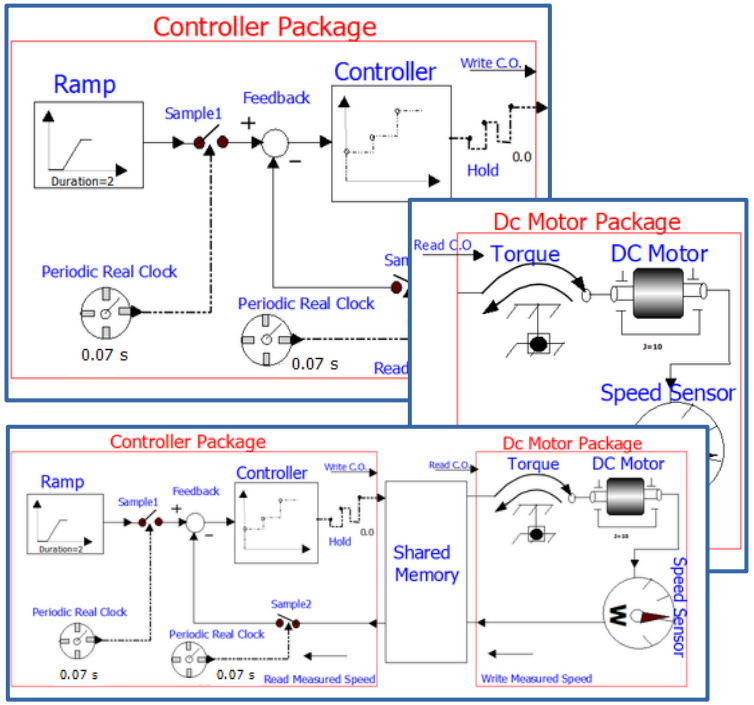
# INTER PROCESS COMMUNICATION v1.0

## A MODELICA MODULE FOR INTER PROCESS & INTER SYSTEM COMMUNICATION

USER MANUAL
2017

www.modelicon.in

# Contents




- Objective
- Introduction
- Library Description
- Library Architecture
- SharedMemory Library
- Making a Model
- Examples Package
- Execution Steps
- Results
- Development
- Limitations and Future Development
- Credits

# Objective

OPENMODELICA is an open-source Modelica-based modeling and simulation environment intended for industrial and academic usage. The goal with the OpenModelica effort is to create a comprehensive Open Source Modelica modeling, compilation and simulation environment based on free software distributed in binary and source code form for research, teaching, and industrial usage. (*src. www.openmodelica.org*). We at Modelicon Infotech India are using OpenModelica extensively for our internal usage and we felt a need for a fast and reliable Inter Process Communication (IPC) and inter system communication library. Though OpenModelica maintains a very good TCP/IP based OPC-UA library for inter system communication, but our requirement needed more number of samples to be transmitted per second for a real time application.

Thus to cater our needs we developed this library that allows us to transmit and receive at a baud rate of up to 115200bps while sampling at every 0.05s. Not just this, we were able to communicate with several other processes such as HMIs on the same system using shared memory at a very fast rate. In an attempt to contribute to the open source community Modelicon Infotech is making this library public. Following document contains all the steps and essentials required for using this library. This library is made open source under GNU GPLV3 license and you are free to modify the library.



# Introduction

Processes can be of two types: Independent process and Co-operating process. An independent process does not depend on any data other than of its own program. A Co-operating process however needs data to be exchanged with other processes, local or remote. Inter Process Communication (IPC) is a set of programing interfaces that allows a co-operating process to interact with other processes in an operating system. IPC in general is used to handle multiple demands for a source or data concurrently. There are various IPC methods and each one has its own advantages and limitations. To name a few, pipes, named pipes, message queueing, semaphores, shared memory and sockets.

InterProcessCommunication library is created for OpenModelica which is an open-source Modelica-based modeling and simulation environment. In current version of InterProcessCommunication library we have implemented IPC using shared memory. Shared memory is that part of memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. By this method program processes can exchange data more quickly than by reading and writing using the regular operating system services. Different OS provide their own APIs for assigning and accessing shared memory. This library is a solution prepared for UNIX based systems as well as windows-PC.



