. Section 2 presents the developed language for describing dynamical systems, SystDynam. Section 3 describes the process for generating C code from a SysDynam file. Section 4 discuss in detail the hardware aspects for running the generated code in a 32-bit MCU. The overall process is evaluated using a couple of examples in Section 5 and finally the conclusions are given at the end of the paper.

## 2. The Language for Describing Dynamical Systems

Although there are many languages for describing dynamical systems, the “SystDynam” language described here allows one to describe a variety of systems and at the same time it is easy to process. These two conditions should be met in order for the above-mentioned language to be convenient

and simple enough to be understood by a senior student or control engineer. To introduce the SystDynam language one should consider the description of a simple pendulum shown in Listing 1. In SystDynam language a system is described by means of three blocks. Each of them begins with a reserved word and end with a semicolon. The reserved words in the beginning of each block are: “Parameters:”; “System:” and “InitialC:”, respectively.

**Listing 1.** Description of a simple pendulum in relation to SystDynam language

```
Parameters:
    ti :=0,
    tf :=10,
    dt :=0.1,
    inputs :=1,
    outputs :=2,
    input_scale:=2,
    output_scale :=2
;
System:
    l = 0.5,
    b = 0.15,
    g = 9.81,
    m = 0.5,
    j = m*l*l,
    #u,
    $y1,
    $y2,
    x1’= x2 ,
    x2’= (1/j)*(-b*x2-m*g*l*sin(x1)+u),
    y1=x1,
    y2=x2
;
InitialC:
    x1_0 = 1.2,
    x2_0 = 0
;
```

The block “Parameters:” is used to define those characteristics of the system that are important to know for the real-time execution of the code. The parameters that can be specified are shown in Table 1. Sampling time is the same as integration step size. Parameter *input_scale* is the number corresponding to *1V* of an ADC input channel. Similarly, *output_scale* means the number that corresponds to each volt of a DAC output channel. In the block *System:,* the system dynamics is properly specified by state equations. Each state equation begins with an alphanumeric symbol starting with a letter and ending with an apostrophe. Hence, in the string $x2' = \exp$, the symbol *x2* is identified as a state name and *exp* as the expression of the derivative of that state. If a line begins with a symbol that does not end with an apostrophe it is interpreted as a system dynamic parameter. Hence in the string $j = m*l*l$, the symbols $m$ and $l$ are identified as constants. Note that $m$ and $l$ must be defined before they can be used in expression for $j$. After $j$ is defined, it can be used in other symbol definitions or for defining state derivatives. Any function defined in the standard C math library can be used for defining state derivatives.

**Table 1.** Parameters – reserved words

| Parameters | Function | Example |
|---|---|---|
| ti | Time to start emulation | ti:=0 |
| tf | Time to finish emulation | tf:=10 |
| dt | Sampling time | dt:=0.1 |
| inputs | Number of inputs | inputs:=1 |
| outputs | Number of outputs | outputs:=2 |
| input_scale | Input factor | input_scale:=2 |
| output_scale | Output factor | output_scale:=2 |

Initial conditions for any state can be defined in *InitialC:* block. The symbols must have the same name as the state variables, with the characters “*_0*” added at the end. For example, if *x*1 is a state identifier, the initial condition for this state would be defined as *x1_0=1.2*.

## 3. Processing a SystDynam File for Generating C Code

The syntax of *SystDynam* was designed to be easy to process. Due to this syntax, students with a certain knowledge of C language and a short training in lexical analysers and parsers can understand how to process a *SystDynam* file and make modifications to the language or the language processor. The *SystDynam* language processor can be generated by using free software tools, such as FLex and Bison. The input of FLex is a file that specifies the rules for valid language patterns. The output of FLex is a file in standard C language which contains definitions and prototypes whose compilation enables the generation of the lexical analyser. The specifications are made by regular expressions (*REGEXP*) and a

ICI Bucharest © Copyright 2012-2019. All rights reserved

set of actions. This analyser, is the lexical scanner that verifies the syntax of SystDynam code. The lexical analyser reads a SystDynam source code, and breaks it down into minimal expressions named tokens, it identifies what tokens match the rules and classifies them depending on their respective type. They can be reserved words, identifiers, names of variables, etc. Furthermore, Flex allows it to specify the actions that must been taken when a string matches a token.

