Developing a Generic Purpose OpenModelica Library for Embedded Applications

Submitted in partial fulfillment of the requirements of the degree of Master of Technology

by

Manas Ranjan Das

Under the guidance of

Prof. Kannan M. Moudgalya
Department of Chemical Engineering
IIT Bombay

Systems & Control Engineering, IIT Bombay
July 11, 2019

Approval Sheet

This thesis entitled "Developing a Generic Purpose OpenModelica Library for Embedded Applications" by Manas Ranjan Das (Roll no. 163236001) may be accepted for being evaluated for the award of the degree of Master of Technology.

....
Prof. Kannan M. Moudgalya (Supervisor)

....
Prof. Leena Vachhani (Examiner)

....
Dr. Sunil S Shah (Examiner)

Declaration

I, **Manas Ranjan Das**, Roll No. 163236001, declare that the work presented in this report is carried out by my own. Also, wherever someone else's ideas have been used, they have been cited and nothing has been copied verbatim. Due credit has been given to all the contributors and authors, whose ideas and theories reflecting in this project report.

....
Manas Ranjan Das (163236001)

Acknowledgment

I am grateful to Prof. Kannan M. Moudgalya for giving me the opportunity to work in FOSSEE, introducing me to OpenModelica and its significance. His guidance and constant encouragement has always been a motivation for me. My sincere thanks to Dr. Sunil Shah, Mr. Ritesh Sharma & Mr. Pavan of Modelicon for their valuable suggestions. I am also thankful to Prof. Peter Fritzson for his valuable suggestions for further improvements. I am grateful to all my Team members, especially Siddharth at FOSSEE, IIT Bombay for his help. I would like to recognize the support and contribution of all the team members at FOSSEE. My sincere gratitude to all who helped me knowingly or unknowingly throughout the project.

Abstract

A library has been developed for OpenModelica that supports different kinds of micro-controllers, Single Board Computers (SBCs) and also 4DIAC, which is an IEC 61499 based framework for process control implementation. The software drivers written in C/C++ is called by OpenModelica at its back-end to facilitate interface between the software and the external hardware. UART and Inter Process Communication (IPC) protocol are used to interface ATmega series of controllers. To interface ARM Cortex-M4 micro-controller, which is a high-end controller, firmata protocol is used. For SBCs like Raspberry Pi and for the software framework like 4DIAC, Open Communications Platform Unified Architecture (OPC UA) is adopted. The developed library also facilitates Hardware In Loop (HIL) simulation with embedded target and Software In Loop (SIL) simulation with 4DIAC framework.

Key Words: OpenModelica, UART, IPC, OPC UA, ARM, SBC, HIL, SIL etc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

OpenModelica (OM) [8] [14] [5] is a free and open source environment based on the Modelica modeling language for simulating, optimizing and analyzing complex dynamic systems. OpenModelica is used in academics as well as industrial environments. Industrial applications include the use of OpenModelica in the domains of automation, power plant optimization, automotive, etc. Models are either built through line by line code or graphical blocks in OpenModelica. It can interact with C, Python languages and can call C, Python functions from within its models. OpenModelica is a powerful tool that can be used to design and simulate complete systems.

Embedded systems can be defined as a computer (software and hardware) buried in a high stress environment (weak consumption, reduced memory capacity, real-time, security, robustness). This project describes the demonstration of low cost development tools and real-time simulation for rapid prototyping of autonomous embedded systems. This is an integration of mechanics, electronics, automation and informatics in design and manufacturing of a product to increase and/or optimize its functionality in minimum cost.

As the demand for compact devices increases, the sizes of processors and microchips keep shrinking, which requires the development of complex control systems. It is necessary to monitor the entire embedded control system and application design processes to optimize the overall system design. Here, the model-based design approach proves to be an effective and efficient means of understanding the product parts such as commercial micro-controllers and processors as well as algorithms and code for the working of both microelectronic and embedded devices. Model-based design (MBD) [18] performs verification and validation through testing in the simulation environment. It covers various disciplines, functional behavior, and cost/performance optimization to deploy a product from early concept of design to final validation and verification testing. But the price of the commercial MBD tools being quite expensive, we came up with an open-source implementation to facilitate MBD for Embedded Systems. Here, OpenModelica, being an Open Source modeling and simulation environment perfectly fits to this purpose. It has a rich set of libraries in different domains for systems modeling and simulation.

OpenModelica inherently doesn't have the facility to support external embedded devices. To support external devices in simulation, there should be a facility to send and receive data from and to the devices connected to the systems. And to facilitate this, there has to be some software drivers to implement different protocols to connect different devices to OpenModelica. This involved developing back-end C/C++ drivers to facilitate communication between OpenModelica and the embedded devices, also front-end GUI block sets for users to create models. But the real issue is to come up with a generic solution so that the user should be able to connect any kind of embedded device, be it micro-controllers or Single Board Computers (SBCs), for example Raspberry Pi with OpenModelica.

Here the task is to implement model based design for different embedded targets i.e. creating models for different embedded applications in OM and communicating with different families of micro-controllers and also with different Single Board Computers (SBCs) such as Raspberry Pi. There are different protocols followed depending on the architecture and flash memory available on the controller. The approaches were different while shifting from micro-controllers to SBCs. ATmega series of controllers have been interfaced by UART protocol approach. The implementation supports all kinds of sensors like digital, analog and pwm in OpenModelica. But due to the high latency, Hardware In Loop (HIL) couldn't be implemented on this implementation. So, to enable HIL, Inter Process Communication (IPC) approach was adopted for ATmega series. Next approach is to develop support for ARM Cortex-M4 which is a 32-bit high end micro-controller. Firmata protocol was considered for this implementation as it has very low latency for data transfer between the software on host computer and the external embedded device. The final task was to provide support for SBCs like Raspberry Pi and another software i.e. 4DIAC which is a standard industrial framework for process control systems. Here, it's quite tedious to follow the firmware approach as in case of micro-controllers as it's quite cumbersome to develop a firmware for a OS that sits on the SBCs. So, a much more generic approach such as Open Platform Communications Unified Architecture (OPC UA) was adopted to communicate with SBCs and 4DIAC. Though, initially there was a heavy dependency on drivers to be written for each and every family of micro-controllers. But at the end, this dependency was overcome by adopting the OPC UA protocol which provides a kind of generic platform for all kinds of controllers and SBCs. This OpenModelica library developed as part of this task supports Software In Loop (SIL) with 4DIAC which is a IEC 61149 platform to implement industrial level controllers and also Hardware In Loop (HIL) simulation on embedded platforms like Arduino and TivaC (ARM Cortex-M4), ATmega16 and Raspberry Pi etc. Different protocols like UART, Inter Process Communication (IPC), firmata and OPC UA were implemented & tested to support different families of micro-controllers and SBCs such as Raspberry Pi.

Chapter 2

UART Protocol for Interfacing AVR ATmega family of controllers

2.1 Motivation

Both Arduino [1] & ATmega16 [2] are based on AVR family of controllers. Arduino is based on ATmega328p with 32kbs of flash memory & ATmega16 has 16KBs of flash memory. The Arduino UNO is a widely used microcontroller board based on ATmega328P microcontroller IC, developed by arduino.cc. It operates at a voltage of 5V. The board contains 14 Digital and 6 Analog input/output (I/O) pins, a 10-bit ADC (Analog to Digital Convertor), 8-bit DAC, an in-built LED connected to digital pin no. 13 and many other features shown in Figure: 2.1

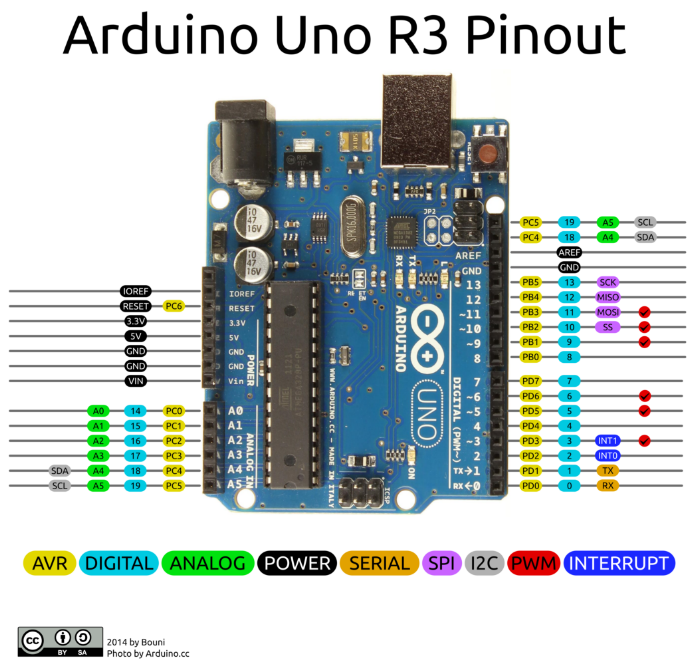


Figure 2.1: Pin Diagram of Arduino UNO

The approach to provide support for these controllers from OpenModelica is based on serial communication by UART protocol.

Basic idea behind serial communication with AVR ATmega series of microcontroller is to configure the port where the required hardware is connected to PC using USB cable and identifying the port. The information is therefore used in establishing serial communication route with these development boards and OpenModelica software running in the system. All the configurations of the serial port are done using external C functions which can be called by OpenModelica [4].

2.2 Algorithm

The architecture of serial communication implementation is as shown in Figure 2.2

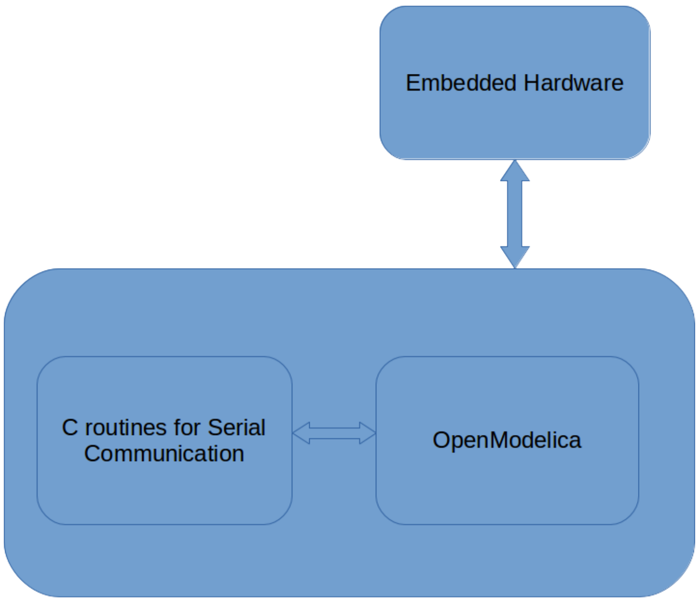


Figure 2.2: Architecture of UART Implementation

2.3 Implementation on OpenModelica part

As OpenModelica doesn't have the capability to interact with embedded devices, there is need of some drivers through which it can send and receive data from the external devices.This implementation establishes serial communication of OpenModelica with the external ATmega based devices through UART protocol. Basic idea behind serial communication with ATmega based device is to configure the port where the board is connected to PC using USB cable and identifying the port. The information is therefore used in establishing serial communication route with board and OpenModelica software running on the system. All the configurations of the serial port are done using external C functions which are called by OpenModelica at the back-end.

2.3.1 Functionalities added to OpenModelica

The five basic functionalities required in this case are: open_serial, close_serial, read_serial, write_serial & status_serial. These functions initiate serial communication with the hardware platform and are used in other interfacing functions to establish communication. Before using these functions, the hardware setup must be loaded with a firmware program. This program contains specific set of identifiers to recognize instructions sent through the serial port.

The basic functions are as follows:

1. **open_serial**:- It takes in parameters as integer handle, port number on which device is attached, and baud rate at which it has to communicate with the device. The function opens the serial port (a file descriptor) and returns 0 if serial port is successfully opened. In case of a bad file descriptor/failure to open serial port it returns an integer. It also calls function **set_interface_attribs** to set the baud rate and other attributes of the serial port interface and the function **set_blocking** to disable blocking.

2. **close_serial**:- It takes in parameters handle to the serial port as an argument. The function closes the serial port (file descriptor) and returns 0. If the port closes successfully then a success message is printed, else not.

3. **read_serial**:- It takes in parameters handle, a character array that will return the characters read from the file identified by handle and the number of characters/bytes to be read from the serial port. The function reads 'n' number of characters from the serial port where 'n' is the size specified by the function caller. If read is successfully performed then the characters are copied to the input argument buffer and a 0 is returned else an integer 2 is returned by the function to denote error.

4. **write_serial**:- It takes in parameters handle, character array to be written to serial port and the size of the character array. The function sends/writes the given char array to the serial port and on successful write, a message is printed, else nothing is printed. The function returns 0.

5. **status_serial**:- It takes in parameter handle and contains the information of the bytes of data read and written through the serial port. It returns 0 on success.

In addition to the above basic functions, there are supporting interfacing functions to support digital, analog and pwm functionalities. All the functions are then called from within OpenModelica from a functions package to avail its functionalities. For a sample implementation for digital data exchange, please refer Appendix E.

2.4 Implementation on Controller part

On the controller part, there has to be a firmware to recognize the data coming from the OpenModelica tool.

2.4.1 Firmware Structure

The firmware has basic set of functionalities to receive, recognize and send different types of data with OpenModelica. Depending on the ASCII character received, the attached pin has been assigned as digital, analog, pwm and also the incoming data is deciphered as digital, analog or pwm and accordingly the task is performed. If the incoming ASCII value ranges from 2 to b, then the pin attached is recognized as digital and accordingly it can be made HIGH (1) or LOW (0). Similarly, ASCII values are defined for analog and pulse width modulation (pwm) functionalities.

2.5 Issues

Major drawback of this implementation is that, this is a soft real-time simulation package and time synchronization with model and real-time data. Due to lack of time synchronization between OpenModelica & system clock, results were not proper for Hardware In Loop (HIL) application.

The next approach is to use Inter Process Communication (IPC) which implements time synchronization functionality in a much better way.

Chapter 3

IPC Protocol for Interfacing AVR ATmega family of controllers

3.1 Motivation

To overcome the short-coming of the time synchronization issue with UART protocol, Inter Process Communication (IPC) was adopted, as the delay in data transfer here is very negligible. Hence, it is best suited for Hardware In Loop (HIL) simulation. The InterProcessCommunication package, which is central to this toolbox, was developed at ModeliCon with the intent of communicating two PCs. We modified it to work with an Arduino Uno/Mega and a PC. The reason that the IPC package was chosen over conventional packages is its ability to transfer data independent of data type. This comes in handy when dealing with the wide range of values that can be assumed while working with a controller such as a PID.

3.2 Algorithm

Inter Process Communication (IPC) is a communication process by which no. of different processes shown in Figure:3.1 can share information with each other running on a same or different systems. There are different methods to implement Inter Process Communication (IPC) but here we are adopting the shared memory [12] approach.

In this approach, a part of memory is shared between two or more processes and then data can read/write to this memory space simultaneously. In our case, one process is OpenModelica and other one is an embedded device such as ATmega328/ATmega16.

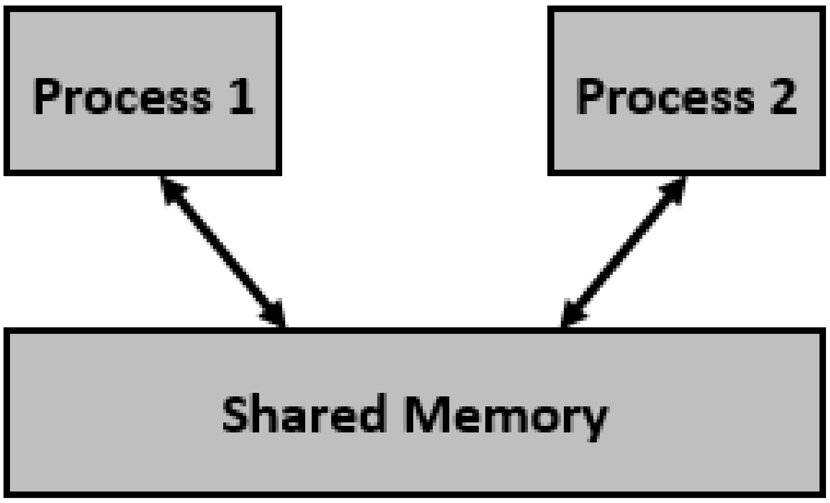


Figure 3.1: Shared Working Memory for IPC

3.3 Implementation on OpenModelica part

The directory structure for the library is described below. Also as package.mo and package.order are present in multiple directories within InterProcessCommunication, and are largely irrelevant to direct usage, only those at the top level have been listed.

```
└─ ArduinoCode
   ├─ ArduIPCWrite
   │  └─ ArduIPCWrite.ino
   ├─ basic_ write
   │  └─ basic_ write.ino
   └─ IPC_ PID
      └─ IPC_ PID.ino
```

```
InterProcessCommunication
    Examples
        CombinedExamples
            PIDandMotor.mo
            PulsePIDandMotor.mo
        InterProcessExamples
            ArduinoIPC.mo
            DC_ Motor_ Arduino.mo
    Info
        Tutorial
            Advanced.mo
            GettingStarted.mo
        Contact.mo
        Overview.mo
    Resources
        Include
            Arduino_ port.sh
            SerialMI.h
            Serial_ SHM.c
            ShmMI.c
        Library
            linux64
                librt.a
                librt.so
    SharedMemory
        SharedMemoryRead.mo
        SharedMemoryWrite.mo
    package.mo
    package.order
```

3.4 Implementation on Controller part

The controller has been implemented and tested on Arduino Uno platform. Details about the controller implementation will be described in Chapter 11 i.e. in HIL implementation.

The following steps explain how to use the IPC package to run real time HIL routines. This has been illustrated using a couple of examples. However, before executing any examples, the user is advised to replace the files **librt.so** and **librt.a** in **Resources/Libraries/linux64** with files with the same name in the **/usr/lib/x86_ 64-linux-gnu** folder of their systems.
The ArduinoIPC example will demonstrate basic capabilities of the Shared Memory paradigm being used for HIL. It is assumed that the user is running OpenModelica as root.

- First, change directory to /InterProcessCommunication/Resources/Include
- Next, run gcc -o Serial_SHM Serial_SHM.c -lrt
- Next, Open Arduino IDE and flash ArduIPCBasic.ino on the Arduino.
- Run sudo bash Arduino_ port.sh | ./Serial_ SHM [baudrate of choice] on the terminal
- Set up the experiment on OpenModelica. A minimum of 10 seconds is advisable in this case.
- Go into the Simulation Flags menu in the Simulation setup, and adding a '-rt' flag in the Additional Simulation Flags textbox.
- Execute.

The DC_ Motor_ arduino example will be used to illustrate more detailed usage. It is assumed that the user is running OpenModelica as root and has the Modelica Device Drivers package loaded, if they wish to obtain the reference wave as well. The reference wave in question is being generated on a function generator.

One of the bits of terminology that is being used here is that of the primary and secondary arduinos. The primary arduino is the one with the PID code, and the secondary arduino is the one that simply transmits a copy of the reference wave to OpenModelica.

- First, change directory to /InterProcessCommunication/Resources/Include
- Next, run gcc -o Serial_SHM Serial_SHM.c -lrt
- Next, flash IPC_ PID.ino on the primary Arduino.
- Flash basic_ write.ino on the secondary Arduino.
- If the user is making use of a function generator, connect it to pin A5 for both the arduinos, set a 4V sine wave with a period of 30 seconds or a square pulse with a similar period, on the function generator.

- If the user is not making use of a function generator, comment out line 6 on basic_ write.ino and uncomment line 7.
- Run sudo bash Arduino_ port.sh | ./Serial_ SHM [baudrate of choice] on the terminal
- Set up the experiment on OpenModelica. A minimum of 60 seconds is advisable in this case.
- Go into the Simulation Flags menu in the Simulation setup, and adding a '-rt' flag in the Additional Simulation Flags textbox.
- Execute.

Here the problem involves building a HIL simulation with a DC motor as in Figure:3.3 as a software plant, and a proportional control algorithm running on the embedded hardware. The process is expected to run in real time which means that a delay should not be used. The problem of HIL for DC motor can be elaborated as in Figure:3.2

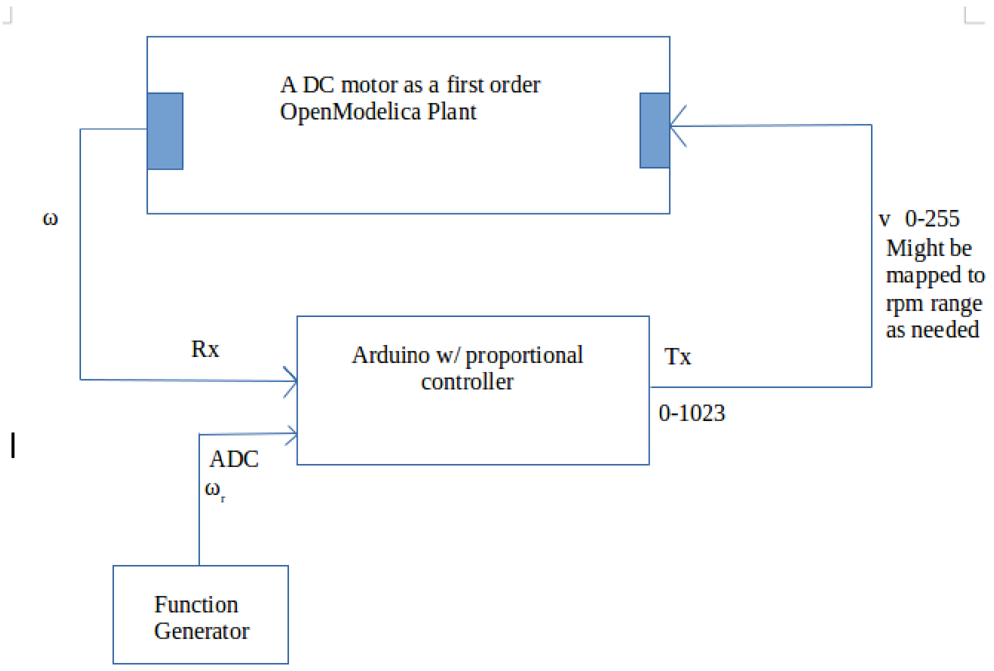

A DC motor as a first order OpenModelica Plant
ω
v 0-255
Might be mapped to rpm range as needed
Rx
Arduino w/ proportional controller
Tx
0-1023
ADC
ω_r
Function Generator

Figure 3.2: HIL Problem Formulation on DC motor

The results obtaiend from HIL simulation of DC motor when PID controller is applied on Arduino by taking virtual pulse signal as a reference as shown in Figure:3.4