# Library Description

- “InterProcessCommunication” is an OM library for exchanging data efficiently between two different systems running their own instances of OpenModelica. Systems can be windows PC or Linux Ubuntu or Raspberry Pi SOC.
- For making data transfer we have used Shared memory for inter process communication (IPC) and RS232 serial port programming for inter system communication.
- Enclosed files contains one package called “InterProcessCommunication” for the demonstration of this idea. It has two models, “Discrete PID” and “DC Motor” model. You can run both the models separately on two systems and exchange data using shared memory and serial communication.
- We have tested and prepared executables for Raspberry Pi and OM models are available for Linux Ubuntu and Windows as well.
- Currently the transfer is being done at the baud rate of 115200bps.
- Its can also be used for HIL simulations, small controllers, data transfer to and from HMIs and other applications.



# Library Architecture

- Library consists of 3 main components.
- **First** is the OpenModelica Model. It can be any model that has at least one output or/and one input seeking data fro other processes. For demonstration we have “DiscretePID” and “DCMotor” models. Either one of them can be compiled and executed on system 1 and the other one on system 2.
- **Second** is the OpenModelica Shared Memory functions those call the C program MIShm.c externally for reading from or writing data to shared memory defined in the program using OS APIs.
- **Third** is the serial communication C program Serial_Shm.c that executes in isolation to OpenModelica. It reads data from shared memory and writes it to serial port and vice versa.

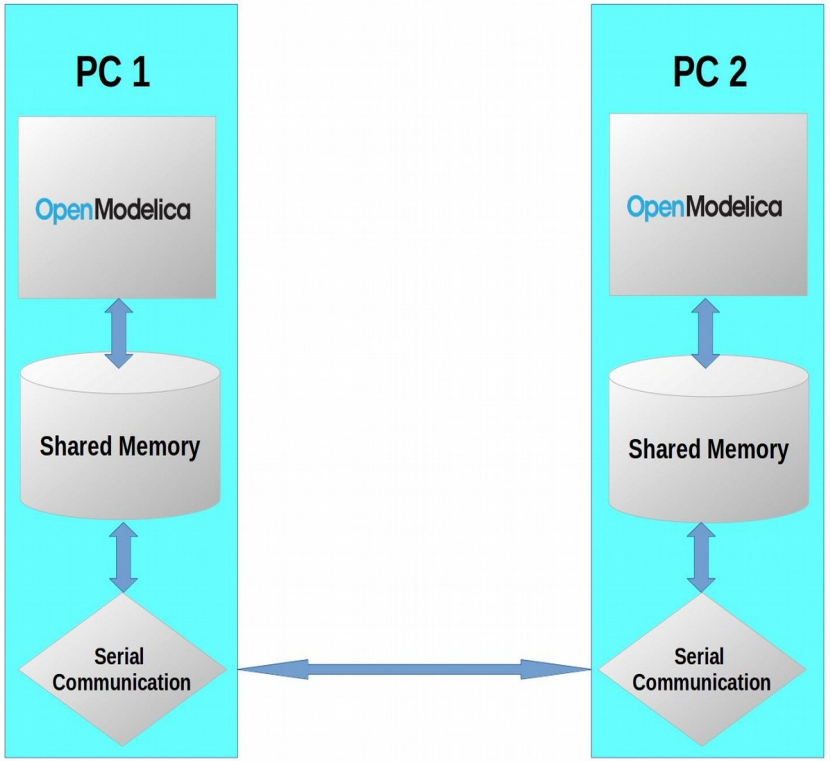



Fig.1 Architecture

# Files In SharedMemory Package

- **Following are the essential files and functions required for the implementation:**

- **SharedMemory**//These function implement the IPC shared memory
  - SharedMemoryRead() //external function for reading shared memory data
  - SharedMemoryWrite() //external function for writing shared memory data

- **Resources**
  - Include //Files included by external function
    - Serial_SHM.c //Serial and shared memory C program, its executable runs in the background
    - SerialMI.h //Serial programming header file
    - ShmMI.c //Shared memory C program included externally by SharedMemory_Functions
  - Library
    - Librt.so and librt.a //rt files only for UNIX based systems. Copied form system files.