Listing 2 shows a part of the Flex file used to build the lexical analyser according to the SystDynam specifications. Here are defined the reserved words and the valid characters. The full Flex file can be consulted on http://github.com/control-lab-org/systdynam/ .

**Listing 2.** Lexical analyser specifications

```
[(]  {yylval.sym=yytext[0];return OFNT;}
[)]  {yylval.sym=yytext[0];return CFNT;}
[-|+]{yylval.sym=yytext[0];return OPA; }
[*|/]{yylval.sym=yytext[0];return OPA1;}
[,]  {yylval.sym=yytext[0];return MORE;}
[=]  {yylval.sym=yytext[0]; return EQL;}

[0-9]+[.]?[0-9]* {strcpy(yylval.val,yytext);
 return NUM; }
[a-z][a-z0-9]*   {strcpy(yylval.val,yytext);
 return VAR; }
[a-z][a-z0-9]*[‘]
{strcpy(yylval.val,yytext);
 return DEQQ;}
[a-z][a-z0-9]*[_][0]
{strcpy(yylval.val,yytext);
 return INITALC;}

“Sistem:”       {  return STRT;      }
“Parameters:”   {  return STRPAR;    }
“InitialC:”     {  return STRINI;    }
“ti:”           {  return TI;        }
“tf:”           {  return TF;        }
“dt:”           {  return DT;        }
“inputs:”       {  return NINPUT;    }
“outputs:”      {  return NOUTPUT;   }
“input_scale::” {  return IN_SCALE;  }
“output_scale:” {  return OUT_SCALE;}
“#”             {  return INPUT_ID;  }
“$”             {  return OUTPUT_ID;}
“;”             {  return STOP;      }

<<EOF>>         { return 0;          }
[ \t\n]+        { }
.               {cout<<”Warning;     }
```

The other tool used for developing the SystDynam language processor is Bison, a software that accepts a context-free grammar specification and generates a deterministic Left to Right general-purpose parsers. In the case of SystDynam, the parser generated by Bison process the file with the description of a dynamical system and produce the equivalent description but in C code. Bison and FLex work together, the lexical analyser generated by FLex is a subroutine of the parser generated by Bison. FLex and Bison generate a syntax-lexical analyser.

To explain how the grammar rules are expressed, Listing 3 shows the set of rules for the block *System:*. To read these rules, the character ‘|’ must be read as ‘or’ and also it should be taken into account that by convention, tokens are written in capitals and non-terminal symbols are in lowercase letters as it is shown in Listing 2. Considering this, the rules in can be read as follows.

**Listing 3.** A part of grammar analyser specifications

```
model:  STRT vvar STOP  {action();}
vvar:   vvar MORE vvar  {action();}
            | static    {action();}
            | deq       {action();}

static: VAR  EQL exp    {action();}
deq:    DEQQ EQL exp    {action();}

exp:    exp MORE exp         {action();}
      | exp OPA1 exp         {action();}
      | exp OPA0 exp         {action();}
      | OPA0 exp %prec OPA1  {action();}
      | OPA1 exp             {action();}
      | ffun %prec VAR       {action();}
      | NUMS                 {action();}
      | VAR                  {action();}
      | OFNT exp CFNT        {action();}

ffun:  VAR OFNT exp CFNT     {action();}
```

A “System” is abstracted by an element called “model:”. A *model* is valid if there is a token STRT (“STRT” = “System:”; see Listing 2) followed by a sequence of symbols (identified by “vvar”) and ending with the STOP token (STOP=“;”). The element *vvar* contains the system body definitions. The element *vvar* is defined recursively. A *vvar* can be formed by two *vvar* separated by the token for MORE (“,”); or be formed by a “static” expression; or a “deq” expression. A *static* expression is a “VAR” token followed by an “EQL” token which is also

followed by an "*exp*" element. The rest of the rules can be read in this way.