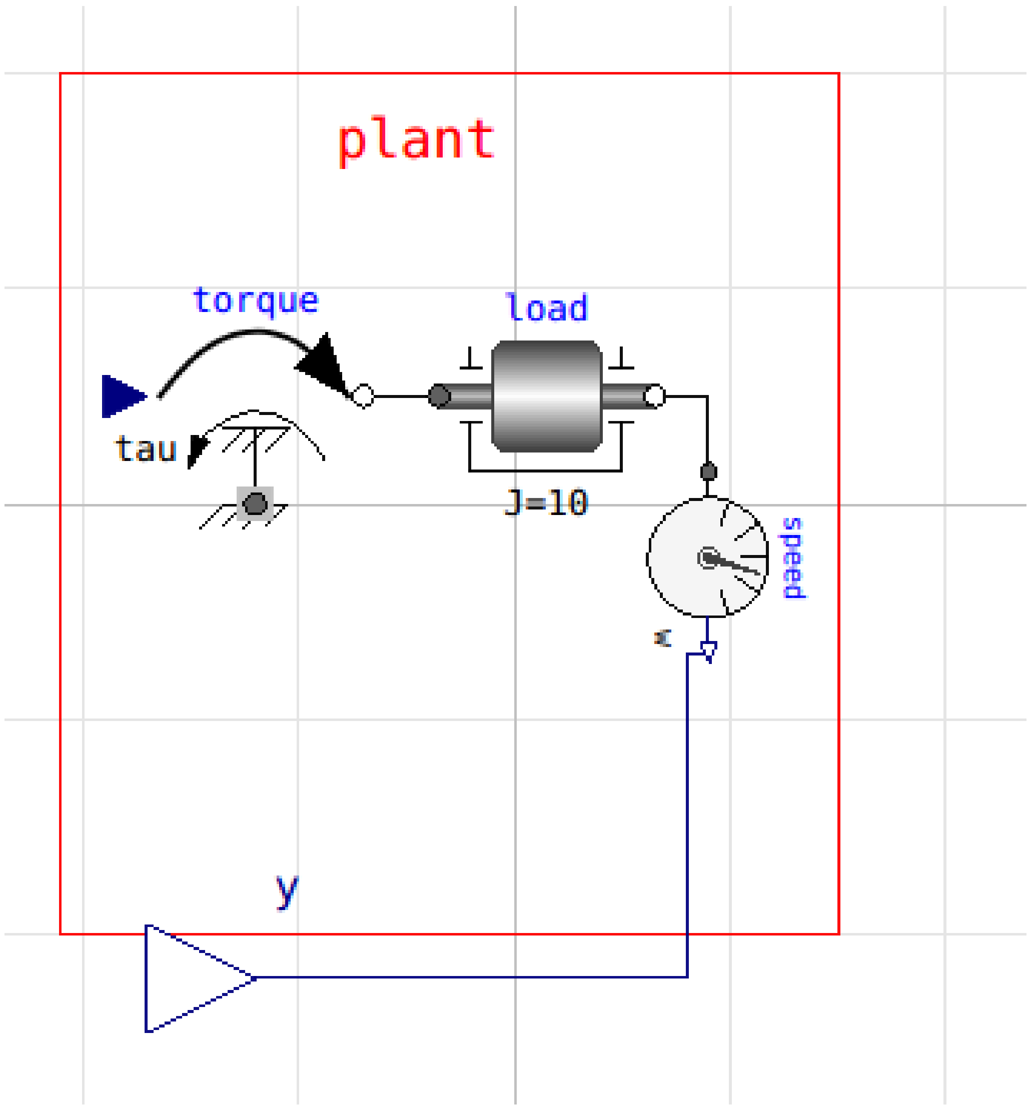


Figure 3.3: IPC: HIL DC Motor model in OpenModelica

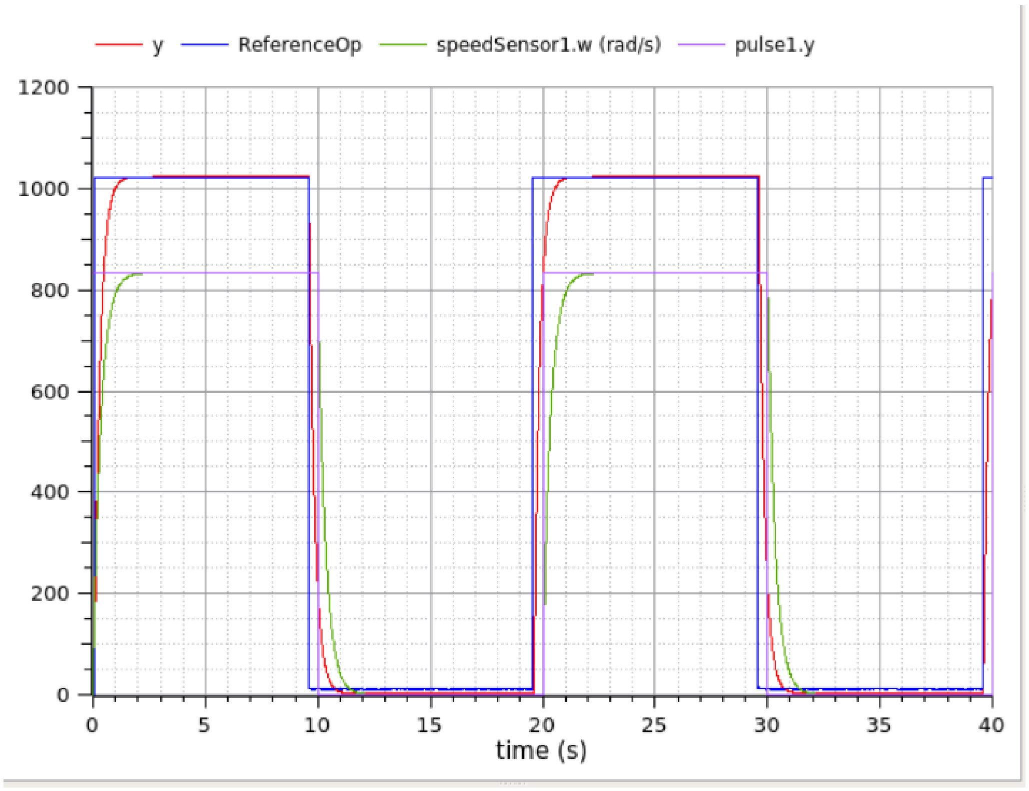


Figure 3.4: IPC HIL: PID on DC Motor with Pulse as reference

3.4.1 Issues

Though this method of HIL simulation worked on almost all the embedded platforms but still it has some issues as listed below:

- Depends on **librt** file of each system which is different for each & every system even if for same operating system
- Cumbersome as number of steps to be followed to execute one model

Chapter 4

Firmata protocol for Interfacing ARM Cortex-M4 series of controllers

4.1 Motivation

To reduce the bottlenecks in IPC, and to support ARM Cortex Series of controllers firmata protocol was adapted.

4.2 Algorithm

This protocol uses MIDI (Musical Interface Digital Interface) [3] is a communication protocol that provides interface of musical instruments to host computers. Though it uses MIDI message format but it doesn't use whole of the protocol. Here sysex (System Exclusive) messages are being used to define set of core and optional features. The package developed (OpenModelicaEmbedded) has several components like micro-controller boards, Digital/Analog Pins, etc. which the user will have to use in the model to make it interact with connected hardware device. These components make call to external C functions present in the library provided in Library directory. Those functions using serial communication communicate with the connected device. This source file will remain same irrespective of the connected hardware device and platform used (Windows, Linux, Mac).

The connected hardware device uses **Firmata** protocol as in Figure:4.1 to communicate with OpenModelica. The source code implementing Firmata protocol on hardware will vary depending on the language/IDE used by that hardware/micro-controller, but the underlying protocol remains the same.

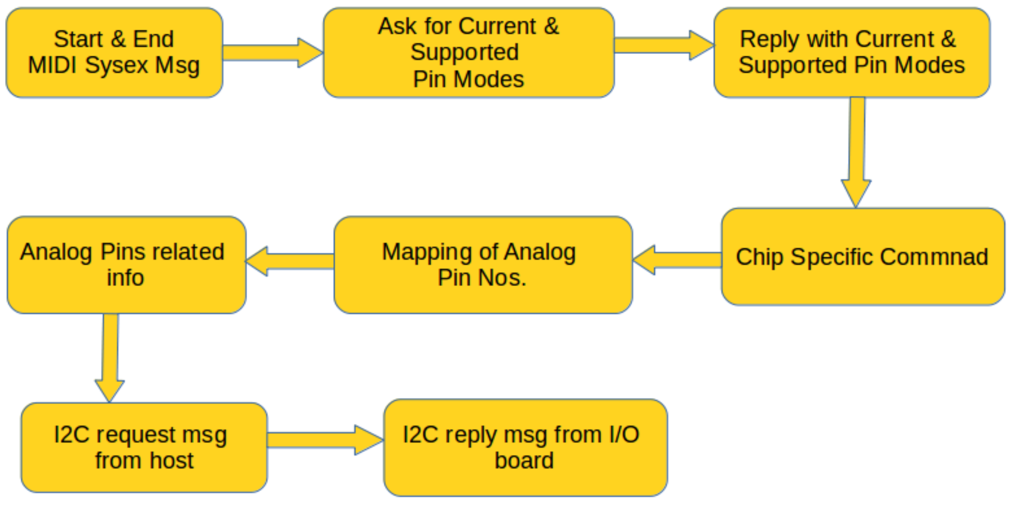

Start & End MIDI Sysex Msg
Ask for Current & Supported Pin Modes
Reply with Current & Supported Pin Modes
Chip Specific Commnad
Mapping of Analog Pin Nos.
Analog Pins related info
I2C request msg from host
I2C reply msg from I/O board

Figure 4.1: Firmata Implementation

4.3 Implementation on OpenModelica part

1. Once you have installed OpenModelica, launch OMEdit and open the OpenModelicaEmbedded package.

2. To use the above package you will also need to load Modelica_DeviceDrivers package. The 'synchronizeRealtime' block present in this package is used to make the simulation of models in real-time. All it does is that, it maps the time interval provided by you before simulation with clock of your PC.

3. The components provided in this package are:

   (a) Pins: It contains Analog input, Analog output, Digital input, Digital output and Servo pins to perform corresponding function in model.

   (b) Boards: Any of the provided board can be used depending on the one you are using, else use the 'customBoard' provided and vary it's parameters to match the configuration of the development board you are using.

4. Take a look at the examples provided along with the package to understand the basics structure of a model. Each model has 'Board' block which represents the development board used. This block when added to a model, on simulation calls a couple of functions present in 'Internal > ExternalFunctions' which sets the initialization parameters for communication like PORT, BAUD rate, etc.

5. These modelica functions present in 'ExternalFunctions' then call external C functions which perform the actual task that the function is supposed to do.

6. These external C functions are bundled together and provided in the form of libraries. The Libraries used will be '*.dll' in case of WindowsOS and '*.so' in case of Linux.

7. After adding a board to your model add pins using blocks provided for the same. If you want to send some data from OpenModelica to connected micro-controller then use Analog/Digital Output Pin, and vice versa. Use Analog Pin while working with real data and Digital pin while working with Boolean.

8. These pin blocks again call functions in similar manner to either send or receive data.

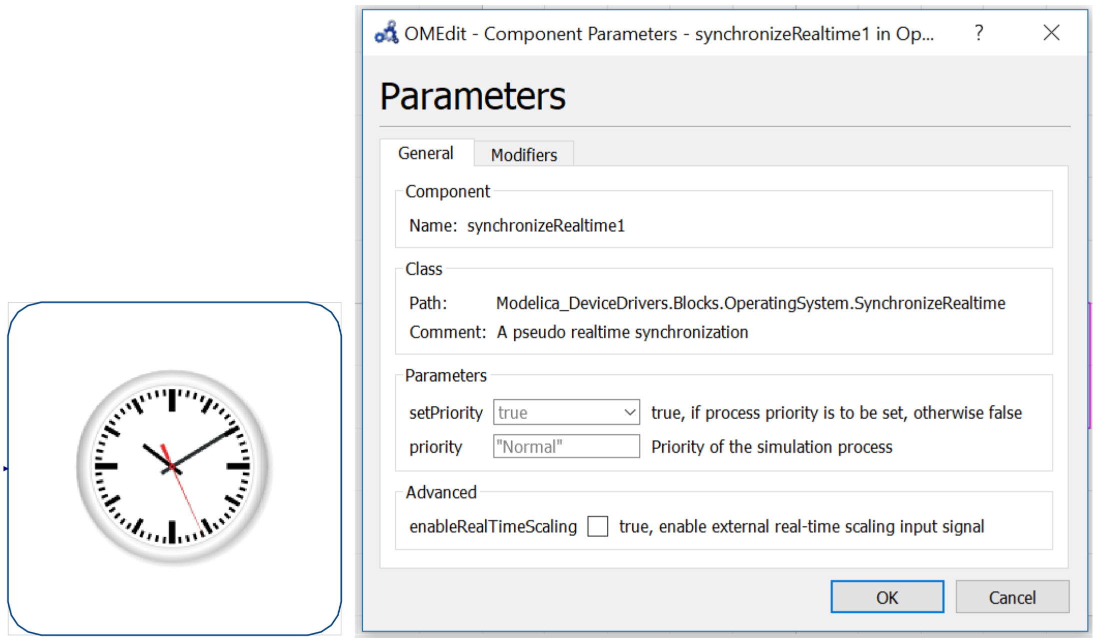
9. Once your model is ready and check is successful, upload appropriate Firmware on microcontroller board connected.

10. The Firmwares for Arduino and Tiva C boards have been provided along with the package. Open Arduino IDE if using Arduino board and Energia IDE if using Tiva C board and upload corresponding firmware on board.

11. The Firmware implements Firmata protocol to establish communication with OpenModelica.

4.3.1 Functionalities added to OpenModelica

- **SynchronizeRealTime Block:** This block is a part of Modelica_DeviceDrivers library used for real-time simulation of the model, i.e., this block synchronizes simulation time of the process to real- time clock of the operating system. Without this block, the models designed using this package will not be able to give proper real-time output. This block works at five different priority levels which can be changed in Parameters dialog box by double-clicking on the block as in Figure:4.2


OMEdit - Component Parameters - synchronizeRealtime1 in Op...

Parameters

General | Modifiers

Component
Name: synchronizeRealtime1

Class
Path: Modelica_DeviceDrivers.Blocks.OperatingSystem.SynchronizeRealtime
Comment: A pseudo realtime synchronization

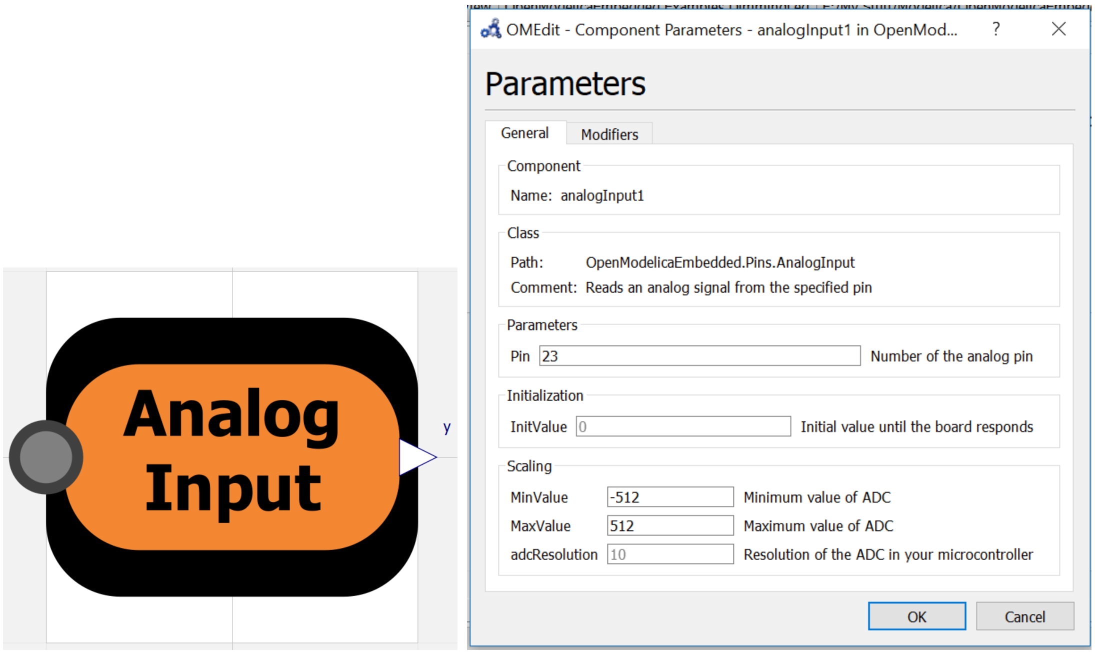
Parameters
setPriority: true — true, if process priority is to be set, otherwise false
priority: "Normal" — Priority of the simulation process

Advanced
enableRealTimeScaling ☐ true, enable external real-time scaling input signal

Figure 4.2: SymchronizeRealTime block

- **Pins:** This package contains blocks which define input and output pins of the board to which our hardware can be connected. These pin components define the properties and working of the pins used in the hardware.

- **AnalogInput:** It reads an analog signal from the specified pin. This component uses the function 'analogRead'. It takes minimum and maximum values of the signal as parameter (default values being 0 and 1 respectively) and gives output depending on the size of ADC (analog to digital converter) 12-bit for Tiva C series TM4C123G board.



OMEdit - Component Parameters - analogInput1 in OpenMod...

Parameters

Component
Name: analogInput1

Class
Path: OpenModelicaEmbedded.Pins.AnalogInput
Comment: Reads an analog signal from the specified pin

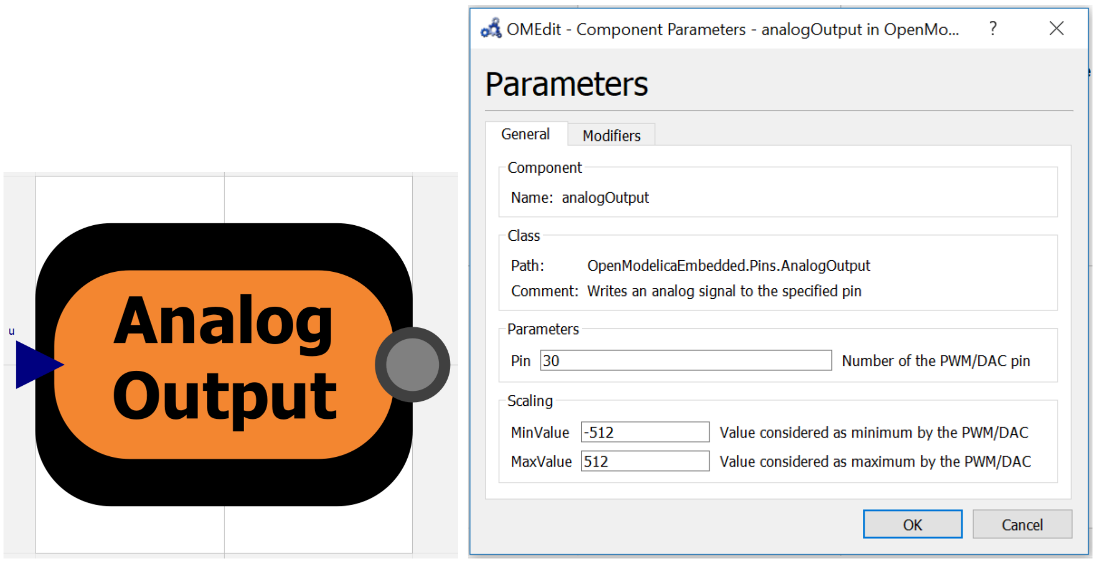
Parameters
Pin 23 Number of the analog pin

Initialization
InitValue 0 Initial value until the board responds

Scaling
MinValue -512 Minimum value of ADC
MaxValue 512 Maximum value of ADC
adcResolution 10 Resolution of the ADC in your microcontroller

Figure 4.3: AnalogInput block

- **AnalogOutput:** It writes analog value (PWM wave) to the specified pin. This component uses the function 'analogWrite'. It takes minimum and maximum values of the signal as parameter (default values being 0 and 1 respectively) and gives output depending on the size of ADC.



OMEdit - Component Parameters - analogOutput in OpenMo... ? ×

Parameters

General | Modifiers

Component
Name: analogOutput

Class
Path: OpenModelicaEmbedded.Pins.AnalogOutput
Comment: Writes an analog signal to the specified pin

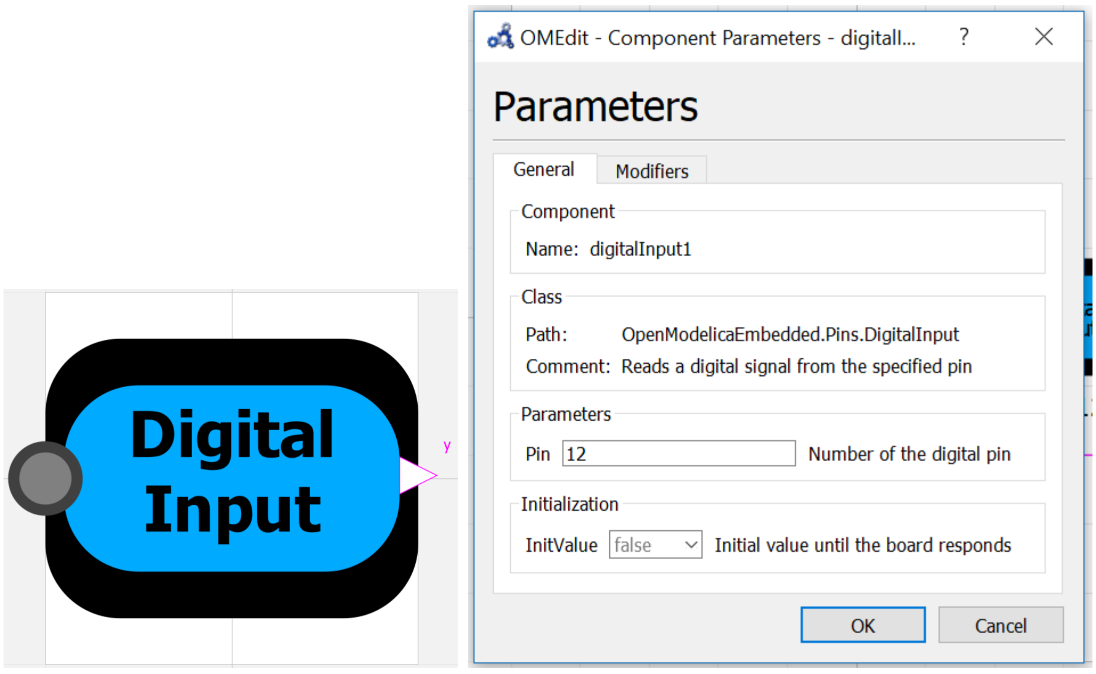
Parameters
Pin 30 Number of the PWM/DAC pin

Scaling
MinValue -512 Value considered as minimum by the PWM/DAC
MaxValue 512 Value considered as maximum by the PWM/DAC

OK | Cancel

Figure 4.4: AnalogOutput block

- **DigitalInput:** It reads an digital signal from the specified pin. This component uses the function 'digitalRead'. It only takes boolean signals.



OMEdit - Component Parameters - digitalI... ? ×

Parameters

General | Modifiers

Component
Name: digitalInput1

Class
Path: OpenModelicaEmbedded.Pins.DigitalInput
Comment: Reads a digital signal from the specified pin

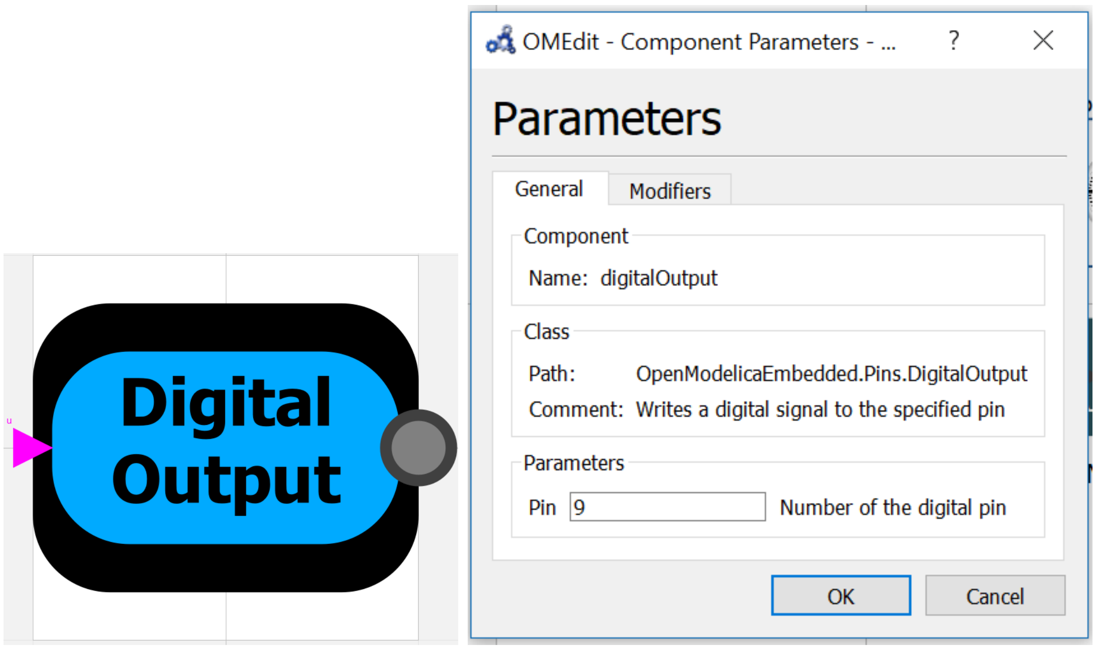
Parameters
Pin 12 Number of the digital pin

Initialization
InitValue false Initial value until the board responds

OK | Cancel

Figure 4.5: DigitalInput block

- **DigitalOutput:** It writes digital value to the specified pin. This component uses the function 'digitalWrite'. It only takes boolean signals.



Component

Name: digitalOutput

Class

Path: OpenModelicaEmbedded.Pins.DigitalOutput

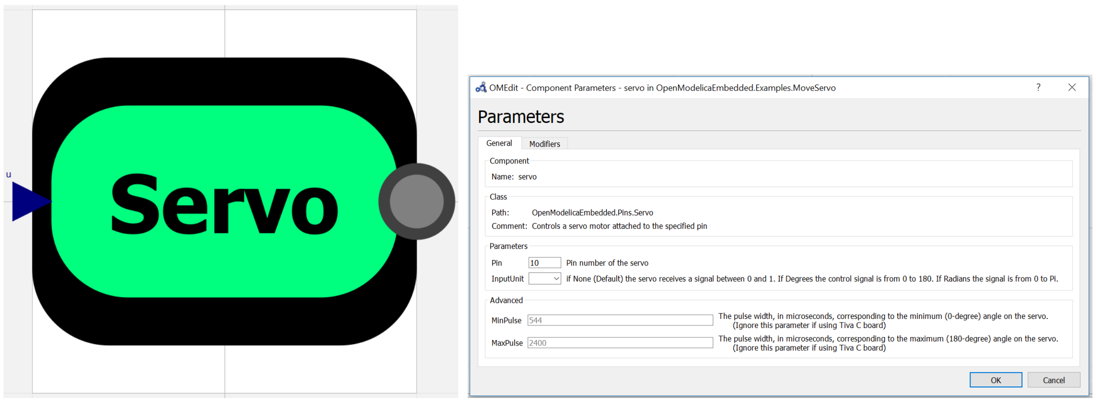
Comment: Writes a digital signal to the specified pin

Parameters

Pin 9 Number of the digital pin

Figure 4.6: DigitalOutput block

- **Servo:** It controls a servo motor attached to the specified pin. This component uses the 'Servo' library. By default, the range goes from 0 to 1, which corresponds to 0 to 180 degrees. If you want to input values in degrees or radians, you can change the parameter 'InputUnit' to 'Degrees' or 'Radians'.


u
Servo
OMEdit - Component Parameters - servo in OpenModelicaEmbedded.Examples.MoveServo
Parameters
General
Modifiers
OK
Cancel

Component

Name: servo

Class

Path: OpenModelicaEmbedded.Pins.Servo

Comment: Controls a servo motor attached to the specified pin

Parameters

Pin 10 Pin number of the servo

InputUnit if None (Default) the servo receives a signal between 0 and 1. If Degrees the control signal is from 0 to 180. If Radians the signal is from 0 to Pi.

Advanced

MinPulse 544 The pulse width, in microseconds, corresponding to the minimum (0-degree) angle on the servo. (Ignore this parameter if using Tiva C board)

MaxPulse 2400 The pulse width, in microseconds, corresponding to the maximum (180-degree) angle on the servo. (Ignore this parameter if using Tiva C board)

Figure 4.7: Servo block

This package contains block components which enable connection with different firmata boards. These components take serial port used for connection as parameter.

- **StandardFirmata:** Connects only to compatible boards.

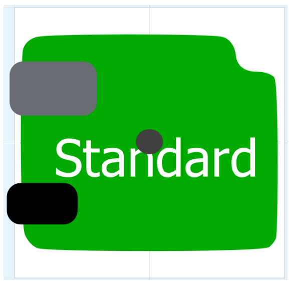


Figure 4.8: StandardFirmata block

- **CustomFirmata:** Supports any board firmata.

Figure 4.9: CustomFirmata block

- **customBoard:** Takes name of the board also as parameter and can be used to connect any board supporting firmata.

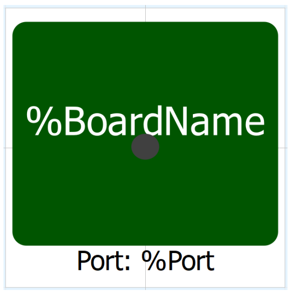


Figure 4.10: customBoard block

4.4 Implementation on Controller part

The Tiva C series Launchpad Evaluation board (EK-TM4C123GXL) is low cost ARM-Cortex-M4F based micro-controller. The board contains 40 I/O pins, two user programmable push buttons, an RGB led and many more features as in Figure:4.11

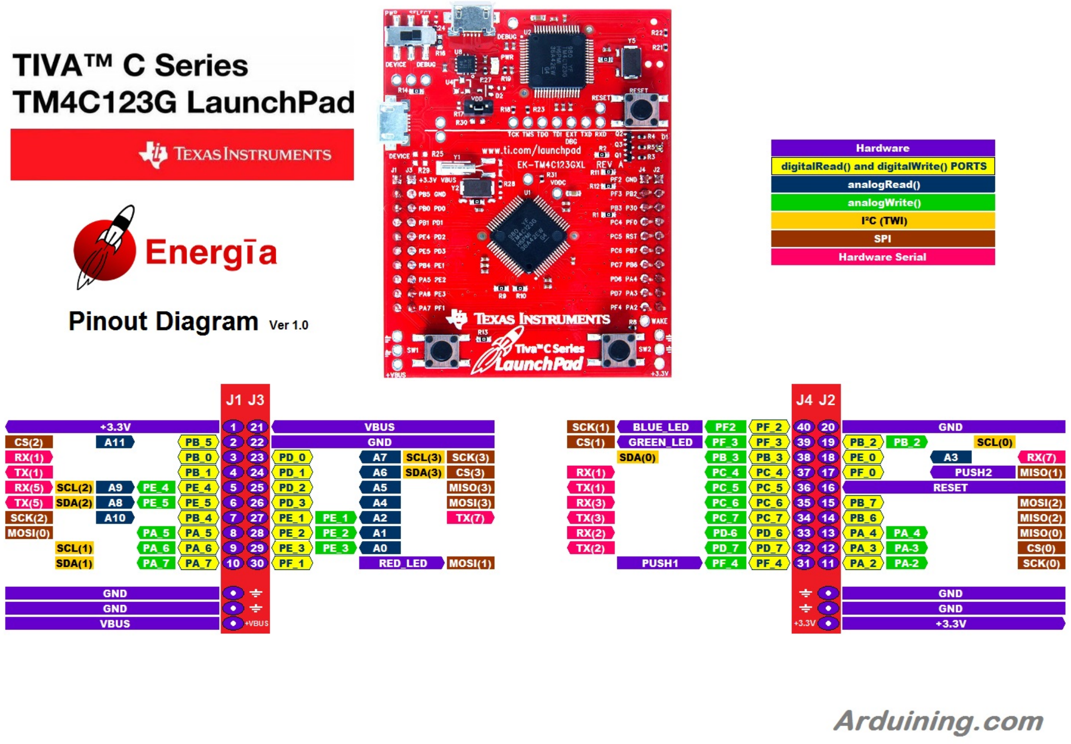


Figure 4.11: Pin Diagram of Tiva C Launchpad

1. Connect the Tiva C board to the computer using a USB cable.
2. Open Energia IDE.
3. In Tools Menu, select Board →Tiva C and Port as the available serial port to which Arduino is connected.
4. If Tiva C board is not present, then click on Board Manager, type Tiva C in search bar and then click on Install to install board library, then apply Step 3.
5. In Sketch menu, Select Include Library →Add .zip library and add the zip file provided in the Firmware folder. Then open StandardFirmata sketch: File →Examples →StandardFirmata.
   OR
   Click File →Open and browse OpenModelicaEmbedded →Firmware →Tiva C →StandardFirmata and open StandardFiramata.ino.
6. Upload the sketch to the board.

The process of interfacing with OpenModelica happens in the following steps

1. Upload StandardFirmata sketch to the Tiva C board.

2. Open package.mo from OpenModelicaEmbedded package, also open package.mo file from Modelica_DeviceDrivers library.
3. In OpenModelicaEmbedded library, open TivaC_Examples package.
4. In Diagram view, change the port name for the board component to the port to which board is connected by double-clicking on it.
5. Simulate the example model.

Examples for TIVA C Controller
TIVA C Examples package consists of example models designed for specifically to work with TIVA C series board. In order to work with these examples, double-click of board block in the Diagram view and change port name to the port to which board is connected.

The following is an example to turn on the blue led indefinitely as in Figure:4.12. Double clicking each block opens the parameter window for it. Change the parameters according to the following image.

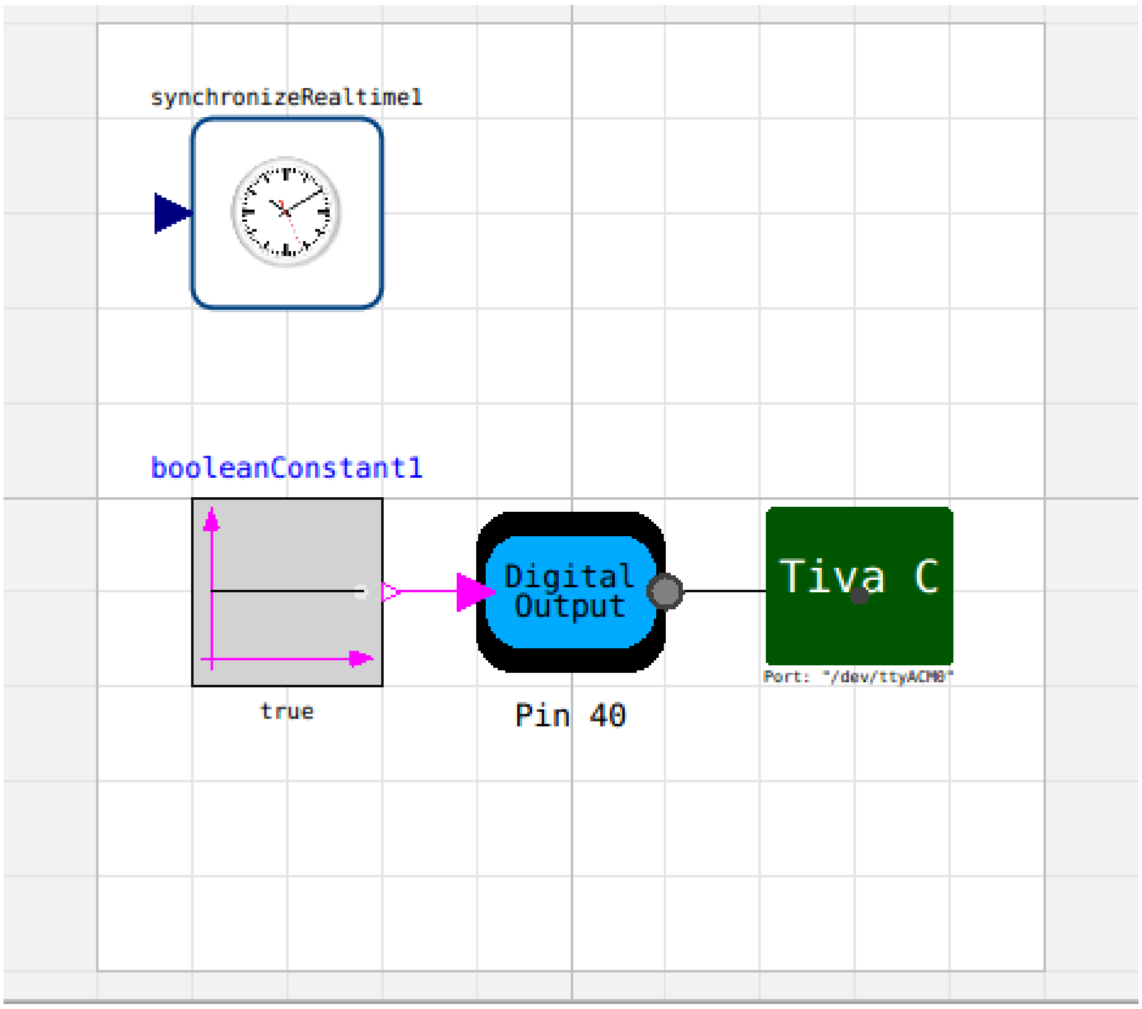


Figure 4.12: TIVA C Led Example

The following example is to read the status of the pushbutton and display it on the serial monitor.

In this model in Figure:4.13, a BooleanValue block is used to show boolean value coming from the digital input pin of TIVA C on Simulation Output. The block BooleanValue can be found at Modelica.Blocks.Interaction.Show.BooleanValue.

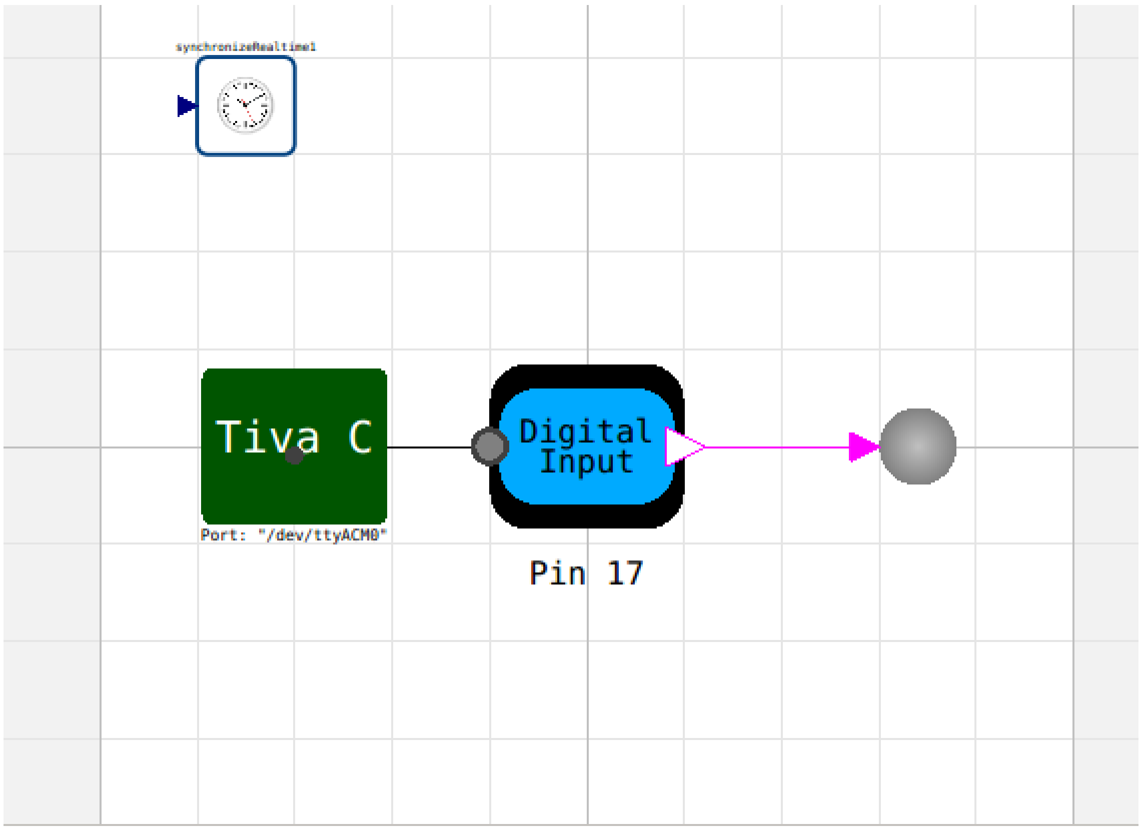


Figure 4.13: TIVA C Push Button Example

Turning the blue LED on and off according to the values of LDR (Light Dependent Resistor).

In the model as in Figure:4.14, two blocks have been used namely Less and Constant.

Less block takes two inputs and gives one output according to the values of input. For example in the case below, when value from pin 19 is less than k = 300, output is true (or 1) and when value from pin 19 is greater than equal to k=300, its output is false (or 0). The block can be found at Modelica.Blocks.Logical.Less.

Constant block provides with a constant value which can be set by user. The block can be found at Modelica.Blocks.Sources.Constant.

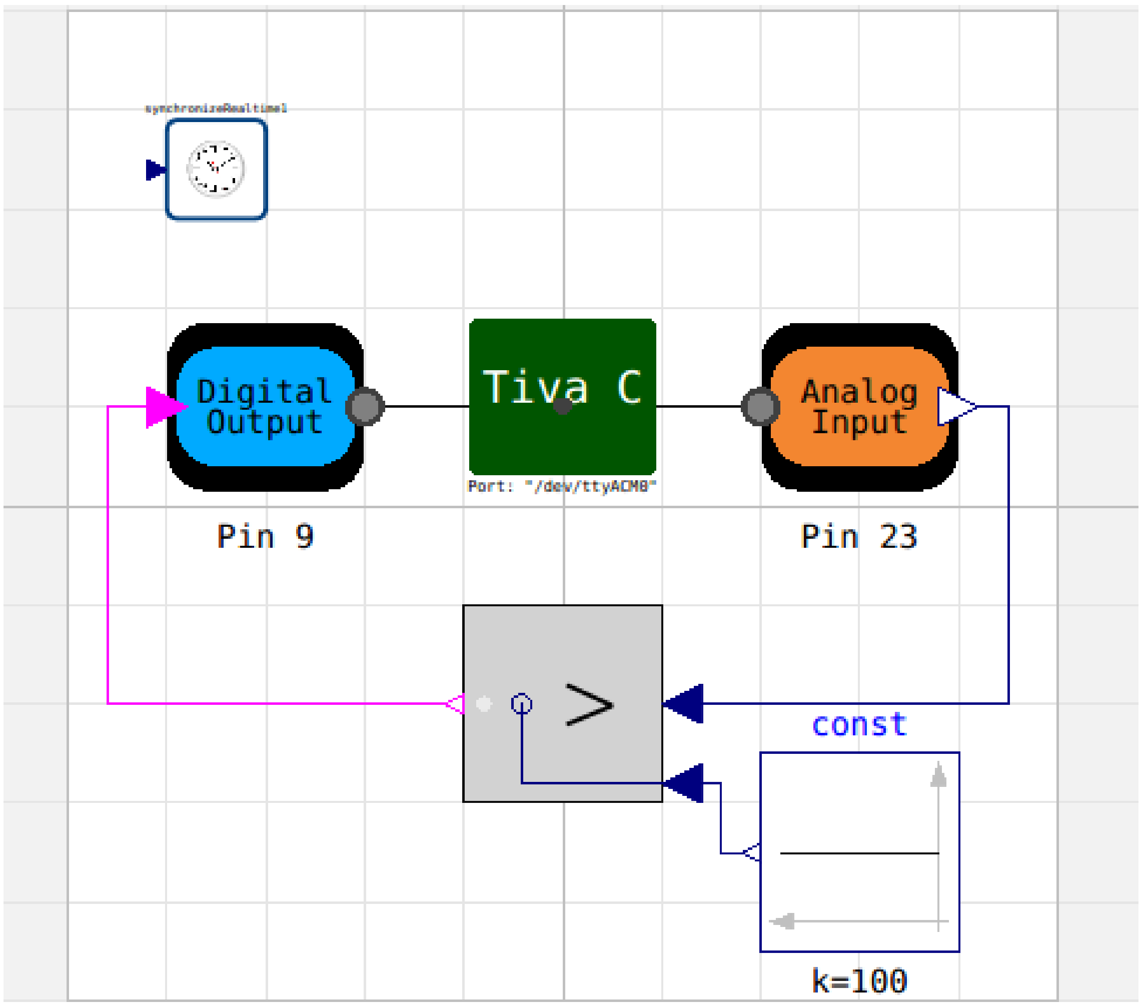
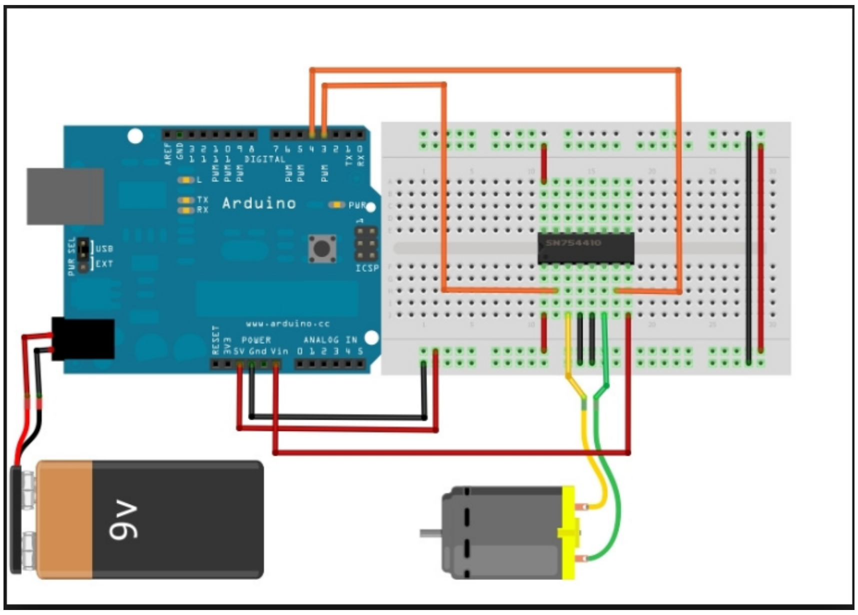


Figure 4.14: TIVA C LDR Example

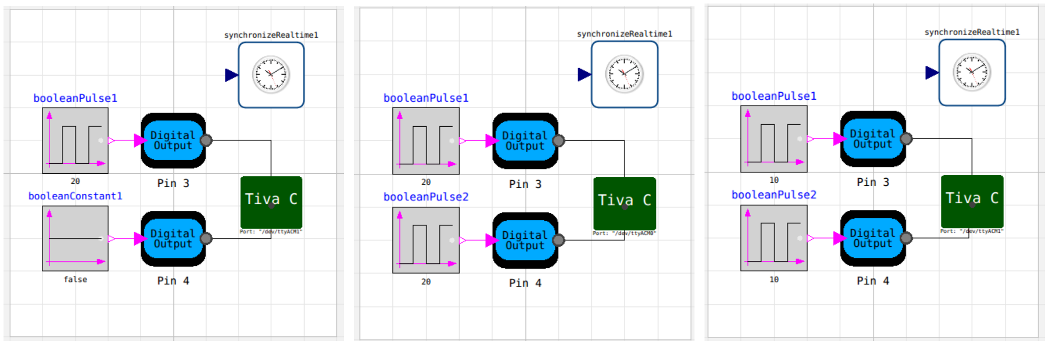
The following example is of rotating the DC motor in both directions. As visible in the model in Figure:4.15, 2 pulse blocks are used to manage this.

A pulse block generates pulse signals of real value. It's amplitude, duty cycle, time period, start time can be varied through changing amplitude, width, period, startTime respectively in the parameter window of the pulse. The block can be found at Modelica.Blocks.Sources.Pulse.

Double clicking each block opens the parameter window for it. Change the parameters according to the following image.



(a) Connections for dc motor



(b) Model for rotating dc motor in both directions

Figure 4.15: TIVA DC Motor Example

Rotating the servo in increments.

The model as in Figure:4.16 contains blocks like Product, RealToInteger, IntegerToReal, Constant and Ramp. The Ramp block gives a strictly increasing value. On using RealToInteger block on the output, it converts it to step function. Now as the Product block accepts 2 input in real format only, there was a need to convert the value back to real using IntegerToReal block.

In Servo pin, set InputUnit to OpenModelicaEmbedded.Internal.Types.ServoUnit.None.

As can be seen from data sheet in Figure:4.17, SG90 has a duty cycle of 5-10% where if it is 5%, the position of motor is -90 degrees and if 10%, it is +90 degrees. So as we were simulating for 10 seconds, MinPulse was 0.5 sec and MaxPulse was 1 sec in Servo pin Parameters.

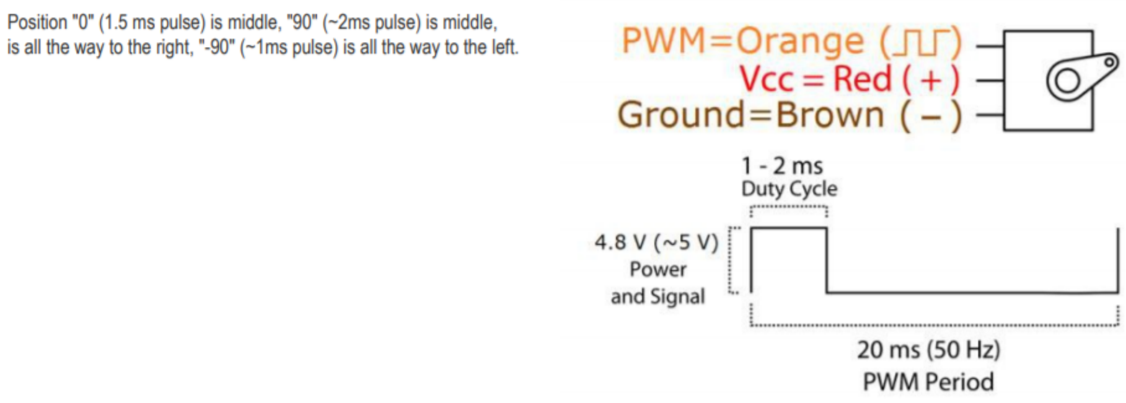

Position "0" (1.5 ms pulse) is middle, "90" (~2ms pulse) is middle, is all the way to the right, "-90" (~1ms pulse) is all the way to the left.