The *action()* function appearing in Listing 3, is not a single function, it represent a set of functions that determines how the symbols of the input file are translated. Basically all mathematical operations are converted and translated into C language functions. State equations are also translated into C functions. These functions are passed as arguments of a numerical integration algorithm. The full grammar code and all semantic actions can be accessed on http://github.com/control-lab-org/systdynam.

To integrate state equations, Euler method was used due to it is easy to implement and it is run-time predictable. However, the integration algorithm is not difficult to change if that be deemed necessary.

To sum up, both Bison and Flex generate source code for syntactic and lexical analysers. When they are compiled based on well-defined semantic actions, the Bison application file yields the compiler of SystDynam language. This application processes a file with a description of a dynamical system written in SystDynam language and delivers a file that describes the same dynamical system in a C code. This file is ready to be compiled and then loaded to be executed in a 32-bit MCU.

## 4. Hardware-related Aspects

The description of a dynamic system in C language can be compiled and loaded as the main task on a RTOS running on a 32-bit MCU board. However, for this to be possible it is required that the board meet the following specifications: a) A minimum size of RAM and ROM memory that would enable it to run a RTOS. b) Hardware floating-point operations (FPU), necessary to efficiently perform numerical computations. c) High-resolution (16-bit) Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) with embedded hardware filters. d) A software for fast project development and debugging.

Until very recently, a hardware with these requirements was expensive. Thanks to the rapid development of technology there are now high-performance and cheaper 32-bit MCUs. For the test presented in Section 5, the Freedom-K64F (FRDM-K64F) (NXP Semiconductors, MCUXpresso IDE User Guide, 2018) development board was employed. This board is powered by a 32-bit Advanced Reduced Instruction Set Computer Machine (ARM) Cortex-M4. With a 1024 KB flash memory, 254 KB RAM, combined with a *120* MHz base clock frequency, various communication ports like SPI, I2C, UART, Ethernet and USB, drives like PWM modules, PLL, and more flexible timers, etc. The manufacturer of this board also provides an Integrated Development Environment (IDE) and Software Development Kit (SDK), that make board configuration, and the FreeRTOS (RTOS used) configuration and installation easier.

FreeRTOS is a type of RTOS that is designed to be small enough to run on a MCU. However, its use is not limited to MCU applications. FreeRTOS is a kernel for embedded systems developed and maintained by the team of Real Time Engineers Ltd. It is an Open-source project distributed under a MIT license. It provides the core Real-time scheduling functionality, inter-task communication, timing and synchronization primitives only. FreeRTOS kernel allows applications to be organized as a collection of independent tasks with priorities. Each priority is assigned by the application developer. For the program that emulates a dynamical system, the necessary tasks are: computing state values, and the communication between ADC, DAC and the plot buffer.

## 5. Evaluation

In order to test SystDynam language, its language processor and the procedures for compiling and loading the C code generated, different dynamical system descriptions were coded in SystDynam. The description of dynamical systems instead of controllers was preferred for evaluation because a dynamical system can be tested separately. By contrast, a controller has to be applied to other dynamical systems. Nevertheless, experiments are meaningful because a dynamical controller is basically a dynamical system, and a static controller can be considered a dynamical system without states. Furthermore, in both experiments a state is actually a feedback into the system.

The dynamical systems chosen to evaluate the tool here-developed are classical in the study of dynamical systems. To make experiments with these systems, a lab equipment is required. One

ICI Bucharest © Copyright 2012-2019. All rights reserved

way is construct a lab prototype with error prone and on the other hand, buy the equipment to a laboratory that is not usually cheap. However, with the SystDynam language and its processor, these systems can be emulated using a low cost 32-bit MCU. In the same way other systems can be emulates such as DC and AC electric motors, simple robots, etc.