Figure 4.16: Data Sheet for Servo Motor SG90

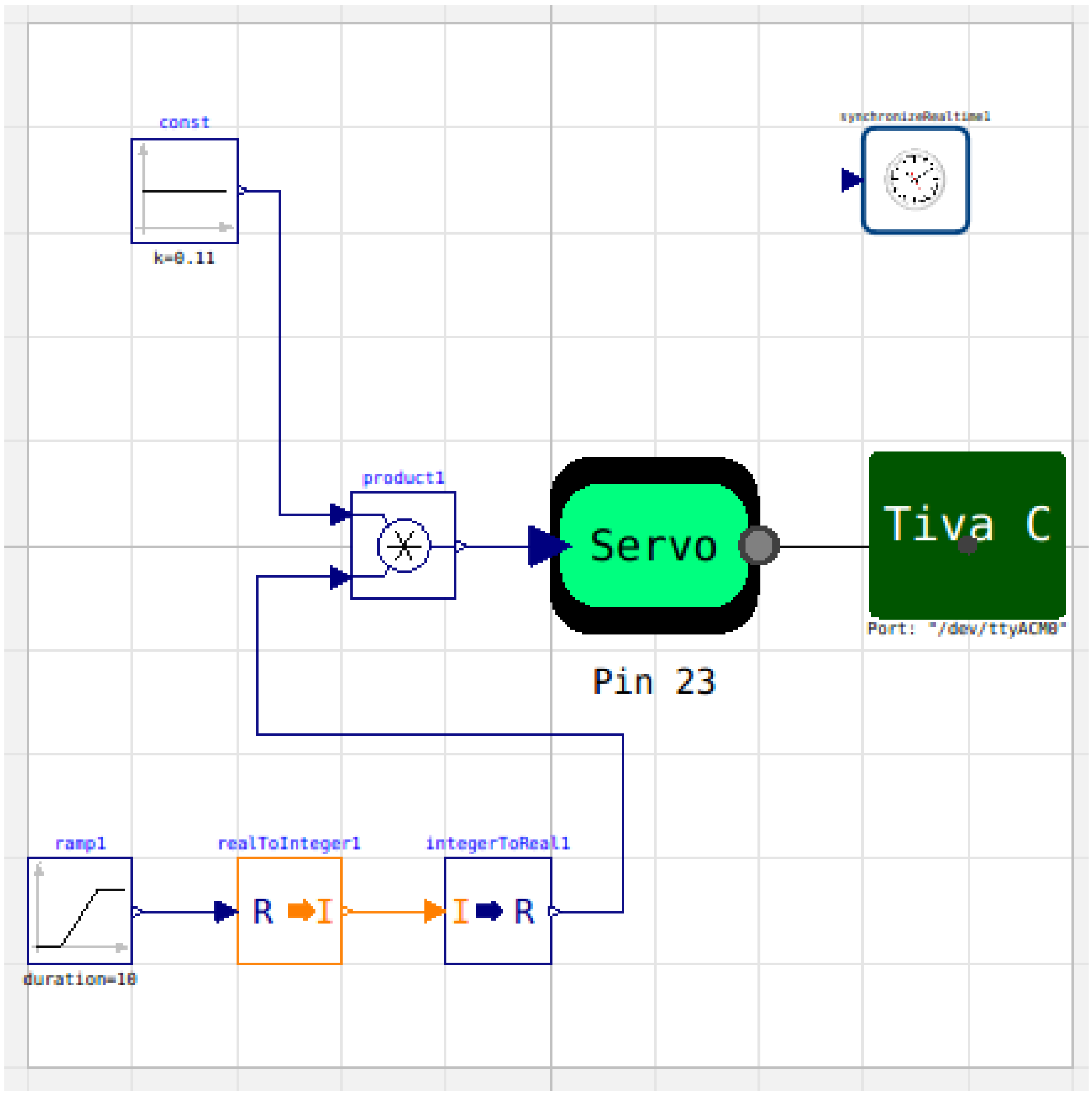


Figure 4.17: TIVA C Servo Motor Example

Chapter 5

Open Platform Communications Unified Architecture (OPC UA)

5.1 Motivation

Now-a-days system designs and information sharing has become border-less of any specific company or any plant. Many a companies are working in-sync without any boundaries together on different common projects and products. Due to the huge demand of integration and interconnecting different systems, there has to be a common standard for their interoperability. The common standard of different information systems reduces cost and time of integration. With standardization, there is also a possibility to create common adoption between different kind of systems. OPC Foundation [6] solves the need of standardization. Initially, the OPC started for OLE for Process Control. But OLE itself is proprietary technology called Object Linking and Embedding. This technology is used to create references between the data objects specifically in Windows OS. Microsoft later published SDK for this technology, which leads to creation of the OPC. OPC Foundation decided to redesign OPC components and technologies with modern, vendor independent solutions. The new specification is called OPC Unified Architecture (OPC UA). Nowadays the OPC means Openness, Productivity and Collaboration. Currently, OPC is the communication standard in automation technology. Migration to OPC UA is needed to increase possible types of the integration solutions for which OPC can be used.

5.2 Technical Overview

5.2.1 Classic OPC

It's an inherited technology and based on a technology from Microsoft which is called COM/DCOM i.e. distributed object model. It was started around twenty years back to connect the process devices in the same way as the printer. The idea is to have a driver for the device and the driver is talking through the proprietary protocol to the industrial device. Also, all the windows application should be able to communicate with the device through the driver. There is facility to

1. Data Access
2. Historical Data Access
3. Alarms & Events

Based on this technology, new requirements are coming up, so the new generation technology OPC UA was developed.

5.2.2 OPC UA

On one side, there is wide adoption of OPC classic but there is a need to use OPC as a common system interface and to do that there is need of something we call it scalability as shown in Figure 5.1. That means the technology can be used on very small embedded devices but the same technology can also be used on larger systems like classic SCADA systems etc. In addition to that scalability, the communication between distributed systems should be platform independent. That is why we need a technology that can be implemented on any operating systems not specific to Microsoft. So, it's not bind to COM/DCOM. Hence, OPC UA has a new platform independent application and we can implement it on all kinds of operating systems. In industry, there is need of redundancy and very fault tolerant and robust systems and also they need to run 24/7 and of course performance isn't an option. It needs very high performance to transfer all the data coming from process devices. Also, when security comes into picture, it becomes more important. It needs to secure the data, needs to sign the data, also needs to authenticate the data so that only the authorized person can only visualize the data. Also, it needs to cross firewall boundaries because there is a need to interconnect in large networks and even over the Internet. In addition to the protocol specific details, there is also a need of data modelling capabilities. This is one of the basic key features of OPC Unified Architecture. It is needed to describe the data that is inside the machinery. So, type of the system need to be described. Also, complex data structure that needs to be transfered. And to describe all these information, there is need to have a meta data model. This meta data model is used to describe the semantics and all the relationships between the data inside the particular device. This concept is applicable to all kinds of data.

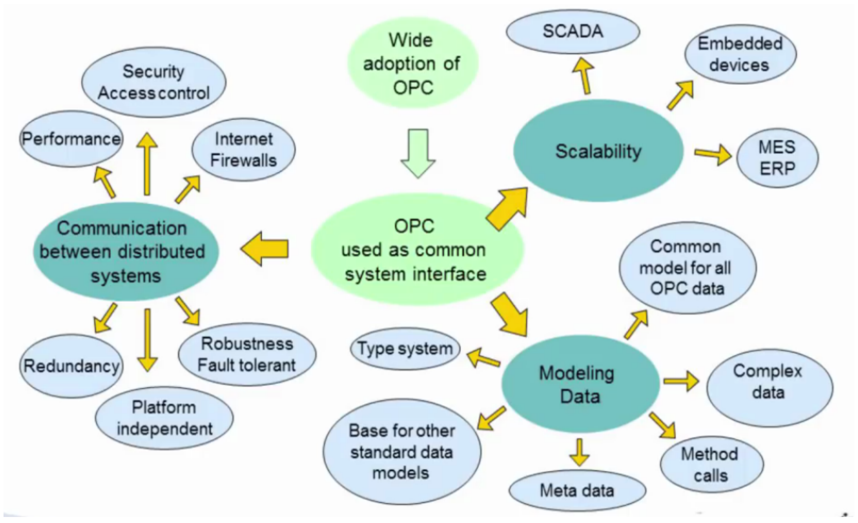


Figure 5.1: Scalability of OPC UA

5.3 Basic Building Blocks of OPC UA

The technology started in 2003 and till 2006 was the definition phase writing the specifications. Then there came the verification phase when actually the community tried to implement what was specified and to see what was doable by coding standards. And finally the specifications were released in 2009. During this time only the first products were showed up in the market. Now, OPC UA has become an IEC standard which is an international standard and the release happened in 2010.

By basic oveview, the OPC UA is the classic OPC standards and it has classic OPC features with additional features such as

1. Platform Independence
2. Standard internet and IP Based standard Protocol
3. Built in Security features
4. Generic object model
5. Extensible type system
6. Scalability through profiles

The object model and the type system are the basic building blocks as shown Figure 5.2. Data modeling and it's capabilities are best described as similar to object oriented programming. So, there is an OPC UA object which is the representation of a data point or some processed value. It has a variable, it can have methods and it can also trigger events. So, it's quite a complex object.

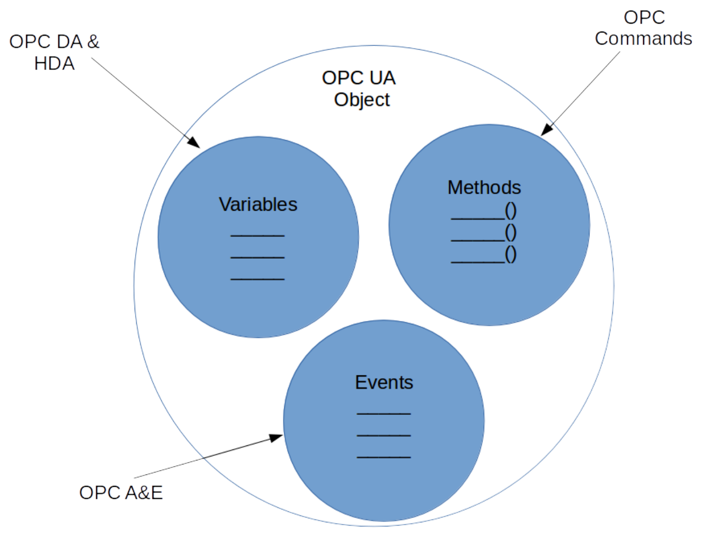


Figure 5.2: OPC UA Object Model

The OPC variables have OPC data access and historical data access. The events are known as OPC alarms & events. The commands are the method invocation of those kind of objects.

Everything in OPC UA is a node. So, the address of a OPC server gives the full description of the information that is inside the device. All this is described with the nodes and there are eight node classes defined by the OPC foundation like the base node classes and from this everything else is derived. In addition to nodes, the base nodes and all the attributes, this can be extended to systems if there occurs any special requirements. There are instances of objects, variables, methods, and there are object types. All the objects and nodes inside the address space of an OPC server are interconnected with references. So, references are like pointers pointing to another node. And with those references, we can build all the relationships between different nodes within the address space.

The next building block is transport. There are different transport bindings. So, OPC UA can be transported over different protocols. And one of the transport is mandatory that every OPC UA server must implement i.e. Optimized OPC UA binary. This is using TCP/IP as a base so it's sitting on top of TCP/IP and then it defines a UA TCP protocol. On top of that we have a security layer which is known as UA secure compenastion layer. It's the layer that encrypts the message and on top of it is encoding which is a UA binary encoding as a special encoding which is very optimal for the use of OPC. This is known as message security because the message content is secured.

5.4 Protocol Implementation - OPC UA Stacks

In the protocol stack, in the Figure: 5.3 there is a client application which calls an API on the stack at the API call and the stack is implemented in different programming languages. When there is an API call, the client application calls into the stack and then the message structure gets encoded and the encoded message gets secured and the secured message gets transported over the wire. On the other side the message goes up the stack through different layers. The transport header is removed and the message is decrypted and the raw message sturcture pop's up on the server API. If the server answers then it goes back to the client and so on. This is the concept of message transfering between the server and client and back. And user doesn't have to implement the stack, it's already there. User just have to integrate it to their applications.

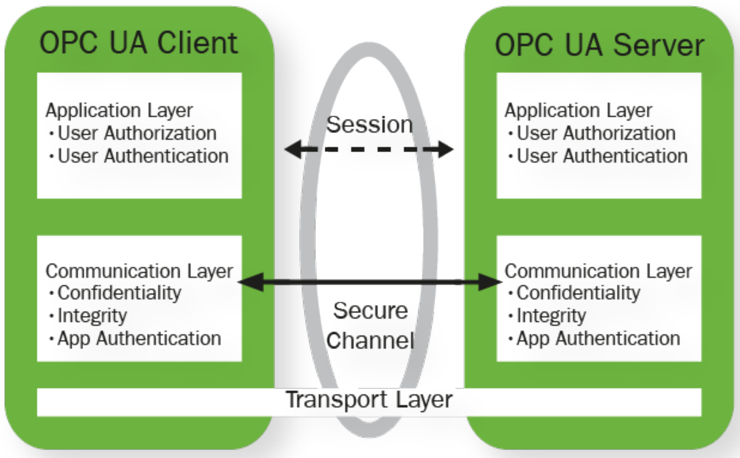


Figure 5.3: OPC UA Stacks

5.5 OPC UA Services - Generic UA API

On top the basic bulding blocks, it has UA based services. It has the following fatures

1. Protocol Independent OPC UA Services
2. Services to
   (a) Discovery Services and Endpoints
   (b) Browse Server Address Space
      i. Instances
      ii. Type Systems
   (c) Read & Write current data
   (d) Read History of Data & Events
   (e) Call Methods
   (f) Subscribe for

        i. Data Changes
        ii. Events
    (g) Create & Delete Nodes & References

3. Generic Services
    (a) No Feature Specific extensions
    (b) Features Added through information Models

5.6 OPC UA built-in Information Models

OPC UA has an information model for data access, alarm, conditions and programs. The model has following features

1. Data Access :-
    (a) Representation of process variables
    (b) e.g. AnalogItemType with unit and range
2. Alarms & conditions :-
    (a) Representation of power alarm systems
    (b) State machines for Alarm States
    (c) Events for State changes
    (d) Methods feedback like Acknowledge
3. Historical Access :- Information about historized data and events
4. Programs :-
    (a) Representation of programs
    (b) Manipulate programs like start, stop
    (c) State of a program execution
    (d) Result data handling

5.7 OPC Companion Model

On top of the OPC information models, there comes the companion models in which organizations use OPC and the data is being described in a companion specifications. Here data is being described in OPC UA and this can be defined already some standard models defined by the OPC. But user can also define their own model directly for any vendor specific model. There were very successful companion models such as PLCopen which defines the way how the PLC data is being exposed in OPC UA object. Also, there is a companion model defined by the Automation group. There is also a specification defined companion model is going on for oil and gas industry.

5.8 Overall OPC UA Model

1. Standard Information Models
    - Use Case Specific Models
    - Industry Specific Models
2. Defintion Based On
    - Collaboration with other models
    - Special use cases and requirements
3. Already Available
    - Device Integration(DI)
    - Anlyzer device Integration(ADI)
    - IEC61131-3(PLCopen)
    - Field Device Integration(FDI)
    - Bulding Automation(BACnet)
4. More Ongoing
    - Oil & Gas subsea (MDIS)
    - Auto Identification (AIM)
    - AutomationML

OPC UA specifications has currently has 14 parts based on OPC technology. These are known as base course specifications.

Chapter 6

OPC UA for interfacing Single Board Computers (SBCs) & OpenModelica

6.1 Motivation

OPC UA can be used to communicate among different platforms such as Windows, Linux, OS X and other Single Board Computers (SBCs) like Raspberry Pi etc. Here OpenModelica being an open-source modeling and simulation tool, it has various block sets for process simulation. Any of the SBCs on which OpenModelica can be installed and can access the GPIO pins will be of great use in Industrial IoT [16] [13] i.e. in cyber- physical production systems [20]. If any analog/digital sensor is interfaced to any of the GPIOs of SBCs while sensor data is made available to a plant model for real-time simulation, it will be of immense value real-time process simulation. As the results obtained are of real-time, that can be directly used for plant setup.

6.2 Architecture

As the OPC UA architecture is platform independent, fetch real-time data and store data history, it will be an ideal platform to access GPIO pin modes from OpenModelica environment. The architecture of the implementation is shown in Figure: 6.1. Here the OPC UA server runs on OpenModelica and the OPC UA client runs which is created by the Python implementation. Once the data from sensors is made available through the serial communication to OPC client, then the OPC server running on OpenModelica can access it and do the computation as defined by the user in the process defined on OpenModelica. The implementation happens in two stages. In the the first stage, it is required to get the IDs of the process, Node and variables.The server implementation architecture is as shown in Figure: 6.2.The Figure: 6.3 shows the Node Ids and variable Ids from the 1st stage implementation. The OpenModelica script for this implementation can be found in Appendix B. The OPC UA python script can be referred in Appendix C. The process ID is shown by the OpenModelica.run field which is here 2 and the variable IDs is given by x and y here as per the modelica script. After getting the IDs the user has to use those specific ID nos to make the data available to OpenModelica workspace. Here, once the node is created, the node objects have methods to read and write node attributes as well as browse or populate the address space. All the child processes in that node will ultimately give access to the variables to which the values from the sensors will be written.

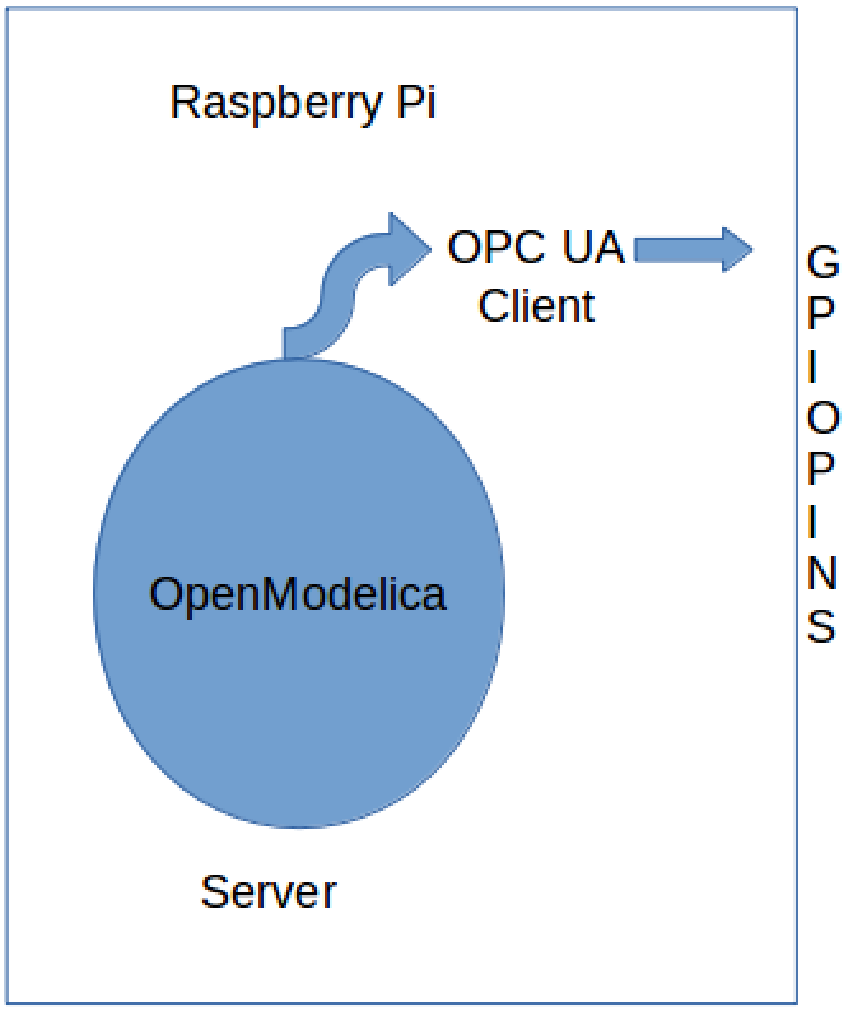


Figure 6.1: Architecture of Raspberry Pi with OpenModelcia

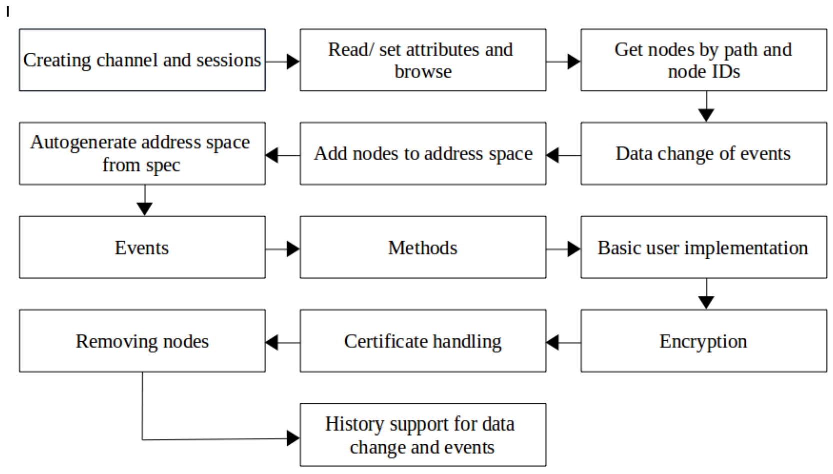
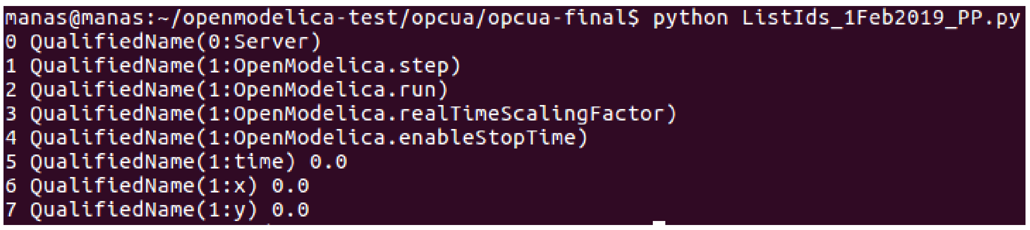

Creating channel and sessions
Read/ set attributes and browse
Get nodes by path and node IDs
Data change of events
Add nodes to address space
Autogenerate address space from spec
Events
Methods
Basic user implementation
Encryption
Certificate handling
Removing nodes
History support for data change and events

Figure 6.2: Architecture of Raspberry Pi with OpenModelcia



```
manas@manas:~/openmodelica-test/opcua/opcua-final$ python ListIds_1Feb2019_PP.py
0 QualifiedName(0:Server)
1 QualifiedName(1:OpenModelica.step)
2 QualifiedName(1:OpenModelica.run)
3 QualifiedName(1:OpenModelica.realTimeScalingFactor)
4 QualifiedName(1:OpenModelica.enableStopTime)
5 QualifiedName(1:time) 0.0
6 QualifiedName(1:x) 0.0
7 QualifiedName(1:y) 0.0
```

Figure 6.3: List Ids of Raspberry Pi with OpenModelica

6.3 Testing & Evaluation on Raspberry Pi

The results in Figure: 6.4 and Figure: 6.6 shows controlling of digital and analog sensor interfaced to Raspberry Pi through OPC UA through OpenModelica. Raspberry Pi doesn't have any ADC and hence it's not feasible to connect any analog sensors source to any of the GPIO pins of RPi. To overcome this issue, one approach could be connecting an external ADC(Analog to Digital Converter) and measure the analog values through it. But a high resolution ADC will be very expensive. So, the other approach is using the ADC of Arduino Uno which has a quite good resolution of 10-bit. It can measure a voltage resolution of 5V/1024 which is around 0.00049V. Also, Arduino can be connected to Rpi through a serial port or USB. Once the real-time sensor values are available through serial communication to Rpi, then OPC UA client on Rpi made these values available to OpenModelica model for further computation. The architecture is being explained in Figure: 6.5

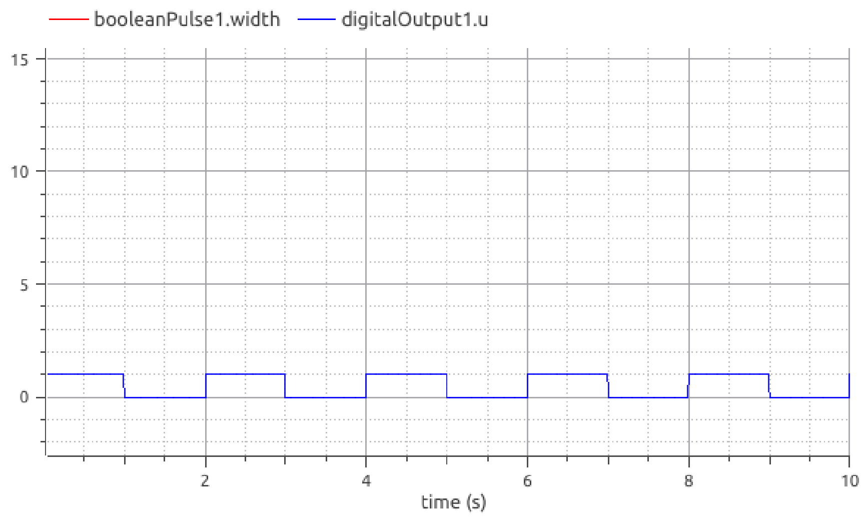


Figure 6.4: Output of Digital sensor with Raspberry Pi with OpenModelica

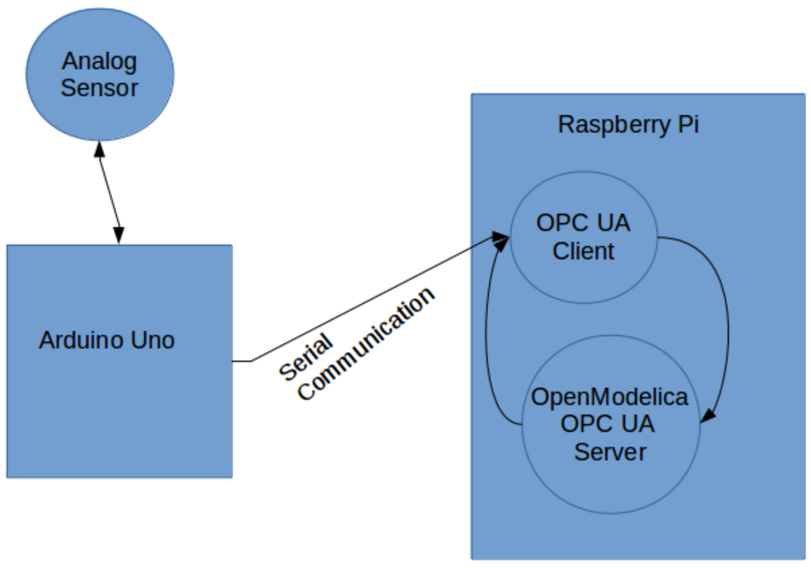


Figure 6.5: Architecture of Analog sensor for Raspberry Pi with OpenModelica

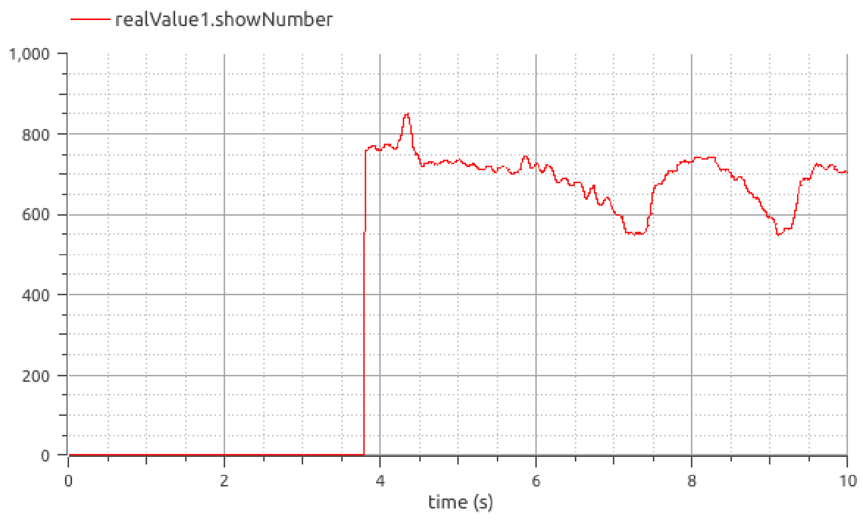
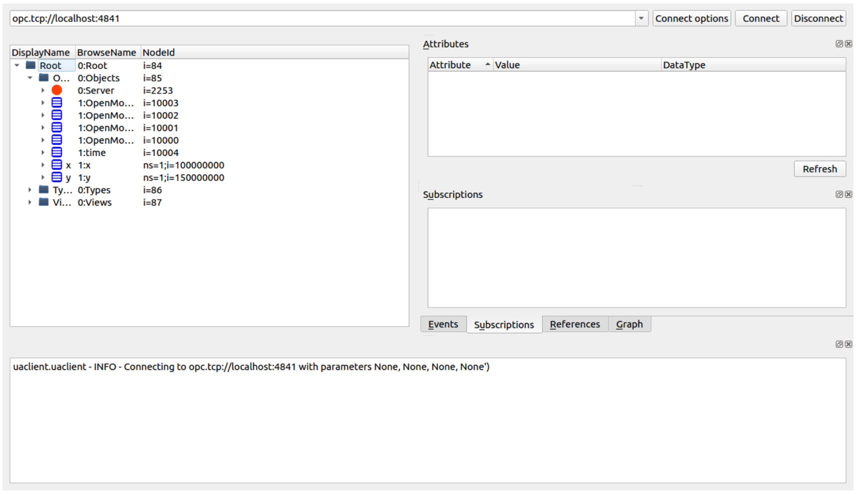


Figure 6.6: Output of Analog sensor with Raspberry Pi with OpenModelica

6.4 Generic Implementation for all Embedded hardwares

OPC UA provides a universal implementation as it is platform independent. Being an user, you just have to define the path to the variable in the algorithm implemented in python. And the path or file structure to the variable can be found by running the opcua-client as shown in Figure: 6.7



DisplayName	BrowseName	NodeId
Root	0:Root	i=84
O…	0:Objects	i=85
	0:Server	i=2253
	1:OpenMo…	i=10003
	1:OpenMo…	i=10002
	1:OpenMo…	i=10001
	1:OpenMo…	i=10000
	1:time	i=10004
x	1:x	ns=1;i=100000000
y	1:y	ns=1;i=150000000
Ty…	0:Types	i=86
Vi…	0:Views	i=87

```
uaclient.uaclient - INFO - Connecting to opc.tcp://localhost:4841 with parameters None, None, None, None')
```

Figure 6.7: Variable Path

Chapter 7

Software In Loop (SIL) simulation with 4DIAC and OpenModelica

7.1 Motivation

Before going for Hardware In Loop (HIL) simulation i.e. to verify the plant model with real-time data input from controller, it's advised to verify the plant characteristics by Software In Loop (SIL) simulation with controller on 4DAIC and plant model on OpenModelica. SIL is a method for software based evaluation of simulation characteristics of plant model. A plant model defined in OpenModelica can be evaluated under simulated input conditions from another software entity and for the purpose of my thesis, it's an open-source framework based on IEC 61149 standard and used for controller implementation for different industries. SIL is a cost effective method for evaluating critical systems before its actual implementation in the real world scenario. SIL is vital stage in Model Based Design (MBD) for embedded system design.

7.2 Architecture of SIL simulation

The architecture of the communication between 4DIAC and OpenModelica happens by OPC UA protocol. As described in the previous chapters the OPC UA has been enabled in both 4DIAC and OpenModelica. Here 4DIAC works as a client and OpenModelica acts as a server. To get the list of IDs of different nodes and variables, please refer the Figure: 7.2 and Figure: 7.3.

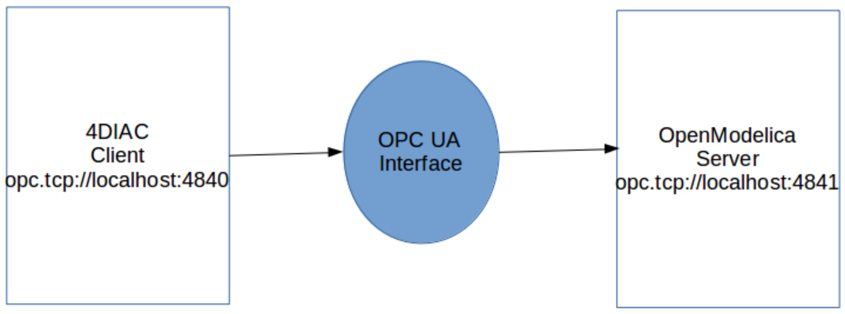
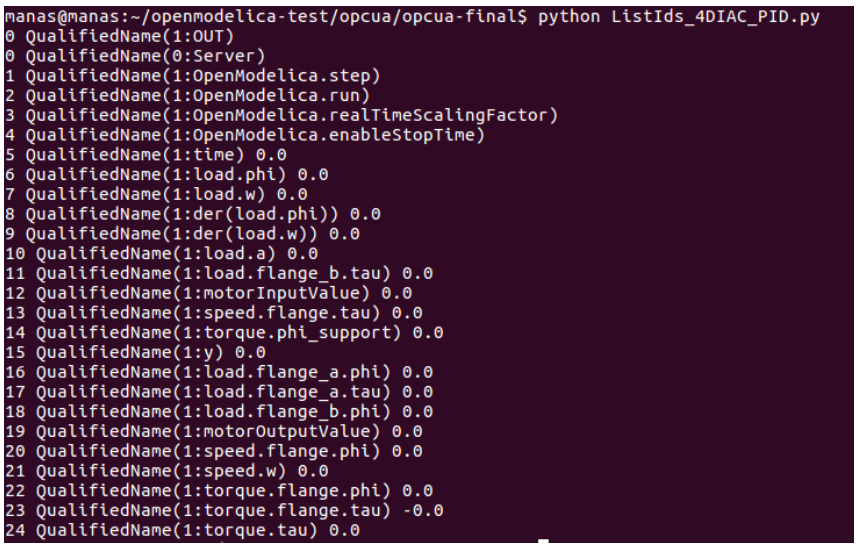
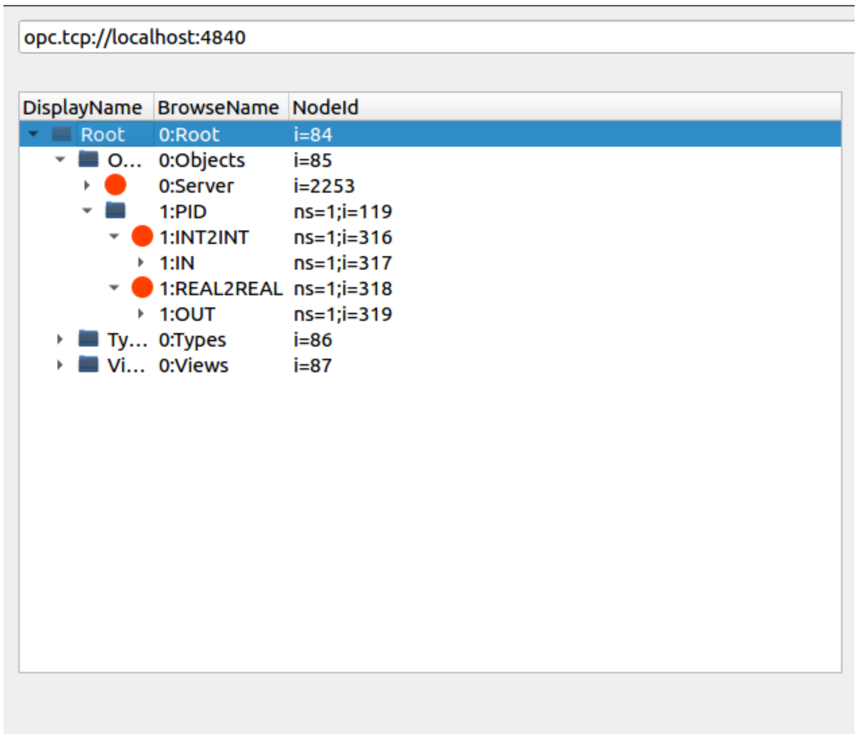


Figure 7.1: SIL Architecture



```
manas@manas:~/openmodelica-test/opcua/opcua-final$ python ListIds_4DIAC_PID.py
0 QualifiedName(1:OUT)
0 QualifiedName(0:Server)
1 QualifiedName(1:OpenModelica.step)
2 QualifiedName(1:OpenModelica.run)
3 QualifiedName(1:OpenModelica.realTimeScalingFactor)
4 QualifiedName(1:OpenModelica.enableStopTime)
5 QualifiedName(1:time) 0.0
6 QualifiedName(1:load.phi) 0.0
7 QualifiedName(1:load.w) 0.0
8 QualifiedName(1:der(load.phi)) 0.0
9 QualifiedName(1:der(load.w)) 0.0
10 QualifiedName(1:load.a) 0.0
11 QualifiedName(1:load.flange_b.tau) 0.0
12 QualifiedName(1:motorInputValue) 0.0
13 QualifiedName(1:speed.flange.tau) 0.0
14 QualifiedName(1:torque.phi_support) 0.0
15 QualifiedName(1:y) 0.0
16 QualifiedName(1:load.flange_a.phi) 0.0
17 QualifiedName(1:load.flange_a.tau) 0.0
18 QualifiedName(1:load.flange_b.phi) 0.0
19 QualifiedName(1:motorOutputValue) 0.0
20 QualifiedName(1:speed.flange.phi) 0.0
21 QualifiedName(1:speed.w) 0.0
22 QualifiedName(1:torque.flange.phi) 0.0
23 QualifiedName(1:torque.flange.tau) -0.0
24 QualifiedName(1:torque.tau) 0.0
```

Figure 7.2: List Ids of SIL DCMotor model



opc.tcp://localhost:4840

DisplayName	BrowseName	NodeId
Root	0:Root	i=84
O…	0:Objects	i=85
●	0:Server	i=2253
	1:PID	ns=1;i=119
●	1:INT2INT	ns=1;i=316
	1:IN	ns=1;i=317
●	1:REAL2REAL	ns=1;i=318
	1:OUT	ns=1;i=319
Ty…	0:Types	i=86
Vi…	0:Views	i=87

Figure 7.3: List Ids of PID on 4DIAC

7.2.1 PID controller in 4DIAC

4DIAC doesn't have a PID block. So, it's like creating a new library for PID controller in 4DIAC. This is a basic FB in 4DIAC. And the process of creating a basic FB in 4DIAC happens in no. of stages. The first stage is defining the PID block as in Figure: 7.2with input and output data types. This uses kp, kd and ki as real inputs to the PID block. Once the PID graphical block is ready, the next step is to define the ECC diagram which behaves as a state machine for the PID block as shown in Figure: 7.3. RESET algorithm as shown in Figure: 7.3 just initializes the internal variables. And the REQ algorithm has PID logic implemented as described in the chapter 11. For details of the structured text code , please refer Appendix D.

Once the PID block is ready, there is need to create an application to communicate with OpenModelica. The PID block with the SUBSCRIBER block which receives data from model output defined in OM and also the PUBLISHER block that made the data available to OM model as input is shown in the Figure: 7.4

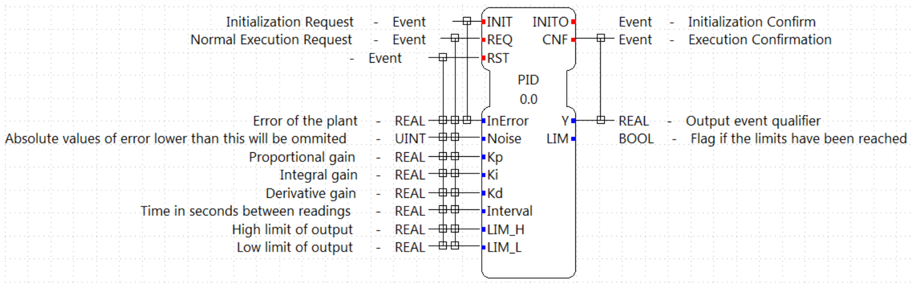

Initialization Request - Event
Normal Execution Request - Event
- Event
Error of the plant - REAL
Absolute values of error lower than this will be ommited - UINT
Proportional gain - REAL
Integral gain - REAL
Derivative gain - REAL
Time in seconds between readings - REAL
High limit of output - REAL
Low limit of output - REAL
INIT
REQ
RST
InError
Noise
Kp
Ki
Kd
Interval
LIM_H
LIM_L
PID
0.0
INITO
CNF
Y
LIM
Event - Initialization Confirm
Event - Execution Confirmation
REAL - Output event qualifier
BOOL - Flag if the limits have been reached

Figure 7.4: PID Block in 4DIAC

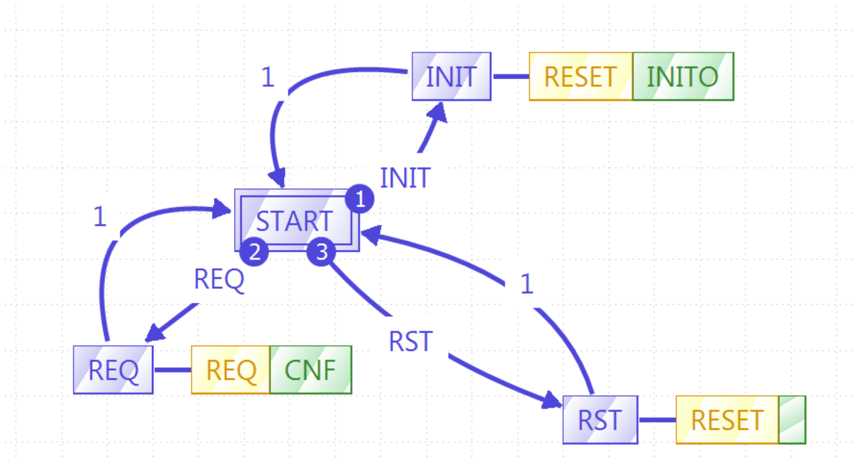


Figure 7.5: ECC of PID Block in 4DIAC

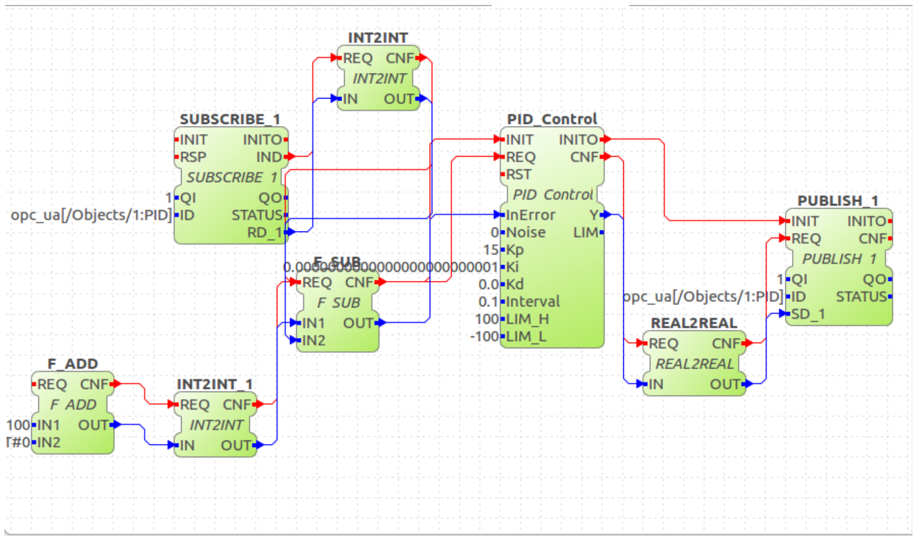


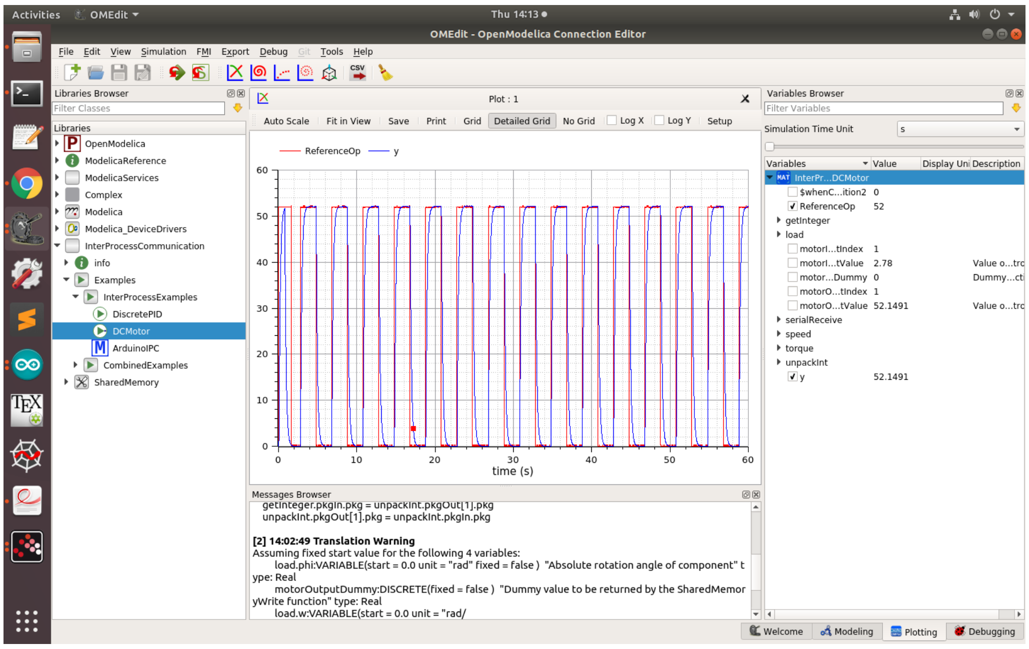
Figure 7.6: PID Controller in 4DIAC

7.2.2 Models in OpenModelica

There are different models defined on OM and tested with 4DIAC for SIL.

7.3 Testing & Evaluation

SIL testing on dc motor model as in Figure: 3.3 with PID controller on 4DIAC is as shown in Figure: 7.7



```
    getInteger.pkgIn.pkg = unpackInt.pkgOut[1].pkg
    unpackInt.pkgOut[1].pkg = unpackInt.pkgIn.pkg

[2] 14:02:49 Translation Warning
Assuming fixed start value for the following 4 variables:
        load.phi:VARIABLE(start = 0.0 unit = "rad" fixed = false )  "Absolute rotation angle of component" t
ype: Real
        motorOutputDummy:DISCRETE(fixed = false )  "Dummy value to be returned by the SharedMemor
yWrite function" type: Real
        load.w:VARIABLE(start = 0.0 unit = "rad/
```

Variables	Value	Display Uni	Description
InterPr...DCMotor			
☐ $whenC...ition2	0		
✓ ReferenceOp	52		
getInteger			
load			
☐ motorI...tIndex	1		
☐ motorI...tValue	2.78		Value o...tro
☐ motor...Dummy	0		Dummy...cti
☐ motorO...tIndex	1		
☐ motorO...tValue	52.1491		Value o...tro
serialReceive			
speed			
torque			
unpackInt			
✓ y	52.1491		

Figure 7.7: SIL test on 4DIAC & OM

Chapter 8

Hardware In Loop (HIL) simulation with Different Embedded hardware and OpenModelica

8.1 Motivation

OpenModelica has proven to be an excellent tool for engineers, with its algebraic modeling capabilities, myriad array of solvers, and strongly typed structure. It has an exceptionally detailed standard library, capable of modeling problems ranging from chemical, fluid dynamics, to power systems. At the same time, it has a dedicated community that keeps developing packages that can handle other complex problem domains such as aerospace engineering, etc.

Hardware-in-the-loop (HIL) simulation [15] [17], or HWIL, is a technique that is used in the development and testing of complex real-time embedded systems. HIL simulation provides an effective platform by adding the complexity of the plant under control to the test platform. The complexity of the plant under control is included in test and development by adding a mathematical representation of all related dynamic systems. These mathematical representations are referred to as the "plant simulation" [11]. The embedded system to be tested interacts with this plant simulation.A HIL simulation must include emulation of sensors and actuators. These emulations act as the interface between the plant simulation and the embedded system under test. The value of each electrically emulated sensor is controlled by the plant simulation and is read by the embedded system under test (feedback). Likewise, the embedded system under test implements its control algorithms by outputting actuator control signals. Changes in the control signals result in changes to variable values in the plant simulation. For embedded systems, however, OpenModelica has only one major package, the Modelica Device Drivers. It is well-made but requires a steep learning curve, and expects a certain degree of expertise. For HIL simulations, it isn't a convenient tool to use. To fill in this gap in the existing software, we developed our own tool to interface OpenModelica with embedded targets.

8.2 Architecture of HIL simulation

The HIL package we developed for OpenModelica consists of three primary parts:

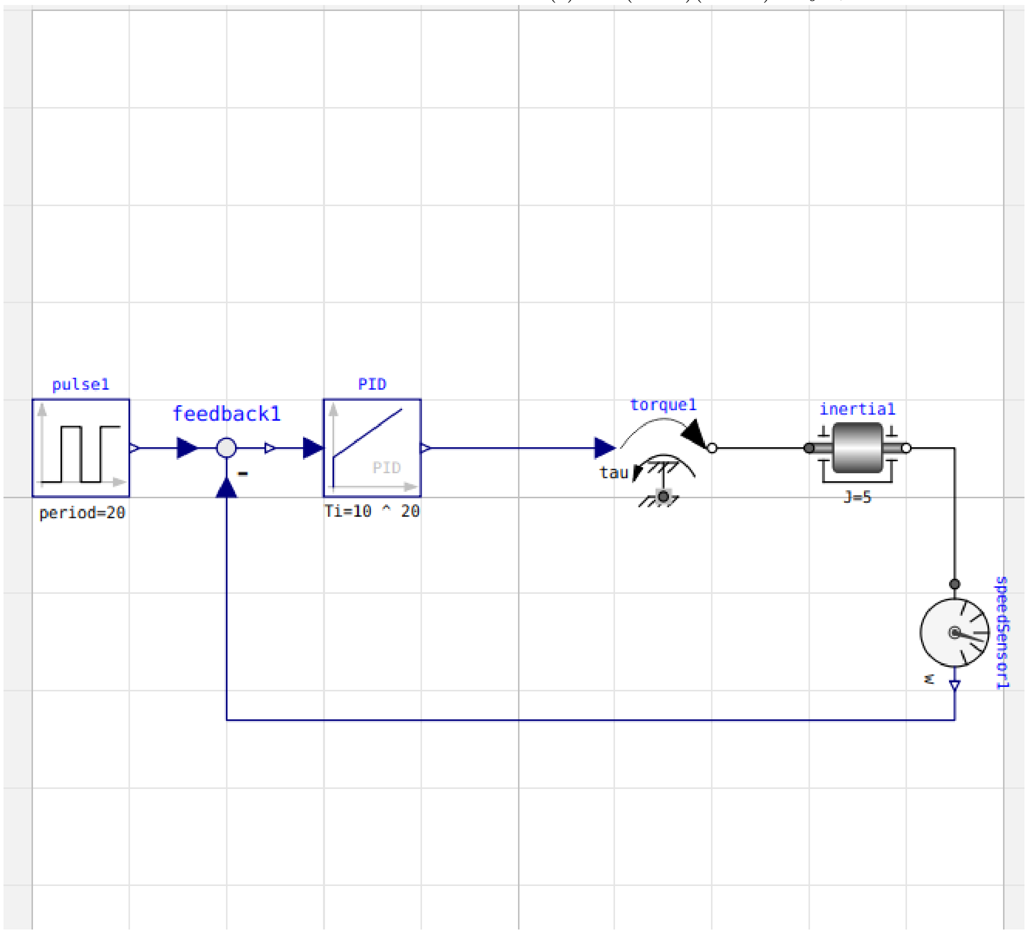
- Models on OpenModelica
- The Interface
- The embedded target interface

8.2.1 Models on OpenModelica

We developed models in OpenModelica to provide examples of HIL systems that can work using our routine. They have been described below in increasing order of complexity. All of them have been displayed with a virtual PID routine for ease of visualization of the feedback loop.