Each system description was input into the language processor and C code was generated. The C code obtained was compiled by a standard C compiler and loaded to the MCU to be executed in real-time by the FreeRTOS operating system. Finally, measurements were taken with the oscilloscope on corresponding output pins. These measurements were compared with the results of a simulation of the same system in Simulink. The method used in Simulink was ODE45 with a relative tolerance of $10^{-6}$.

The MCU employed limited the analogue signal range of the ADC input and the DAC output to 0-3.3v. This can be modified by external hardware. However, to reduce the influence of external hardware to a minimum, in examples presented below, numbers that pass through the ADC and DAC to be measured externally are manipulated such that zero correspond to *3.3/2=1.65 V*. Thus, positive numbers result in the range *[1.65V-3.3V]* and the range for negatives is *[0V-1.65V)*. Proper output scaling should be specified for each SystDynam description to keep the output voltage within these ranges.

## 5.1 Evaluation of SystDynam Using a Simple Pendulum Model

A simple pendulum is modelled by (1), (2).

$$\dot{x}_1 = x_2 \tag{1}$$

$$\dot{x}_2 = \frac{1}{J}(-bx_2 - mgl\sin(x_1) + u) \tag{2}$$

Where $x_1$ and $x_2$ are the angular position and velocity respectively, $m$ is the pendulum mass, $l$ is the pendulum length, $J$ is angular momentum, $b$ is the friction coefficient and the input $u$ is a torque applied to the pivot. Models (1) and (2) are described in SystDynam by the code illustrated in Listing 1.

The input and output scale was ×*2*, hence each volt at the input represents two radians and also each volt at the output represents two radians. The initial values of the pendulum states are $(x_1, x_2) = (1.2, 0)$. The same values were used in the Simulink model.

Results obtained when $u = -x_2$, are shown in Figure 1, Figure 2 and Figure 3. The numerical values calculated by the MCU, which are superimposed to the Simulink simulation results are shown in Figure 1 and Figure 2.

These results are available thanks to the board debugging tools that allow one to record and access the internal values generated when the program is running. To achieve this feedback, the corresponding DAC output channel was wired to the ADC input channel programmed to be $u$. The minus sign was set by software. Note how the $x_2$ feedback affected the MCU calculated values in almost the same manner as Simulink.