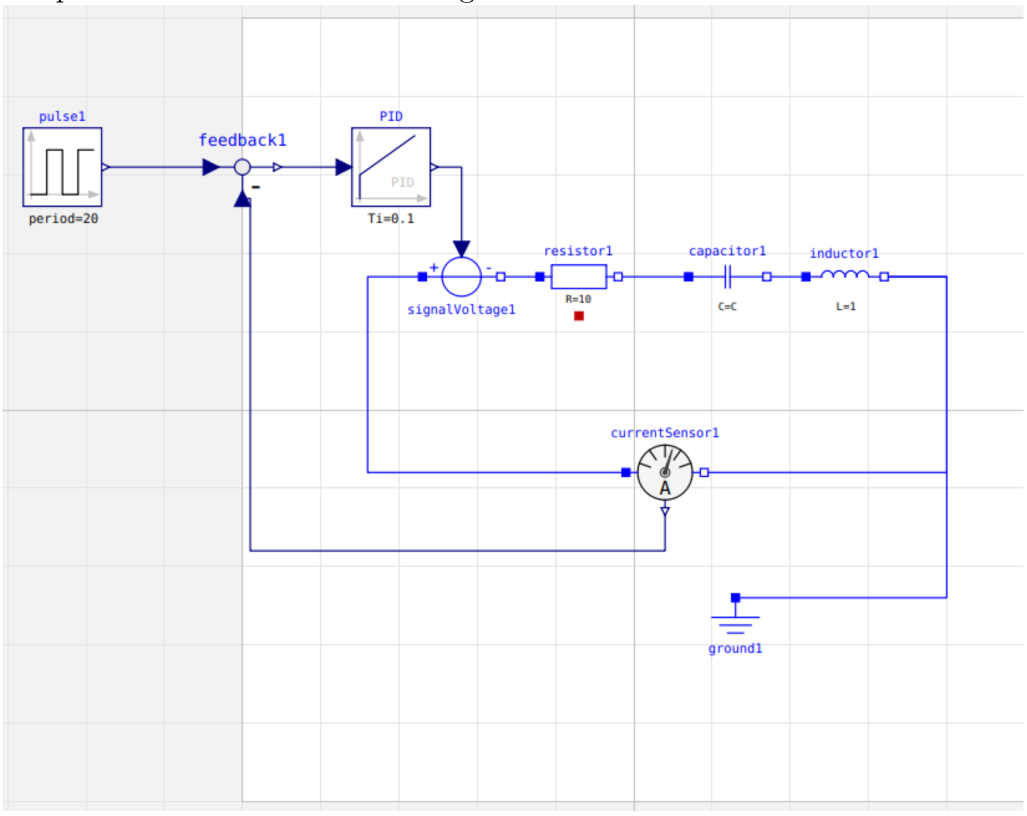
First order DC Motor

This model is a straightforward first-order DC motor. The angular velocity is the process value and the torque is being used as the control variable to do so. The controller is a PID.
The equation of the model is: $P(s) = \frac{\Theta(s)}{V(s)} = \frac{K_t}{(Js+b)(Ls+R)+K_bK_t}$



RLC Circuit

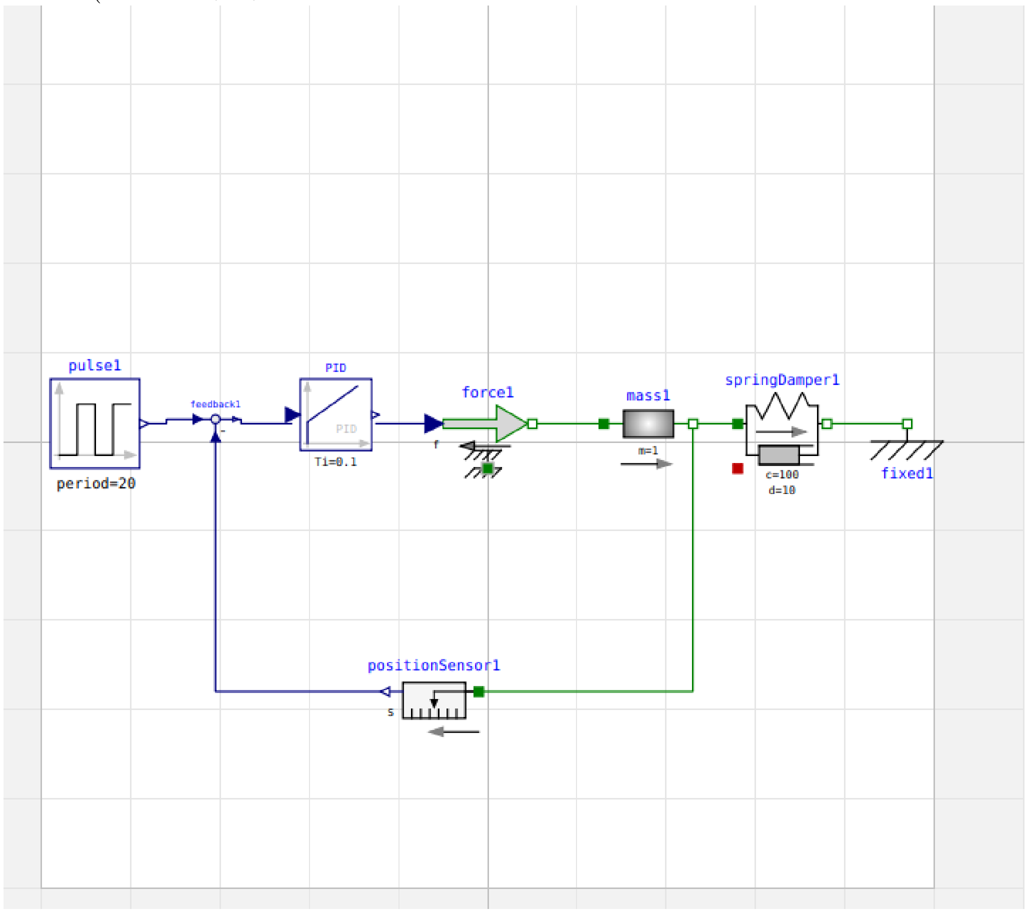
This model is a 2nd order electric circuit, containing a resistor, inductor, and capacitor connected in series, and connected to a signal voltage source. The current is the process variable and the voltage is the control variable. The controller is a PID.



Spring-Mass System

This is a 2nd order mechanical model. The system involves a mass attached to a spring-damper system. The position of the mass is the process variable and the force applied to it is the control variable. The controller is a PID.

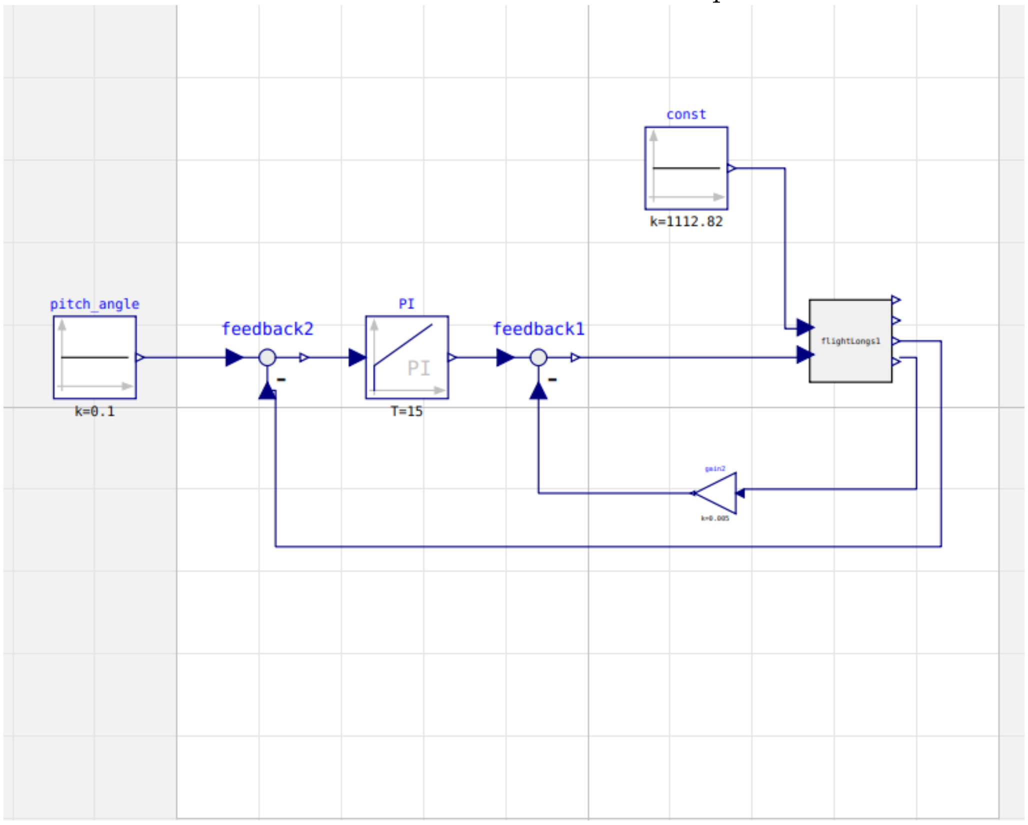
The transfer function for this model is:
$H(s) = \frac{X(s)}{F(s} = \frac{1}{ms^2+cs+k}$



Flight Pitch Controller

This is a complex model involving a longitudinal flight model. The target is to maintain a constant pitch for the aircraft. This can be done by altering both thrust and elevation. For the purposes of this example, we will keep a constant thrust, and control the pitch using only the elevation angle. The control model is a CAS as detailed in

The transfer function of this model is out of the scope of the current discussion.



8.2.2 The Interface

We have used the three communication protocols described in previous chapters to exchange data between OpenModelica and the embedded targets. Below I will explain the process using which the data was passed.

In describing the process, I will assume that the data is processed in OpenModelica, then being passed first from OpenModelica to the embedded target, being processed on the embedded target, and then the embedded target output is being passed to OpenModelica.

- A float or an integer is processed by an OpenModelica block.
- It is specified in the shmWrite function along with a position number. This passes it through the shared memory buffer.
- The communication protocol handles the data in the form of chars.
- On the embedded target, the chars are read, tokenized, and parsed based on their token position. The data is then converted to the relevant floats or integers.
- The control algorithm on the embedded target then uses these quantities as input and provides the relevant output.
- The output is converted to chars, given a position number, and then written onto the serial port.
- It is then read by OpenModelica via the shmRead function and the relevant position number.

8.2.3 PID controller on Embedded hardware

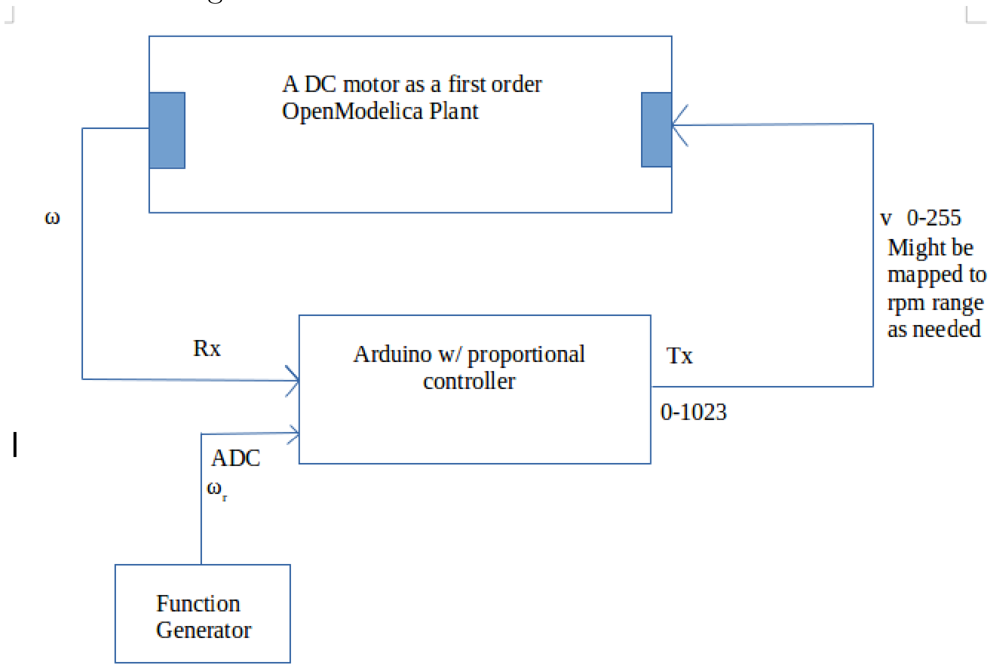
We started by implementing a PID routine on an embedded target. The equations for this are detailed below:
Output = $\mathrm{K}_p e(t) + K_I \int e(t)dt + K_D \frac{d}{dt} e(t)dt$
where e = Setpoint-Input

The code for the PID implementation for the embedded target, Arduino & TIVA C series has been included in Appendix E

8.3 Testing & Evaluation

In this section, we set up and test out the HIL simulation. The embedded target we are using is an arduino, TIVA C series ARM cortex-M4 controller and Raspbery Pi. The model we are using is the DC Motor described in Section 11.2.1.

The circuit diagram is as shown below:

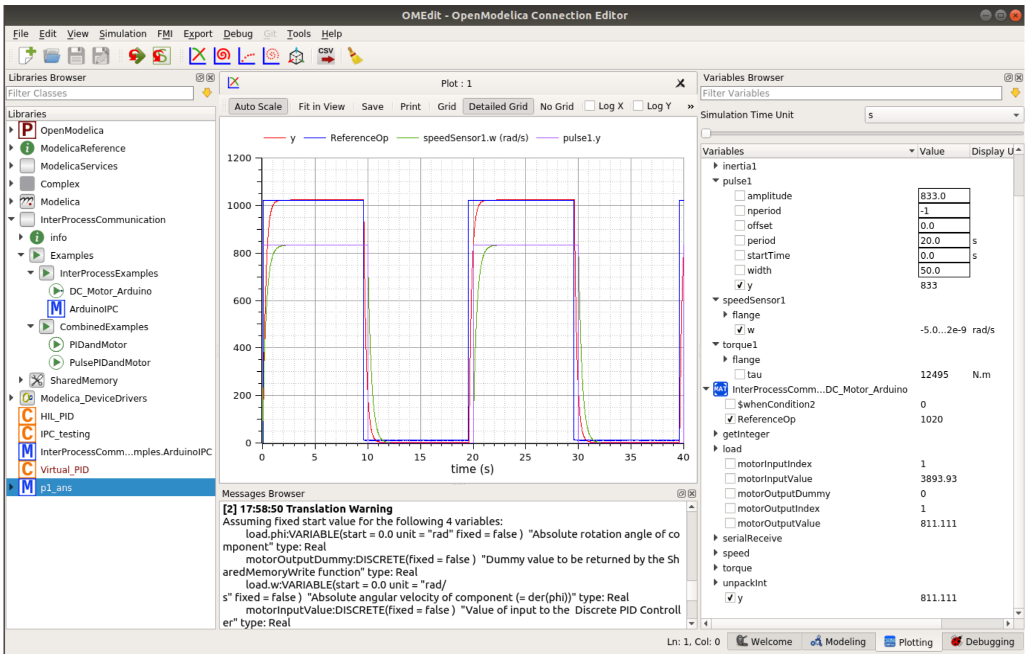

A DC motor as a first order OpenModelica Plant
ω
v 0-255
Might be mapped to rpm range as needed
Rx
Arduino w/ proportional controller
Tx
0-1023
ADC
ω_r
Function Generator

8.3.1 DC motor

We start by testing the DC Motor model.
The following are the quantities used in this test:

- The setpoint here is obtained via a function generator, and is a pulse with an amplitude of 1020 rad/s and period of 20 seconds. The software setpoint is set at 833 rad/s with a similar period.
- The controller is a discrete proportional controller [19] with $k_p = 15$.
- The controller is discrete because it is on an embedded target.
- The plant is a DC motor with $J = 5m^2$.
- The process variable as previously mentioned, is the angular velocity.
- The control variable is the torque of the motor.
- The experiment is run for 40 seconds.

The output of the experiment is shown below:



Messages Browser

```
[2] 17:58:50 Translation Warning
Assuming fixed start value for the following 4 variables:
      load.phi:VARIABLE(start = 0.0 unit = "rad" fixed = false )  "Absolute rotation angle of co
mponent" type: Real
      motorOutputDummy:DISCRETE(fixed = false )  "Dummy value to be returned by the Sh
aredMemoryWrite function" type: Real
      load.w:VARIABLE(start = 0.0 unit = "rad/
s" fixed = false )  "Absolute angular velocity of component (= der(phi))" type: Real
      motorInputValue:DISCRETE(fixed = false )  "Value of input to the  Discrete PID Controll
er" type: Real
```

Ln: 1, Col: 0

Simulation Time Unit: s

Variables	Value	Display U
inertia1		
pulse1		
amplitude	833.0	
nperiod	-1	
offset	0.0	
period	20.0	s
startTime	0.0	s
width	50.0	
✓ y	833	
speedSensor1		
flange		
✓ w	-5.0...2e-9	rad/s
torque1		
flange		
tau	12495	N.m
InterProcessComm...DC_Motor_Arduino		
$whenCondition2	0	
✓ ReferenceOp	1020	
getInteger		
load		
motorInputIndex	1	
motorInputValue	3893.93	
motorOutputDummy	0	
motorOutputIndex	1	
motorOutputValue	811.111	
serialReceive		
speed		
torque		
unpackInt		
✓ y	811.111	

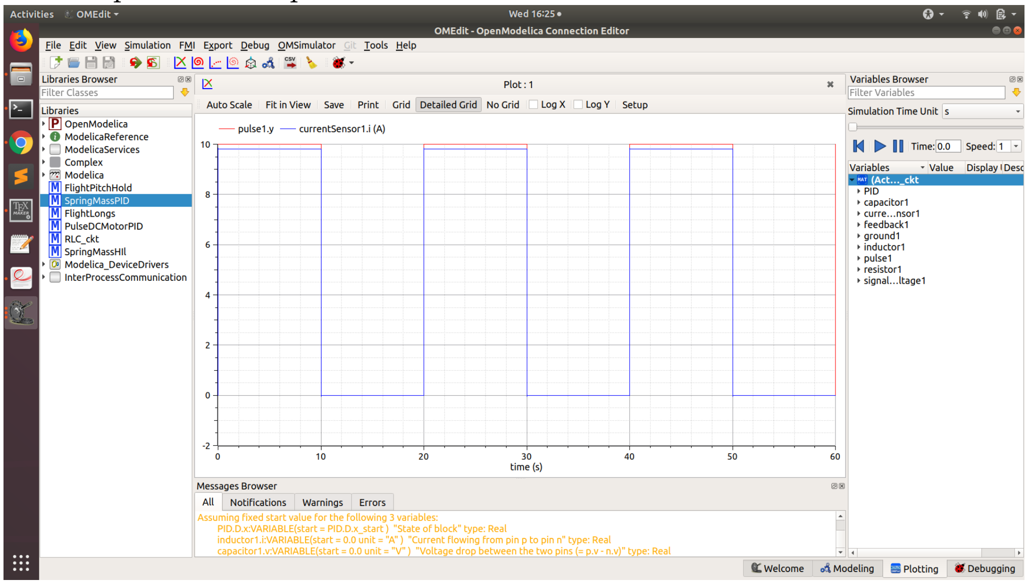
As can be seen, the output signal generated by the proportional controller on the arduino tracks the setpoint as comparably well as the software controller.

8.3.2 RLC Circuit

Now we look at the RLC circuit model. This is a second order electrical model as mentioned previously. The following are the quantities used in this test:

- The setpoint here is obtained via a function generator, and is a pulse with an amplitude of 100 A and period of 20 seconds. The software setpoint is set at 100 A with a similar period.
- The controller is a discrete PID controller with:
  $k_p = 30$
  $k_i = 500$
  $k_d = 2$.
- The controller is discrete because it is on an embedded target.
- The plant is a series RLC circuit with:
  $R = 10$
  $L = 30mH$
  $C = 100\mu F$.
- The process variable as previously mentioned, is the current.
- The control variable is the voltage of the system.
- The experiment is run for 60 seconds.

The output of the experiment is shown below:



```
Assuming fixed start value for the following 3 variables:
      PID.D.x:VARIABLE(start = PID.D.x_start )  "State of block" type: Real
      inductor1.i:VARIABLE(start = 0.0 unit = "A" )  "Current flowing from pin p to pin n" type: Real
      capacitor1.v:VARIABLE(start = 0.0 unit = "V" )  "Voltage drop between the two pins (= p.v - n.v)" type: Real
```

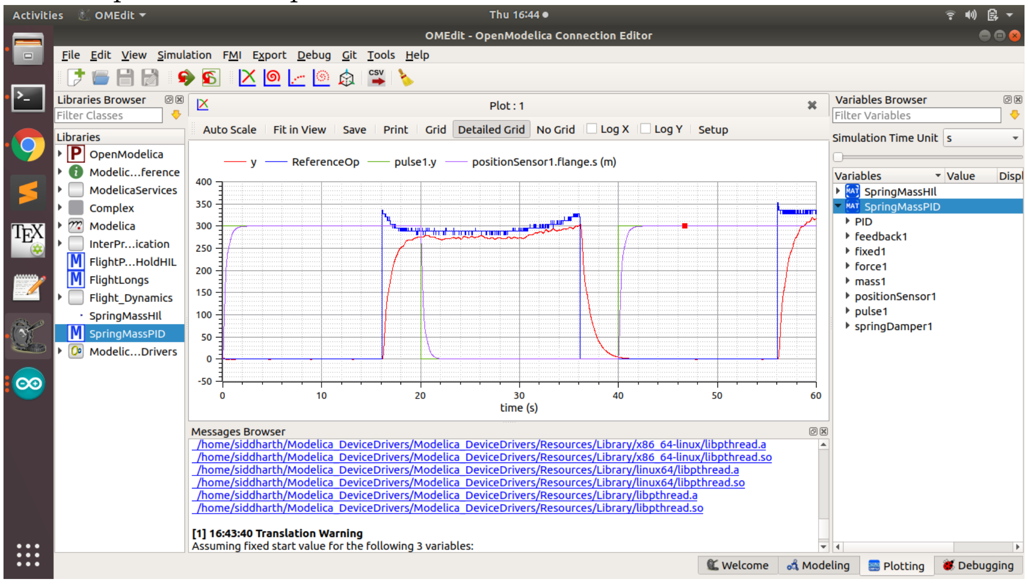
As can be seen, the output signal generated by the PID controller on the arduino tracks the setpoint as comparably well as the software controller.

8.3.3 Spring-Mass Model

Now we look at a mechanical model. This is a second order spring-mass model as mentioned previously. The following are the quantities used in this test:

- The setpoint here is obtained via a function generator, and is a pulse with an amplitude of 300 m and period of 20 seconds. The software setpoint is set at 300 m with a similar period.
- The controller is a discrete PID controller with:
  $k_p = 30$
  $k_i = 10$
  $k_d = 0.1$.
- The controller is discrete because it is on an embedded target.
- The plant is a spring-mass model with:
  $R = 1kg$
  $d = 10N.s/m$
  $c = 100N/m$
  $= 1m$.
- The process variable as previously mentioned, is the displacement of the mass.
- The control variable is the force applied to the system.
- The experiment is run for 60 seconds.

The output of the experiment is shown below:



```
/home/siddharth/Modelica_DeviceDrivers/Modelica_DeviceDrivers/Resources/Library/x86_64-linux/libpthread.a
/home/siddharth/Modelica_DeviceDrivers/Modelica_DeviceDrivers/Resources/Library/x86_64-linux/libpthread.so
/home/siddharth/Modelica_DeviceDrivers/Modelica_DeviceDrivers/Resources/Library/linux64/libpthread.a
/home/siddharth/Modelica_DeviceDrivers/Modelica_DeviceDrivers/Resources/Library/linux64/libpthread.so
/home/siddharth/Modelica_DeviceDrivers/Modelica_DeviceDrivers/Resources/Library/libpthread.a
/home/siddharth/Modelica_DeviceDrivers/Modelica_DeviceDrivers/Resources/Library/libpthread.so

[1] 16:43:40 Translation Warning
Assuming fixed start value for the following 3 variables:
```

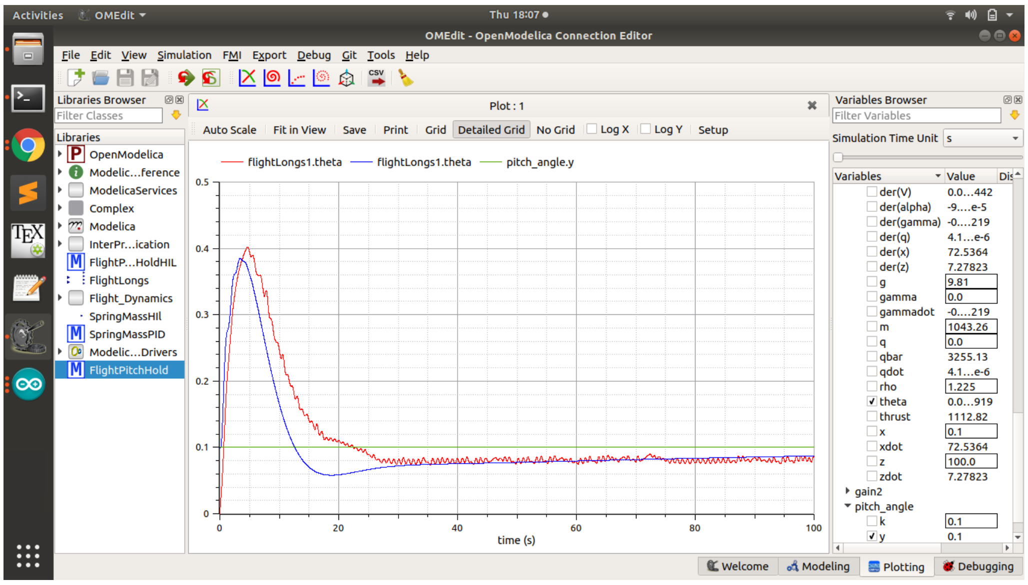
As can be seen, the output signal generated by the PID controller on the arduino tracks the set-point as comparably well as the software controller.

8.3.4 Flight Pitch Control Model

Lastly, we will perform an evaluation of the flight pitch model. This is a second order spring-mass model as mentioned previously. The following are the quantities used in this test:

- The setpoint here is set at 0.1 rad. Seeing as the signal is uncomplicated, a function generator is not used.
- The controller is a discrete CAS controller. The inner loop has a proportional controller with $k_p = 0.005$. The outer loop has a PI controller with $k_p = 0.2$ and $k_i = 0.06$.
- The controller is discrete because it is on an embedded target.
- The plant is a longitudinal cessna model with the standard characteristics[cessnau].
- The process variable as previously mentioned, is the pitch of the flight.
- The control variables are the thrust (constant for this experiment) and the elevation angle.
- The experiment is run for 100 seconds.

The output of the experiment is shown below:



Simulation Time Unit s

Variables	Value
der(V)	0.0...442
der(alpha)	-9....e-5
der(gamma)	-0....219
der(q)	4.1...e-6
der(x)	72.5364
der(z)	7.27823
g	9.81
gamma	0.0
gammadot	-0....219
m	1043.26
q	0.0
qbar	3255.13
qdot	4.1...e-6
rho	1.225
✓ theta	0.0...919
thrust	1112.82
x	0.1
xdot	72.5364
z	100.0
zdot	7.27823
gain2	
pitch_angle	
k	0.1
✓ y	0.1

As can be seen, the output signal generated by the CAS controller on the arduino tracks the setpoint as comparably well as the software controller.

Chapter 9

SIL & HIL Comparison

9.1 Motivation

In this chapter, we will explore in detail the comparisons between using a hardware controller in the loop, and a software controller. SIL usually gives us much cleaner control loops but we expect HIL to capture the situations a controller will placed under in the real world. This makes it extremely important that we compare the two and are aware of the reasons the two might differ.
We will compare the errors between the controlled variable output and the setpoints for both the HIL and SIL simulations. We will use the models we described in the previous chapter to do this.
Lastly, we will discuss the applications that these systems will find themselves in.

9.2 Error Comparison

Here we will compare the error generated by a SIL controller and the hardware controller. We will do so for all the models discussed in chapter 11. The following assumptions are being made for all models for this particular error comparison.

- The controller is on an arduino, ARM Cortex-M4, 4DIAC and another Raspberry Pi.
- The controller for the first and second order models is a PID routine as described in Chapter 11.
- The setpoint is being generated by a function generator.

9.3 Timing Comparison

SIL is tested on a Linux 64 bit PC having i5 processor. HIL model remains on the above configuration PC with controller on external hardware.

Table 9.1: Error Comparison SIL vs HIL

Model	Error
DC Motor	0.000012
RLC	0.000010
Spring Mass	0.000010
Flight Pitch control	0.0000023

Table 9.2: Timing Comparison SIL vs HIL

Method	Max Time(ms)	Avg Time(ms)
SIL	12	10
HIL	30	25

9.4 Applications

As industry is moving towards Industry 4.0, HIL and SIL bears a significant importance. In cyber physical production systems, the HIL can give real-time parameters for actual plant setup.

Chapter 10

Conclusion

The ideaa behind "Developing a Generic Purpose OpenModelica Package for Embedded Applications", is to develop a library for OpenModelica to support different families of micro controllers, Single Board Computers like Raspberry Pi, and also 4DIAC software. Hardware support should comply with HIL testing support and software support should comply with SIL support. With the work completed till now, the package supports ATmega328p (Arduino),TIVA-C series (ARM Cortex-M4) boards, Raspberry Pi and 4DIAC. Initially, UART implementation was tested on ATmega series of controllers. But due to time synchronization issues, MDD was explored as it has the functionality to generate Embedded C code from OpenModelica models. But HIL implementation wasn't possible with MDD. Then to support HIL simulations, the IPC approach was adapted which is found to be supported on most AVR controllers. The drawbacks of IPC, specifically that of system dependency, was handled by using the Firmata protocol. A number of experiments were performed on the Tiva-c board. Hardware-in-loop simulations were implemented using the PID controller. But for SBCs the approach has to be totally different as it is difficult to develope a firmware or facilitate device drivers for SBCs. So, a more generic and platform independent protocol OPC UA was adopted. It also solved the issue of interfacing with 4DIAC for SIL simulation. OPC UA is found to be a generic approach which can be used any of the embedded hardware targets with OpenModelica.

Chapter 11

Future Work

With this development, OpenModelica now has support for most of the micro-controllers, Single Board Computers and 4DIAC. Future work involves :-

1. Optimize the implementation for time critical systems
2. HIL & SIL testing for actual industrial set-up
3. HIL & SIL testing for different domains
4. Coming up with a common GUI for different embedded devices

Appendix A

OPC UA on 4DIAC

A.1 Motivation

The motivation behind this integration is to transform industrial control systems from the standard pyramid topology into the reconfigurable systems. 4DIAC framework is based upon to the IEC 61499 standards. It allows an user to create any type of control application and then implements the feature of reconfiguring the application by creating and modifying functional blocks (FBs) and making connections in-between them. But, the implemented application runs only on one layer of the industrial hierarchical pyramid. The issue with this implementation of control system is that the systems on other layers do not react to this reconfiguration, because they are not capable of automatic detecting this kind of reconfiguration. But the issue was solved by OPC UA implementation as the data sharing protocol. The information model of OPC UA which allows not only to store data, but also to store them in the structured series of interconnected nodes. The developed solution uses OPC UA not only to share values, but also structure of FBs with other systems in network. This brings new possibilities to the systems on other layers of the pyramid with detecting reconfiguration in 4DIAC application.

A.2 IEC 61499

This section will give a brief idea into the IEC 61499 standard. 4DIAC framework is built on IEC 61499 standard. IEC 61499 is a new family of standards for Industrial Process and Control Systems. This standard mainly consists of 4 parts. Such as

1. IEC 61499-I : Function Blocks - Part-1: Architecture
2. IEC 61499-II : Function Blocks - Part-2: Software tools requirements
3. IEC 61499-III : Function Blocks - Part-3: for Industrial Process Control Systems
4. IEC 61499-IV : Function Blocks - Part-4: Rules for creating compatibility profiles

The main purpose of this standard is defining Function Bocks (FBs) and creating a network by connecting the functional blocks. IEC 61499 is based on an older standard known as IEC 61131 family of standards which is the most widely adopted standard in industrial process and control systems domain.As IEC 61499 is built upon IEC 61131 and hence it's easier to adopt this standard. The features which makes it easier for user acceptance because of its distributive nature, modularity, reconfigurability and event-triggered model. In IEC 61499 standards, models can be defined through graphical block diagrams to create a distributive control application.

Models that can be defined in this standard are the application model, the system model, the device model, the resource model and the FB model. This models are in a hierarchical manner such as application model has multiple system models, system model in return consists of multiple device models etc. The most basic and important model is the Functional Block (FB) model. FB as Figure: A.1 in is an independent self-sufficient entity which provides specified functionalities.

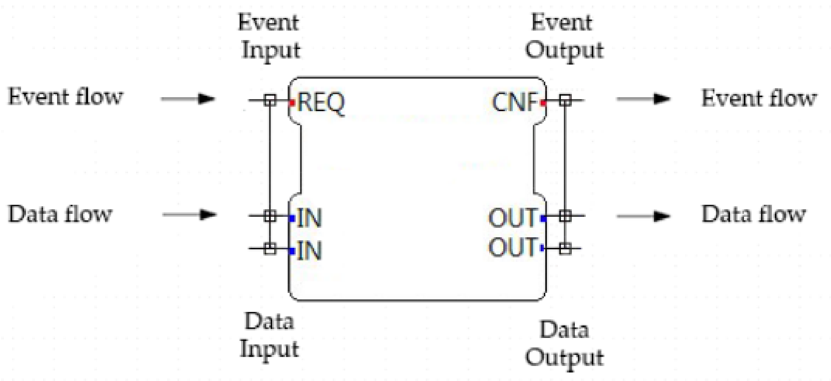


Figure A.1: IEC 61499 FB

A.3 IEC 61499 Base Model

Implementing model in IEC 61499 system can be done in two phases. In the first phase, user creates network of Function Block (FB) by interconnecting the FBs with data and event connection. At this point, the developer has only functionality part in mind and it is independent of any device or control infrastructure. After the implementation of functionalities in the first phase, the system model already created are mapped to control devices. The IEC 61499 model is executed on devices. Each device comprises of managing device component, communication interface that facilitates communication between the devices, to access the sensors, process interface provides the services, also present are different actuators and other physical devices for controlling the processes. Resources can also be part of devices. These are functional entities which may have the whole applications or the parts of applications. But resources are device independent. So, resources in any particular device can be modified, added and/or removed without any modification in any of other resources. This is a vital step to get to the goal of reconfiguration. The goal of the resource is to provide an execution environment, to deliver event notifications.

A.4 IEC 61499 applications

Very basic principles of the 4DIAC framework is creating different types of applications by using function blocks and then deploying the applications to use. The essential idea of having 4DIAC framework is compiling own version of 4DIAC runtime environment dedicated for the current applications of IEC 61499 which can be differentiated into the research and as well as industrial sectors. The IEC 61499 standard came to a proper existence only in January 2005. But before the standardization came to existence, around 2000 it existed in a format known as Public Available Specification. Though IEC 61499 was available in different forms for quite a long time, but most work published up to now has been mostly in academics or only prototypes for industrial test cases. The Industrial sector adopted IEC 61499 primarily for case studies and prototypes. A number of test cases has been implemented through the Function Block Development Kit (FBDK) / Function Block Run-Time (FBRT) package. Java and Java Classes are used to implement FBRT and IEC 61499 elements. This package is considered as a reference implementation and was used to test models and standards. In FBRT the event notification is handled by function call. The source FB calls notification function of the event connection object and this object triggers event on destination FB by calling his event function. This approach creates delays and is also one of the greatest reasons why FBRT has never been adopted by industry sector. Another reason is also that this Java implementation was not able to run on small industrial control platforms.

A.5 The 4DIAC initiative

4DIAC open source initiative was founded by collaboration of The Automation and Control group of Vienna University of Technology and PROFAC-TOR GmbH. The aim of 4DIAC initiative is to create an open-source framework based on IEC 61499 standard which will provide reference implementation of execution model for IEC 61499. 4DIAC initiative is currently focusing and developing two major projects of IEC 61499 compliant

1. DIAC IDE - Engineering tool
2. FORTE - Runtime environment

To work with 4DIAC framework you have to use both of this parts. You can find instructions how to install and run this project on your own computer in Appendix A. Brief information about 4DIAC IDE and FORTE are given in the next section..

A.6 4DIAC IDE & Runtime Environment

A.6.1 4DIAC IDE

The development environment of IEC 61499 is implemented by 4DIAC IDE and the 4DIAC IDE is built on top of Eclipse open source framework. The Eclipse framework makes 4DIAC IDE an open source IDE and platform independent. All the other tools built on Eclipse works well in 4DIAC IDE. Each user can create their own user specific application. There are three default environments which gets created in 4DIAC IDE. These are required to create a basic application in 4DIAC. FBs can be modified/added/deleted, events can be created and data connection should happen. The system configuration of one of the examples supplied with 4DIAC IDE can be taken as a reference. The system configuration as in Figure: A.2 consists of one device connected via Ethernet. Each of this device includes two resources. One of the resources is management resource always named MGR_ID and is read-only. The FB Network running on the resource can be edited by double clicking on it. Type Management facility is dedicated to edit and create developers' own FBs. The application window having the required block-sets for the current application is somewhat looks like as in Figure: A.3. In case of basic FB you can edit function of this FB by editing its ECC or Algorithm written in pseudo-code as in Figure: A.4. The function of the Composite FB can by modified or created by editing Composite Network. Only Service Interface FBs (SIFBs) function is not allowed to change in 4DIAC editor. Function of SIFBs can be modified only by editing FORTE source. All changes made in Type library have to be exported into the FORTE code. To use this modified FBs in control system it is necessary to recompile the FORTE with these updated function block. Deployment Perspective is dedicated to the deployment and upload application into the control system devices by clicking on Download button. There is also possibility to run local FORTE and FBRT directly from Deployment Perspective. In case of local FORTE runtime, all its output are shown in Console window.

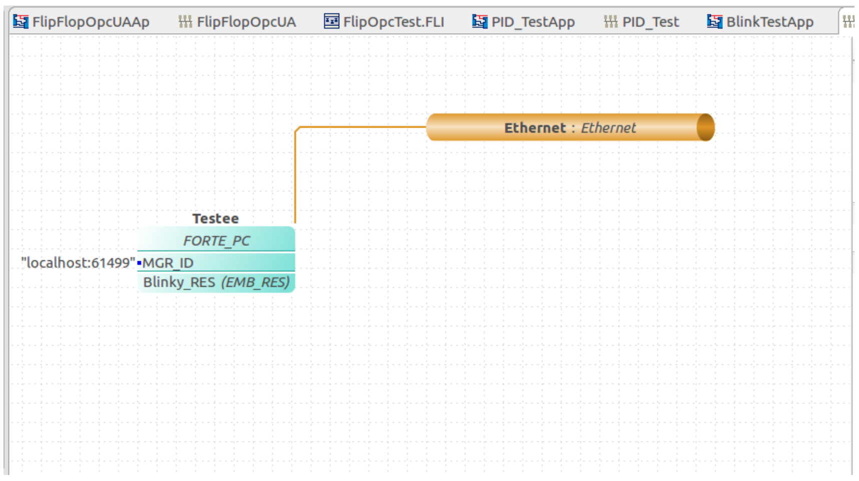


Figure A.2: 4DIAC system configuration

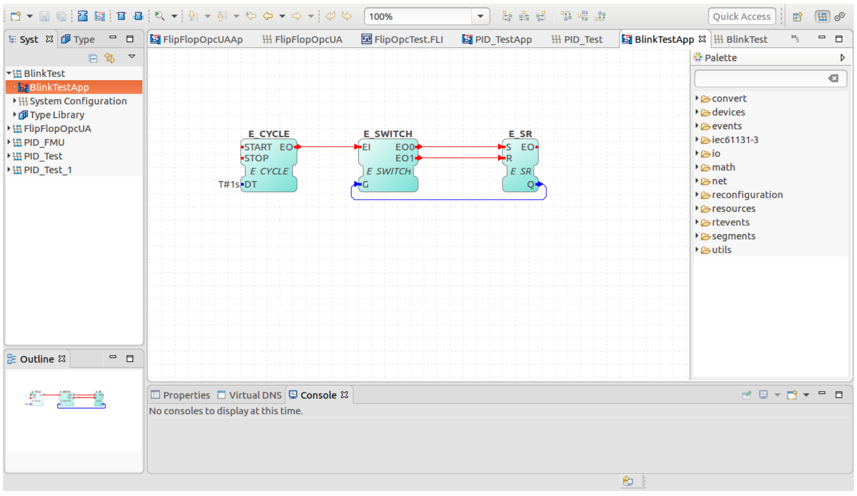


Figure A.3: Application Window

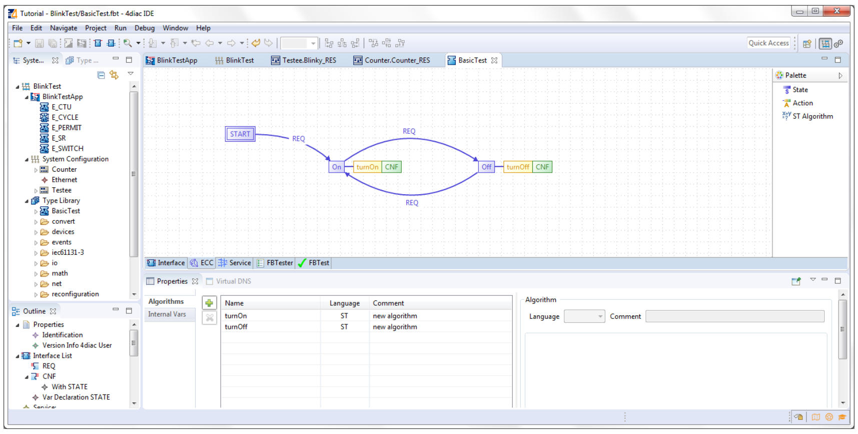


Figure A.4: ECC for 4DIAC blink application

A.6.2 4DIAC RUNTIME ENVIRONMENT - FORTE

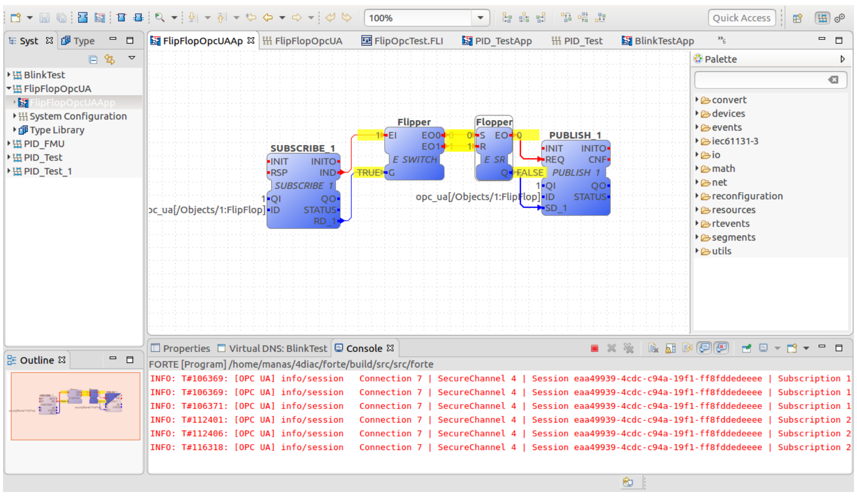
The FORTE is a portable C++ implementation of an IEC 61499 runtime environment. It is focused on small embedded control devices like 16/32bit controllers and provides execution of all IEC 61499 types of functions blocks. Currently FORTE is available for Windows, POSIX (Cygwin, Linux), NET+OS 7, and eCos. It can also be used on small embedded boards like RaspberryPi, BeagleBone, etc.

A.7 open62541 stack

Different OPC UA standards are published by the OPC UA Foundation. No official communication stack has been announced yet. The OPC Foundation has just published some example codes in Ansi C and JAVA, but there is no complete SDK or even documentation present. However there are a few open source or proprietary stacks available. On the OPCConnect website, OPC UA stacks overview, one can find brief description of available SDKs and toolkits. There are also some open stacks available for OPC UA. But these stacks are often published under license which is not compatible with the 4DIAC license. OpenOpcUa is open source, but to use it, one needs to pay a one time fee. Also, there is FreeOpcUa hosted by GitHub(https://github.com/FreeOpcUa/freeopcua), but this SDK is not fully working and the lack of documentation makes it impossible to use it for the purpose of this thesis. However FreeOpcUa is a C/C++ and Python SDK, and in the Python version, much more progress has been made. This SDK provides a great open-source Python GUI interface for discovering the OPC UA server. Considering two important parameters: license and documentation, the open62541 stack seems to be optimal. This stack is used to integrate OPC UA into FORTE. Open 62541 is a communication stack based on OPC UA standards published as IEC 62541 licensed under LGPL and is freely available on GitHub. This stack is fully scalable, supports multi-threaded architecture, where each connection or session is operated by a separate thread. Open 62541 is written in C99 with POSIX support, so it is able to run on Windows, Linux, MacOS and Android. POSIX Linux support means open62541 stack can also run on small embedded machines like RaspberryPi, PLCs, etc.

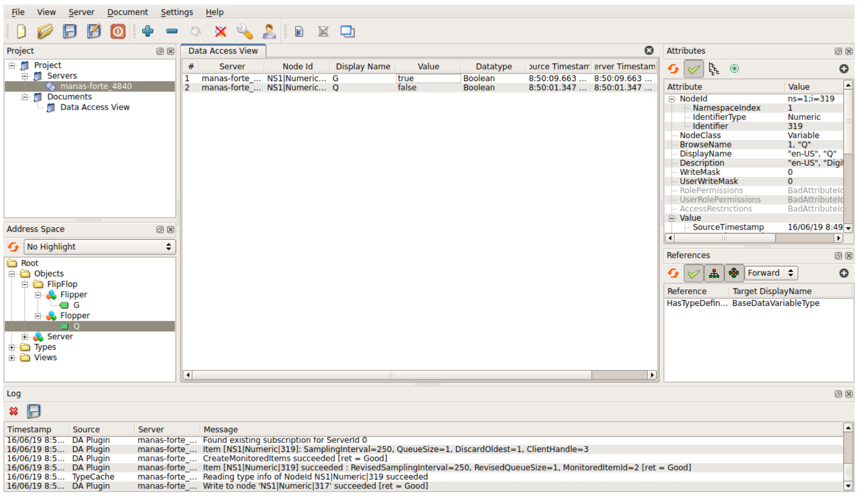
A.8 Integration of open62541 stack for OPC UA in 4DIAC

4DIAC by default doesn't support OPC UA. Users have to build open62541 with FORTE so that OPC UA functionality is made available in 4DIAC. The building instructions are in Appendix A. During the first stage, module open62541 enables 4DIAC to create a dummy server and client stack. 4DIAC tool already has a different method of data transfer in network by using PUBLISH and SUBSCRIBE FBs, but this connection is just peer-to-peer. This means only transfer between two resources, devices, or applications is possible. While built-in solution in 4DIAC allows only connection between two points, both running 4DIAC runtime, OPC_UA_WRITE and OPC_UA_READ can write and read values from any distant or local server, which can create connections among multiple devices, not necessarily using 4DIAC. The SUBSCRIBE block subscribes or receives one or more variable values available from the other servers into the address space defined. The PUBLISHER block sends or makes the computation value available to the server on the other side. But the address space of the variables should be same for the subscriber and publisher. With the INIT event, the OPC UA server starts with a default IP address of opc.tcp://localhost:4840. Here only one OPC UA server is created and the address space is shared between all the FBs. Please refer to the Figure: A.5 for flip-flop application and testing with UaExpert in Figure: A.6. In this application the binary value available in the subscriber end gets toggled and the toggled value is available for publish.



```
FORTE [Program] /home/manas/4diac/forte/build/src/src/forte
INFO: T#106369: [OPC UA] info/session   Connection 7 | SecureChannel 4 | Session eaa49939-4cdc-c94a-19f1-ff8fddedeeee | Subscription 1
INFO: T#106369: [OPC UA] info/session   Connection 7 | SecureChannel 4 | Session eaa49939-4cdc-c94a-19f1-ff8fddedeeee | Subscription 1
INFO: T#106371: [OPC UA] info/session   Connection 7 | SecureChannel 4 | Session eaa49939-4cdc-c94a-19f1-ff8fddedeeee | Subscription 1
INFO: T#112401: [OPC UA] info/session   Connection 7 | SecureChannel 4 | Session eaa49939-4cdc-c94a-19f1-ff8fddedeeee | Subscription 2
INFO: T#112406: [OPC UA] info/session   Connection 7 | SecureChannel 4 | Session eaa49939-4cdc-c94a-19f1-ff8fddedeeee | Subscription 2
INFO: T#116318: [OPC UA] info/session   Connection 7 | SecureChannel 4 | Session eaa49939-4cdc-c94a-19f1-ff8fddedeeee | Subscription 2
```

Figure A.5: 4DIAC flip flop application



#	Server	Node Id	Display Name	Value	Datatype	ource Timestam	erver Timestam
1	manas-forte_...	NS1|Numeric...	G	true	Boolean	8:50:09.663 ...	8:50:09.663 ...
2	manas-forte_...	NS1|Numeric...	Q	false	Boolean	8:50:01.347 ...	8:50:01.347 ...