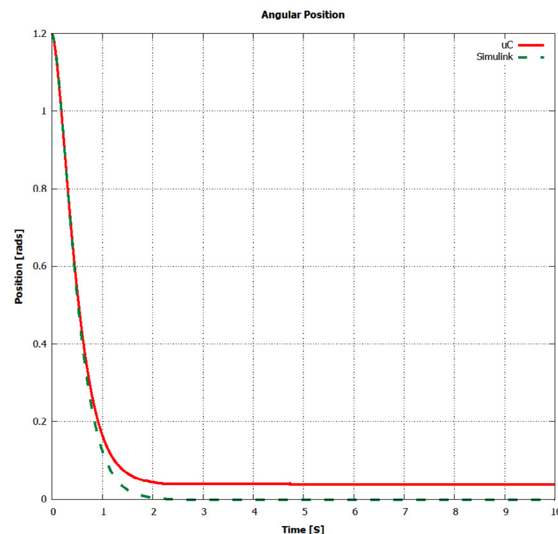


**Figure 1**. Pendulum Angular Position

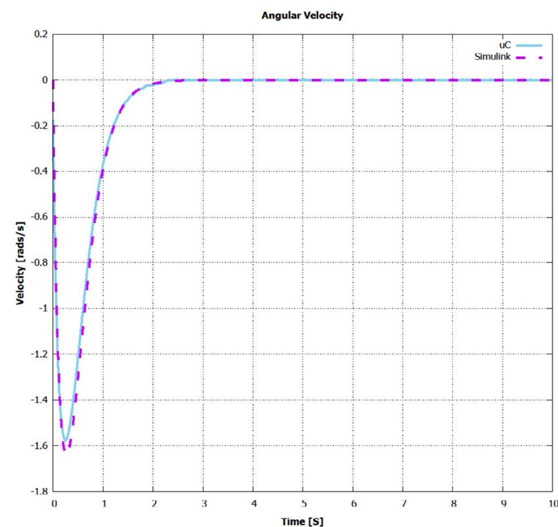


**Figure 2**. Pendulum Angular Velocity

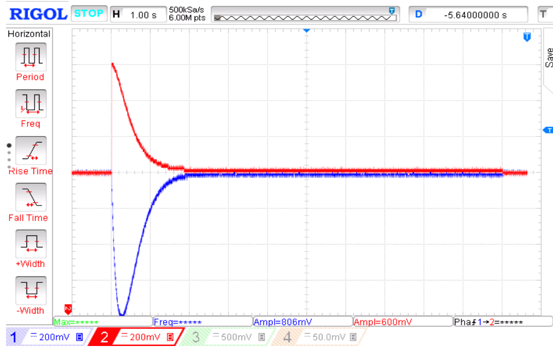

**Figure 3**. Simple pendulum ($u = -x_2$): signals measured at DAC output

State signals measured at the DAC channels of the board are illustrated in Figure 3. The state $x_1$ does not reach zero in the MCU because the maximum DAC output is not 3.3V but 3.26V. Hence, the offset of 1.65V does not correspond exactly to zero. As a consequence, the system behaves as if it received a small constant input.

## 5.2 Van Der Pol Model Evaluation

A classical model of an excited Van Der Pol oscillator is given by (3) and (4).

$$\dot{x}_1 = x_2 \tag{3}$$

$$\dot{x}_2 = \xi\left(x_2 - \frac{x_2^{\ 3}}{3}\right) - x_1 + u \tag{4}$$

where $\xi$ is a constant and $u$ is an external input. The SystDynam code for this example is shown in Listing 4.

**Listing 4.** Description of a Van Der Pol model on SystDynam language

```
Parameters:
    ti:=0 ,
    tf:=10 ,
    dt:=0.01 ,
    inputs:=1 ,
    outputs: =2 ,
    input_scale: =2,
    output_scale: =2;
System:
    epsi=0.01,
    #u,
    $y1,
    $y2,
    x1'=x2,
    x2' =epsi*(x2-(x2*x2*x2)/3)-x1+u,
    y1=x1,
    y2=x2;
InitialC:
    x1_0 = -1.0,
    x2_0 = 0;
```

This example is interesting because depending on $u$ three behaviors can be noticed. If $u = 0$ a stable oscillation is obtained. If $u = x_1$, the origin is an equilibrium point. If $u = x_2$, increasing oscillations can be noticed. The experiment and results for increasing oscillations, $u = x_2$, are shown below in Figure 4, Figure 5 and Figure 6.