Attribute	Value
NodeId	ns=1;i=319
NamespaceIndex	1
IdentifierType	Numeric
Identifier	319
NodeClass	Variable
BrowseName	1, "Q"
DisplayName	"en-US", "Q"
Description	"en-US", "Digi
WriteMask	0
UserWriteMask	0
RolePermissions	BadAttributeId
UserRolePermissions	BadAttributeId
AccessRestrictions	BadAttributeId
Value	
SourceTimestamp	16/06/19 8:49

Reference	Target DisplayName
HasTypeDefin...	BaseDataVariableType

```
Timestamp          Source      Server            Message
16/06/19 8:5...    DA Plugin   manas-forte_...   Found existing subscription for ServerId 0
16/06/19 8:5...    DA Plugin   manas-forte_...   Item [NS1|Numeric|319]: SamplingInterval=250, QueueSize=1, DiscardOldest=1, ClientHandle=3
16/06/19 8:5...    DA Plugin   manas-forte_...   CreateMonitoredItems succeeded [ret = Good]
16/06/19 8:5...    DA Plugin   manas-forte_...   Item [NS1|Numeric|319] succeeded : RevisedSamplingInterval=250, RevisedQueueSize=1, MonitoredItemId=2 [ret = Good]
16/06/19 8:5...    TypeCache   manas-forte_...   Reading type info of NodeId NS1|Numeric|319 succeeded
16/06/19 8:5...    DA Plugin   manas-forte_...   Write to node 'NS1|Numeric|317' succeeded [ret = Good]
```

Figure A.6: Testing with UaExpert

Download the FORTE source from http://git.eclipse.org/c/4diac/org.eclipse.4diac.forte.git:

```
$ mkdir ~/4diac && cd "$_"
$ git clone -b develop https://git.eclipse.org/r/4diac/org.
eclipse.4diac.forte forte
$ cd forte && mkdir build
```

Download the source for open62541 from https://github.com/open62541/open62541:

```
$ cd ~/4diac
$ git clone https://github.com/open62541/open62541.git
--branch=v0.3.0 open62541
```

Build open62541. If you are running the code on production devices we suggest setting the build type to Release.

```
$ cd ~/4diac/open62541 && mkdir build && cd $_
$ cmake -DBUILD_SHARED_LIBS=ON -DCMAKE_BUILD_TYPE=Debug
-DUA_ENABLE_AMALGAMATION=ON ..
$ make -j
```

Set FORTE to include open62541. If you are running the code on production devices we suggesst setting the build type to Release. If you are using the 0.2 branch of open62541 make sure that you set the correct value for FORTE_COM_OPC_UA_VERSION=0.2

```
$ cd ~/4diac/forte/build
$ cmake -DCMAKE_BUILD_TYPE=Debug
-DFORTE_ARCHITECTURE=Posix -DFORTE_MODULE_CONVERT=ON \
    -DFORTE_COM_ETH=ON -DFORTE_MODULE_IEC61131=ON
-DFORTE_COM_OPC_UA=ON \
    -DFORTE_COM_OPC_UA_INCLUDE_DIR=$HOME/4diac/open62541/build \
    -DFORTE_COM_OPC_UA_LIB_DIR=$HOME/4diac/open62541/build/bin \
    -DFORTE_COM_OPC_UA_LIB=libopen62541.so
$ make -j
```

Appendix B

OpenModelica

Sample OpenModelica script to test with OPC UA server

```
 model first
        input Real x;
        Real y;

equation
        y=x;

end first;
```

Appendix C

Python

Python script to get node and variable ids for OPC UA server on OpenModelica

```
import sys, traceback
sys.path.insert(0, "..")
import logging
from opcua import Client
from opcua import ua
from opcua import Node
from opcua import crypto
from opcua.tools import endpoint_to_strings

from time import sleep

if __name__ == "__main__":

        logging.basicConfig(level=logging.WARN)
        modelica = Client("opc.tcp://localhost:4841")

        try:
                modelica.connect()

                modelicaObject = modelica.get_objects_node()

                modelicaID = {}
                modelicaVars = {}

                i=0
                tmp={}

                modelicaID = modelicaObject.get_children()

                for i in range(len(modelicaID)):
                        modelicaVars[i] = modelicaID[i].get_browse_name()
                        if i>4:
```

```
                        tmp[i] = modelicaID[i].get_value()
                        print i, modelicaVars[i], tmp[i]
                    else:
                        print i, modelicaVars[i]
        finally:
            modelica.disconnect()
```

Python script to intiate communication between Raspberry Pi and OpenModelica

```
import sys, traceback
sys.path.insert(0, "..")
import logging
from opcua import Client
from opcua import ua
from opcua import Node
from opcua import crypto
from opcua.tools import endpoint_to_strings

from time import sleep

if __name__ == "__main__":

        logging.basicConfig(level=logging.WARN)
        modelica = Client("opc.tcp://localhost:4841")

        try:
                modelica.connect()
                modelicaObject = modelica.get_objects_node()
                runModelID = 2
                varWriteID=5    # ID of Modelica variable to write
                varReadID=6
                modelicaID = {}
                modelicaID = modelicaObject.get_children()
                modelicaID[runModelID].set_value(True)
                sleep(0.05)
                while True:
                        # sensorData = fun()
                        # modelicaID[varWriteID].set_value(sensorData)
                        ledOp =  modelicaID[varReadID].get_value()
                        print ledOp
                        sleep(0.05)
        except KeyboardInterrupt:
                print "Stopping␣Sequence"
        finally:
                modelica.disconnect()
```

Appendix D

PID Structured text code for 4DIAC

PID structured text code

```
VAR
  proportionalPart: REAL;
  integralPart: REAL;
  derivativePart: REAL;
  Ydesired: REAL;
  errorFiltered: REAL;
END_VAR;

  IF ABS(InError) <= Noise THEN
    errorFiltered := 0;
  ELSE
    errorFiltered := InError;
  END_IF;
  (* calculate proportional part *)
  proportionalPart := Kp * errorFiltered;

  (* run integrator *)
  integralPart := errorFiltered * Interval * Ki + INTEGRATION_ACCUM;

  (* run derivative*)

  derivativePart := (errorFiltered - LAST_ERROR) * Kd / Interval;

  LAST_ERROR := errorFiltered;

  (* calculate output Y *)
  Ydesired := proportionalPart + integralPart + derivativePart;

  (* check output for limits *)
  IF Ydesired >= LIM_H THEN
    Y := LIM_H;
    IF Ki <> 0.0 THEN
```

```
      IF (errorFiltered < 0 AND Kp > 0) OR (errorFiltered > 0 AND Kp < 0)
        INTEGRATION_ACCUM := integralPart;
      END_IF;
    ELSE
      INTEGRATION_ACCUM := 0.0;
    END_IF;
    LIM := TRUE;
  ELSIF Ydesired <= LIM_L THEN
    Y := LIM_L;
    IF Ki <> 0.0 THEN
      IF (errorFiltered > 0 AND Kp > 0) OR (errorFiltered < 0 AND Kp < 0)
        INTEGRATION_ACCUM := integralPart;
      END_IF;
    ELSE
      INTEGRATION_ACCUM := 0.0;
    END_IF;
    LIM := TRUE;
  ELSE
    Y := Ydesired;
    INTEGRATION_ACCUM := integralPart;
    LIM := FALSE;
  END_IF;
```

Appendix E

Arduino & Energia PID

Arduino implementation of PID for ATmega controllers

```
void setup()
{
  Serial.begin(115200); //serial begin
}

unsigned long lastTime;
double Input, Output, Setpoint;
double errSum, lastErr;
double kp = 15;
double ki =0;
double kd =0;
void Compute()
{
   /*How long since we last calculated*/
   unsigned long now = millis();
   double timeChange = (double)(now - lastTime)/10;

   /*Compute all the working error variables*/
   double error = Setpoint - Input;
   errSum += (error * timeChange);
   double dErr = (error - lastErr) / timeChange;

   /*Compute PID Output*/
   Output = kp * error + ki * errSum + kd * dErr;

   /*Remember some variables for next time*/
   lastErr = error;
   lastTime = now;
}


void loop()
```

```
{
  String readStr = ""; //some variables
  String readVal = "";


  if (Serial.available())
  { //when serial data comes from modelica
  while(Serial.available())
  {
    char readChar = (char)Serial.read();
    readStr+=readChar;
    if(readChar == '\n') break;
  } //read the data and store in a string
    for (int i = 1; i < (readStr.length()-1); i++)
    {
      readVal += readStr[i];
    }


      Setpoint = 100*sin(millis()*3.1412/(20*180))+100;
      //Setpoint = double(analogRead(A5));


    Input = readVal.toDouble(); //extract value
    Compute();
    Serial.println("1,"+String(Output)+"\n");
    delay(5);
  }
}
```

ARM implementation of PID for TIVA C series

```
#define AREAD_PIN A2

#define PID_OUTPUT_MIN -512.0
#define PID_OUTPUT_MAX 512.0
#define PID_KP 5
#define PID_KI 3
#define PID_KD 3
#define PID_BANG_BANG 40
#define PID_INTERVAL  100

#define I2C_WRITE                    B00000000
#define I2C_READ                     B00001000
#define I2C_READ_CONTINUOUSLY        B00010000
#define I2C_STOP_READING             B00011000
#define I2C_READ_WRITE_MODE_MASK     B00011000
#define I2C_10BIT_ADDRESS_MODE_MASK  B00100000
```

```
#define I2C_END_TX_MASK                B01000000
#define I2C_STOP_TX                    1
#define I2C_RESTART_TX                 0
#define I2C_MAX_QUERIES                8
#define I2C_REGISTER_NOT_SPECIFIED     -1
```

Appendix F

Sample C drivers for digital data exchange with OpenModelica

C drivers for digital data exchange for ATmega and OpenModelica

```
#include <errno.h>
#include <termios.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h> /* memset */
#include <stdio.h>
#include <stdlib.h>
#include "../Include/serial.h"
#include "../Include/digital.h"

int cmd_digital_out(int h,int pin_no,int val)
{
        int wr;
        char pin[6]="Da";
    char v[2],temp[2];
        sprintf(temp,"%c",pin_no+48);
        strcat(pin,temp);
        strcat(pin,"1");
    //printf("%s",pin);
        wr=write_serial(h,pin,4);
        if (val > 0.5)
        val = 1;
        else
        val = 0;

        sprintf(v,"%d",val);
        strcpy(pin,"Dw");
    strcat(pin,temp);
        strcat(pin,v);
    //printf("%s",pin);
```

```
        wr=write_serial(h,pin,4);
    return wr;
}

int cmd_digital_in(int h,int pin_no)
{
    int value = 0;
        char pin[6]="Da";
        char v1[2],v2[2];
        int wr1, wr2;
        sprintf(v1,"%c",pin_no+48);
        strcat(pin,v1);
        strcat(pin,"0");
        //printf("%s\n",pin);
        wr1=write_serial(h,pin,4);

        strcpy(pin,"Dr");
        sprintf(v2,"%c",pin_no+48);
        strcat(pin,v2);
        wr2=write_serial(1,pin,3);
        //binary transfer
        int stat;
        int num_bytes[2];
        char st[10];
        stat=status_serial(h,num_bytes);
        while(num_bytes[0]<1)
            stat=status_serial(h,num_bytes);
        char* temp;
        int wr=read_serial(h,st,1);
        value=strtod(st,&temp);
        //printf("%d\n",value);
        return value;
}
```

Appendix G

OPC UA on OpenModelica

G.1 Motivation

With time and growing demand, systems have become quite large, complex, and mathematically tedious to build. We continue to rely on age old tools but these aren't capable of analyzing, building such complex systems as the computational complexity and the control of such systems is beyond their capability. Also with the availability of large powerful engines, speed of computation has increased by a huge factor. So, as there is a availability of large computation power, the simulation and modeling tools can be used to analyze and build large and complex systems. OpenModelica [8] is such a tool which has become the standard of modeling and simulation to build complex systems. It's always too costly to perform experiments on real-time systems and most of the times the resources are also not properly used. Hence, it's has become a standard practice for academia and also for the industry to perform the virtual experiments on some modeling and simulation tools before going for the actual implementation of the plant setup. Modeling and simulation of real-time systems is cost-effective as compared to implementing real-time systems directly as it's quite a bit easier to add/ delete or modify the components as per the demand of the situation. The results from simulation become more real-time and practical if the simulation tool can interact with the external devices in real-time. And it does this by using the open62541 stack at its back-end. open62541 is a open-source library implemented in C/C++ for OPC UA architecture [7]. OPC UA being platform independent, can be used to interface with any of the external devices to connect with OpenModelica. OPC and its latest version OPC UA are specifications that can be used in communication among both software and hardware components in technical systems, especially in process control and manufacturing automation systems. In this chapter, a method for incorporating the OPC interfaces, especially OPC UA, into OpenModelica is implemented. In order to monitor variables efficiently in real-time, the OPC UA interface to OpenModelica simulations was developed. OPC UA is called the pioneer of Industry 4.0 [21]as it is a communication architecture aiming at the standardization of communication in industry.

G.2 Previous and Related Work

There used to be an OpenModelica OPC-UA/OPC-DA interface implemented around 2011 in [9], but it has number of issues problems:

1. Closed source code, making updates hard.
2. Windows-only, making it not run on the primary platform for OpenModelica (Linux).
3. Not maintained for several years.

OpenModelica used to have support for interactive simulations in the 1.5.0 release. Then it was using a custom protocol that was not suitable for real-time simulation and was not maintained for several years. The functionality of both OPC-UA/OPC-DA and interactive simulation was lost around the same time, when the simulation runtime was completely restructured since nobody working on OpenModelica could use the interface. It is also possible to use FMUs for interactive simulation. If FMI for co-simulation is used, this can be done in a straight-forward manner, just making a step, displaying variables, and synchronizing to real-time. However, OpenModelica does not support advanced numerical solvers for co-simulation FMUs. If FMI for modelica exchange is used, the interactive simulation tool would need to become a full-fledged FMI simulation tool like OMSimulator; but OpenModelica model exchange FMUs coupled with OMSimulator cannot simulate as many models as OpenModelica simulations at the moment.

G.3 OPC Interfaces

In OpenModelica, a model can be simulated through the OPC interfaces. It is possible to choose between two similar interfaces, OPC Data Access (DA) and OPC Unified Automation (UA). Both interfaces offer about the same functionality. However, OPC DA is a part of OPC Classic which is an older interface based on Microsoft Windows technology. OPC DA is a standardized way of communicating data in terms of values, time and other quality information, bound to Windows platforms. The communication takes place in a client-server model, which means that the OPC server and client communicate over a computer network. OPC UA is based on a client and a server communicating over a network. It is fully possible to connect several OPC clients to an OPC server. However, the OPC DA server in OpenModelica is, according to the documentation, currently broken. Instead, let us have a closer look at OPC UA which is currently in an experimental state.

OPC UA [22] is a standardized protocol for industrial communication in the ISO/IEC 62541:2015, OPC Unified Architecture. This standard aims to define information exchange between clients regardless of the hardware in use. The OPC Foundation[OPC2] released OPC UA in 2008 and it integrates the previous OPC Classic specification functionality into one unified architecture. OPC UA is backward compatible with OPC Classic as well as hardware independent. It can run on several platforms such as Windows, Linux, Android and Mac. In other words, OPC

UA can be used as a communication protocol between smaller embedded devices as well as between large network infrastructures. OPC UA can be used in both closed networks and over the Internet. Authentication, access control and encryption is built into the protocol for security measures in case it is intended for usage outside a closed network. OPC UA is functionality equivalent to OPC Classic but also extended with further capabilities. For example, it is possible to invoke a Remote Procedure Call (RPC) which makes it possible to call functions and execute programs on the server side, from a client. Another interesting feature is subscriptions. A client can subscribe to a server and monitor interacting data. If the interacting, monitored data item changes, it will be reported back from the server to the client. This feature can therefore reduce the amount of network traffic by only sending data as it becomes relevant to the client. It can be valuable in different aspects in terms of reduced overhead cost for a small embedded device or reduced network traffic for a large scale cloud infrastructure. Another new functionality added to OPC UA is the ability for a OPC UA client to discover OPC UA servers on a local computer or network, or both. All data on a OPC UA server is represented hierarchically using folders. A folder can contain other folders and data items.

Furthermore, an important difference between OPC DA and OPC UA is the address space model. The primary object for the address space is to provide a standardized way for the server to represent objects to clients. It contains metadata about the server as well as data items, referred to as nodes. Each node is assigned to a node class which in turn represent a different object in the object model. Nodes are especially interesting because they are described by attributes. They can also contain information about the relation to other nodes. However, attributes contains information about each node such as the node id, value and data type. They can be accessed by a client using the read, write, query or subscription services. There is also possible to protect attributes of a node from being written by the WriteMask attribute. It can be useful in cases where read-only nodes are exposed to clients.

G.4 Implementation

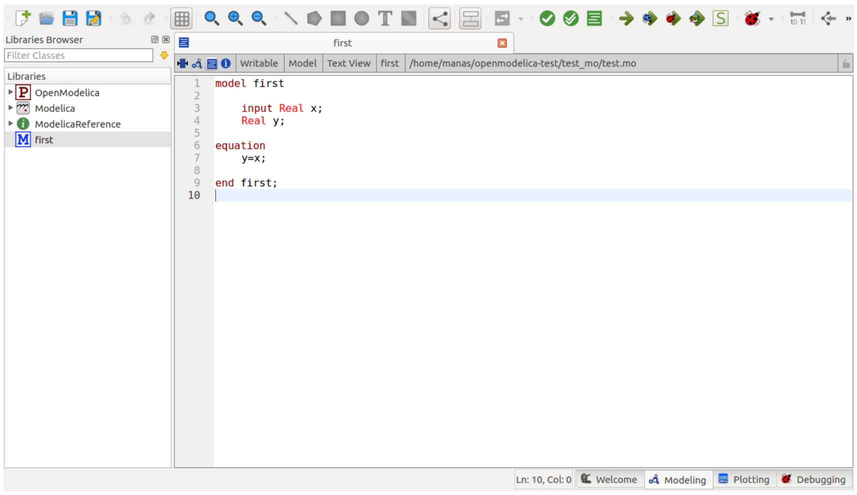
open62541, which is an open-source OPC UA implementation, is being used in the backend of OpenModelica. OPC UA in OpenModleica can be activated by using the simulation flag **-embeddedServer=opc-ua**. Inside OpenModelica, internally in the simulation, open62541 runs on a separate thread created and it samples variables chosen by the user to monitor, and performs the communication with OPC UA clients which may be an another software or hardware. This approach has a negligible effect on the performance of the actual simulation and will not interfere with the real-time properties of the simulation assuming that the user has a separate processor core to spare for the thread sampling the simulation variables. The simulation can be controlled by setting variables through the following OPC UA interfaces:

- OpenModelica.step
- OpenModelica.run Runs asynchronously and synchronizing to real-time after each time step

- OpenModelica.realTimeScalingFactor 0.0 disables, 1.0 synchronizes in real-time
- OpenModelica.enableStopTime When disabled, simulation continues without stop-time

G.5 Testing with UaExpert

UaExpert [10] is an C/C++ based OPC UA client with a graphical user interface. It can connect to any of the OPC UA server running on the local host and also anywhere on the network. Let's take up an example to explain it better as shown in Figure: G.1 and in Figure: G.2. This will help to understand better how the OPC UA client server communication between OpenModelica and UaExpert works.



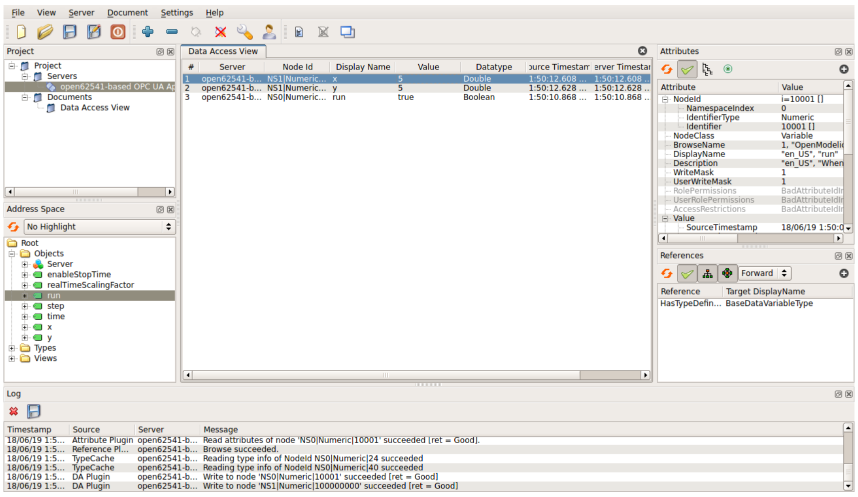
```
model first

    input Real x;
    Real y;

equation
    y=x;

end first;
```

Figure G.1: OM Code Test with UaExpert



#	Server	Node Id	Display Name	Value	Datatype	ource Timestam	erver Timestan
1	open62541-b...	NS1|Numeric...	x	5	Double	1:50:12.608 ...	1:50:12.608 ...
2	open62541-b...	NS1|Numeric...	y	5	Double	1:50:12.628 ...	1:50:12.628 ...
3	open62541-b...	NS0|Numeric...	run	true	Boolean	1:50:10.868 ...	1:50:10.868 ...

Attribute	Value
NodeId	i=10001 []
NamespaceIndex	0
IdentifierType	Numeric
Identifier	10001 []
NodeClass	Variable
BrowseName	1, "OpenModelic
DisplayName	"en_US", "run"
Description	"en_US", "When
WriteMask	1
UserWriteMask	1
RolePermissions	BadAttributeIdIı
UserRolePermissions	BadAttributeIdIı
AccessRestrictions	BadAttributeIdIı
Value	
SourceTimestamp	18/06/19 1:50:0

Reference	Target DisplayName
HasTypeDefin...	BaseDataVariableType

```
Timestamp         Source            Server           Message
18/06/19 1:5...   Attribute Plugin  open62541-b...   Read attributes of node 'NS0|Numeric|10001' succeeded [ret = Good].
18/06/19 1:5...   Reference Pl...   open62541-b...   Browse succeeded.
18/06/19 1:5...   TypeCache         open62541-b...   Reading type info of NodeId NS0|Numeric|24 succeeded
18/06/19 1:5...   TypeCache         open62541-b...   Reading type info of NodeId NS0|Numeric|40 succeeded
18/06/19 1:5...   DA Plugin         open62541-b...   Write to node 'NS0|Numeric|10001' succeeded [ret = Good]
18/06/19 1:5...   DA Plugin         open62541-b...   Write to node 'NS1|Numeric|100000000' succeeded [ret = Good]
```

Figure G.2: UaExpert client connected to OM

An example can help to obtain better understanding about OPC UA and how it works. The following example uses UaExpert which is an OPC UA test client with a GUI. Figure: G.2 depicts UaExpert connected to the simulation of the model on OpenModelica. The address space to the left contains the exposed nodes in the address space model hierarchy. Attributes related to the variable node step have been read and the response is shown in the attributes area to the right. Several attributes are available as Figure: G.2 illustrates, including the attribute NodeClass which defines that step is a variable node. OPC UA is a communication protocol which is platform independent. It is quite possible to implement this architecture in a number of ways and also by using different programming frameworks. OpenModelica at its back end uses open62541 for OPC UA server implementation. The reason for choosing the open62541 library for this implementation is it is open-source and quite well-documented. The other available libraries are not well-documented, though they are open-source. Also, OpenModelica being an open-source entity, it has a third party package which has to be open-source and not a closed package. open62541 is an OPC UA protocol written in C. As mentioned earlier, OpenModelica is open source which means that the third party libraries available is limited to the open source domain. Also, this very protocol is intended to be platform independent. That includes hardware and OS independence. A stable and exact implementation of OPC UA was required therefore to be able to communicate with different clients and servers independent of their actual implementation. But users can go for any other implementation of OPC UA if they think it better than open62541. open62541 is the backbone of OPC UA architecture for OPC UA in OpenModelica.

Bibliography

[1] Arduino home. https://www.arduino.cc/. Accessed: 2018-10-25.

[2] ATmega16 data sheet. http://ww1.microchip.com/downloads/en/DeviceDoc/doc2466.pdf. Accessed: 2018-07-15.

[3] GitHub firmata processing. https://github.com/firmata/processing. Accessed: 2018-06-20.

[4] Interoperability c and python. https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/interop_c_python.html. Accessed: 2018-05-23.

[5] Modelica information. https://build.openmodelica.org/Documentation/Modelica.html. Accessed: 2017-09-30.

[6] OPC Foundation opc unified architecture. https://opcfoundation.org/wp-content/uploads/2016/05/OPC-UA-Interoperability-For-Industrie4-and-IoT-EN-v5.pdf. Accessed: 2019-05-03.

[7] open62541 an open source implementation of opc ua. https://open62541.org. Accessed: 2019-03-25.

[8] OpenModelica introduction. https://www.openmodelica.org/. Accessed: 2018-09-30.

[9] Tuomas Miettinen opc interfaces in openmodelica – technical specification (task 5.3). tech. rep. 2011. https://github.com/OpenModelica/OpenModelica-doc/blob/476cafa/opc/OPC_Interfaces_in_OpenModelica.doc. Accessed: 2019-02-02.

[10] Uaexpert—a full-featured opc ua client. https://www.unified-automation.com/products/development-tools/uaexpert.html. Accessed: 2019-05-03.

[11] Marco Bonvini, Filippo Donido, and Alberto Leva. Modelica as a design tool for hardware-in-the-loop simulation. In *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009*, number 043, pages 378–385. Linköping University Electronic Press, 2009.

[12] Mako D Cvetkovic and Milun S Jevtic. Interprocess communication in real-time linux. In *Telecommunications in Modern Satellite, Cable and Broadcasting Service, 2003. TELSIKS 2003. 6th International Conference on*, volume 2, pages 618–621. IEEE, 2003.

[13] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, 10(4):2233–2243, 2014.

[14] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach.* John Wiley & Sons, 2014.

[15] Wojciech Grega. Hardware-in-the-loop simulation and its application in control education. In *Frontiers in Education Conference, 1999. FIE'99. 29th Annual*, volume 2, pages 12B6–7. IEEE, 1999.

[16] Vasile Gheorghita Gaitan Ioan Ungurean, Nicoleta-Cristina Gaitan. An iot architecture for things from industrial environment. *2014 10th International Conference on Communications (COMM)*, 10(4):2233–2243, 2014.

[17] Rolf Isermann, Jochen Schaffnit, and Stefan Sinsel. Hardware-in-the-loop simulation for the design and testing of engine-control systems. *Control Engineering Practice*, 7(5):643–653, 1999.

[18] Pieter Mosterman. Model-based design of embedded systems. In *IEEE International Conference on Microelectronic Systems Education*, pages 373–389. IEEE, 2007.

[19] Kannan M. Moudgalya. *Digital Control.* John Wiley & Sons, 2007.

[20] Luis Ribeiro. Cyber-physical production systems' design challenges. In *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, pages 1189–1194. IEEE, 2017.

[21] Miriam Schleipen, Syed-Shiraz Gilani, Tino Bischoff, and Julius Pfrommer. Opc ua & industrie 4.0-enabling technology with high diversity and variability. *Procedia Cirp*, 57:315–320, 2016.

[22] Michael H Schwarz and Josef Börcsök. A survey on opc and opc-ua: About the standard, developments and investigations. In *2013 XXIV International Conference on Information, Communication and Automation Technologies (ICAT)*, pages 1–6. IEEE, 2013.