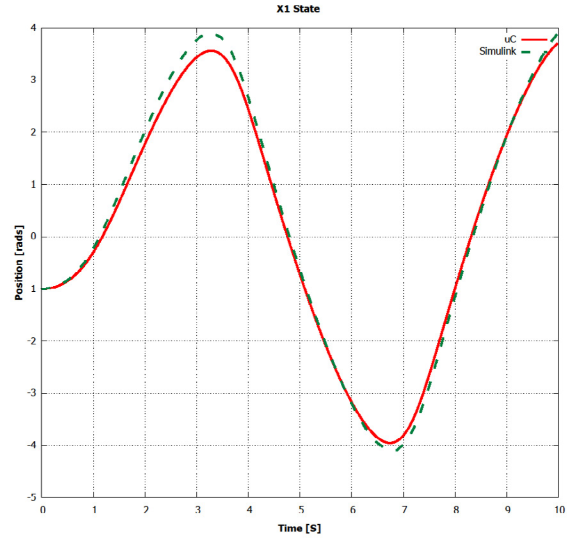


**Figure 4**. $x_1$ state, Van Der Pol linear oscillator

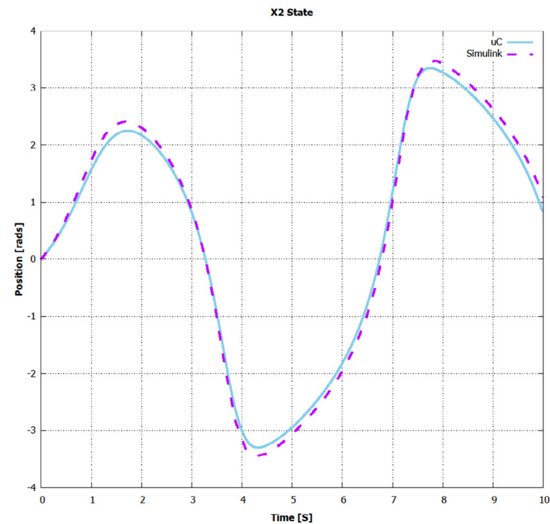


**Figure 5**. $x_2$ state, Van Der Pol linear oscillator

To make the feedback $u = x_2$, the pin signal corresponding to the $x_2$ state, in the output channel of the DAC, was routed to the corresponding control input pin of the ADC channel.

Both states were calculated by the MCU and superimposed with the data obtained by Simulink. Note that oscillations increased as it was expected. Differences between MCU data and Simulink as

ICI Bucharest © Copyright 2012-2019. All rights reserved

illustrated by these graphs are due to quantization errors in the ADC and a more notable deviation between Euler method used in the MCU and the *ode45* method used in Simulink for unstable systems. Figure 6 depicts both system states measured in the respective DAC output channels with the oscilloscope. Here, increased oscillations can also be noticed. However, due to DAC has a limited output range, as signals` amplitude increases and saturation can be noticed.

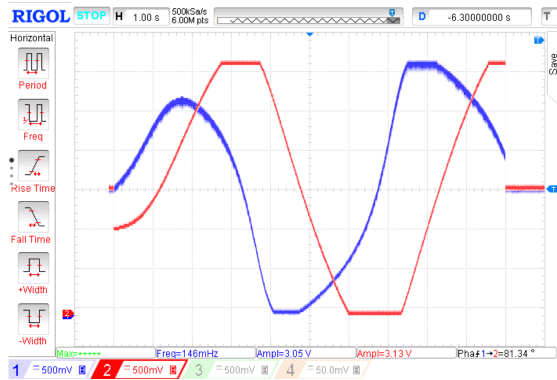


**Figure 6.** Van Der Pol linear oscillator ($u = x_2$): signals measured at DAC output channels

## 6. Conclusion

Having the experience of physically manipulate and control dynamical systems is important for control engineering students. In this paper it was described a set of tools to ease acquiring such experience without hiding any step of the process. Thus, a student will be able to modify or fine-tune any component of hardware or software involved in the experiment. These tools include SystDynam, a high-level language designed to describe static or dynamical systems. The language, which was designed to easily describe a dynamical and be simple to translate the description into a C code. In this way, a senior student or a control engineer can understand how the C code is generated. A language processor that is fed with a SystDynam description and generates a C code was developed and is freely available.

The generated C code can be compiled and run as the main task of a real-time operating system in a MCU. The results obtained by using the FreeRTOS running in a FRDM-K64F powered by an ARM Cortex M4 were presented.

The SystDynam language and its language processor is meant to ease controller implementations. Furthermore, it can be used to emulate dynamical systems and make simulations with hardware in the loop (HIL). With a pair of boards, a dynamical system can be emulated in one board and the controller in the other one, which would give way to a variety of experiments. It can also be used for rapid prototyping, particularly in small technology-oriented businesses that cannot afford the cost of other tools. Because, SystDynam is easy to translate but powerful at the same time, it also can be useful in other engineering contexts, like signal processing, programming, real-time systems, embedded systems, compilers, etc.

## REFERENCES


1. Banerjee, P., Haldar, M., Nayak, A., Kim, V., Saxena, V. S. & Uribe, J. (2004). Overview of a compiler for synthesizing MATLAB programs onto FPGAs, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 12*(3), 312-324.

2. Beck, D., Brand, H., Karagiannis, C. & Rauth, C. (2006). The first approach to object oriented programming for LabVIEW real-time targets, *IEEE Transactions on Nuclear Science, 53*(3), 930-935. DOI: 10.1109/TNS.2006.873532

3. Chacon, J., Vargas, H., Farias, G., Sanchez, J. & Dormido, S. (2015). EJS, JIL Server, and LabVIEW: An Architecture for Rapid Development of Remote Labs, *IEEE Transactions on Learning Technologies, 8*(4), 393-401.

4. Chini, A., Azza, H. B., Jemli, M. & Sellami, A. (2017). Nonlinear Discrete-Time Integral Sliding Mode Control of an Induction Motor: Real-Time Implementation, *Studies in Informatics and Control, 26*(1), 23-32. DOI: 10.24846/v26i1y201703

5. Donnelly, C. & Stallman, R. (2019). *Bison Manual,* 51-126. Free Software Fundation.

6. Elmqvist, H. (1977). SIMNON - An Interactive Simulation Program for Non-Linear Systems. In *Proceedings of the*

*International Symposium SIMULATION '77*, Montreux, Switzerland, M. Hamza (ed.), ACTA Press, Anaheim, CA (pp. 85-90).

7. Ermentrout, B. (2012). Xppaut. In: Le Novère N. (eds) *Computational Systems Neurobiology,* 519-531. Springer, Dordrecht. DOI: 10.1007/978-94-007-3858-4_17

8. Gehani, N. & Ramamritham, K. (1991). Real-time concurrent C: a language for programming dynamic real-time systems, *Real-Time Systems, 3*(4), 377-405.

9. Gumzej, R. (2007). Modeling distributed real-time applications with specification PEARL, *Real-Time Systems, 35*(3), 181-208.

10. How, J. P. (2018). Future Controls Courses [From the Editor], *IEEE Control Systems Magazine, 38*, 3-4.

11. How, J. P. (2019). Why Do Experiments? [From the Editor], *IEEE Control Systems Magazine, 39*, 4-6.

12. Hull, M., Ewart, S. & Hanna, J. (2004). Modeling Complex Real-Time and Embedded Systems - The UML and DORIS Combination, *Real-Time Systems, 26*(2), 135-159.

13. Nilsson, K., Blomdell, A. & Laurin, O. (1998) Open Embedded Control, *Real-Time Systems, 14*(3), 325-343.

14. NXP Semiconductors (2018). *MCUXpresso IDE User Guide,* 37-130.

15. Paxson, V., Estes, W. & Millaway, J. (2015). *Lexical Analysis with Flex.* The Regents of the University of California.

16. Sujeeth, A. K, Brown, K. J., Lee, H., Rompf, T., Chafi, H., Odersky, M. & Delite, K. O. (2014). A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages, *ACM Transactions on Embedded Computing Systems (TECS), 13*(4), 1-25.

17. Zarrad, O., Hajjaji, M. A. & Mansouri, M. N. (2019). Hardware Implementation of Hybrid Wind-Solar Energy System for Pumping Water Based on Artificial Neural Network Controller, *Studies in Informatics and Control, 28*(1), 35-44. DOI:10.24846/v28i1y201904

ICI Bucharest © Copyright 2012-2019. All rights reserved