# SharedMemory Package

- **SerialMemoryRead( <address> )**
  - External function to read data from given address in shared memory
  - Address can be any integer from 0 to 10 but it should match the address given to SerialMemoryWrite() function in the other model
  - Uses shmRead( int <num>) function from C program ShmMI.c

- **SerialMemoryWrite( <address>, <data> )**
  - External function to write data to given address in shared memory
  - Address can be any integer from 0 to 10
  - Uses shmWrite( int <num>, double <value>) function from C program ShmMI.c



# ShmMI.c

- C program for using shared memory.
- It uses POSIX and windows API for shared memory and is called externally by OpenModelica SharedMemoryRead() and SharedMemoryWrite() functions.
- Main functions are:
  - **_shmAccess()**:
    - Shared memory is named and allocated in this function.
    - If there is any error in allocating space this function returns error and stops rest of the execution
  - **_shmRead( addr )**:
    - This function is used for reading data stored in shared memory location pointed by the address
  - **_shmWrite( addr, data)**:
    - This function is used for writing the data on to the shared memory location pointed by the address.
    - Incoming double data is converted to string before storing. String contains the address and data separated by ‘,'.



# Serial_SHM.c

- C program to read data from shared memory and send it to other system using serial port and receiving data from other system and writing it to the shared memory. Its functions are defined in an header file SerialMI.h
- Main functions are:
  - **_serialBegin(“deviceID”, baudrate)**:
    - Takes device ID (string) and baudrate (int) as input
    - By default deviceID is set to /dev/ttyUSB0 and baudrate to 115200
    - You can modify as explained on pg. 13
  - **_serialRead()**:
    - Reads serial data and returns a string of data to the main function when called.
  - **_serialWrite( data )**:
    - Writes a string of data on to the serial port.
  - **_serialAvailable()**:
    - Returns a non zero number if data is available in serial buffer.
  - It also has shmAccess(), shmRead() and shmWrite() functions whose functionality is same as described on pg. 11



# Configuring Serial Port

- You will need two USB to serial converters, one each for two computers to be connected and 3 F-F jumper cables. Connect them in following fashion or equivalent if you have any other device.

| System 1 | System 2 |
|---|---|
| Gnd | Gnd |
| Rx | Tx |
| Tx | Rx |

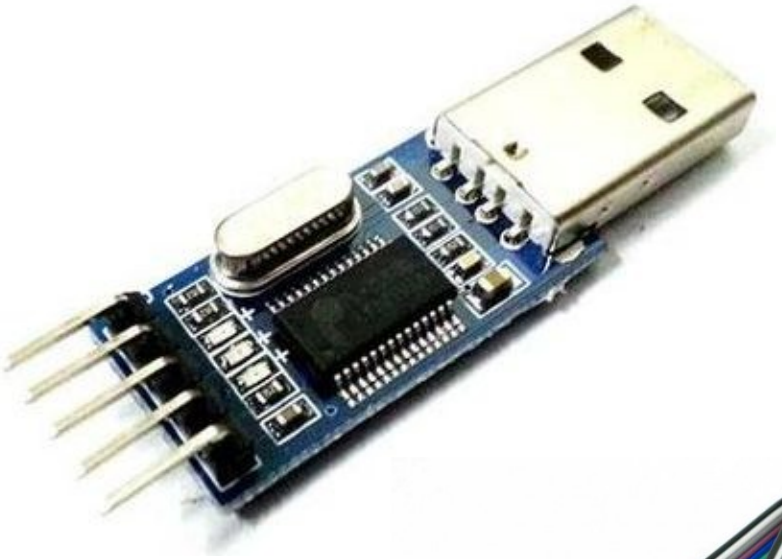


Fig.2 PL2303

Fig.3 F-F Jumper

# Serial Port Settings (Linux)

- **Linux Ubuntu and Raspbian:**
  - Connect USB to serial device or FTDI device to the computers
  - Find its address using command ls /dev/
  - It will probably be ttyUSB0 or ttyUSB1
  - Go to directory Linux/InterProcessCommunication/Resources/Include/ and find the c file Serial_SHM.c in it
  - Compile the program using command gcc -o Serial_SHM Serial_SHM.c -lrt
  - Execute the executable file by using command ./Serial_SHM
  - Fill in the device ID (e.g. /dev/ttyUSB0) and baud rate (e.g. 115200) when prompted to do so. It will establish the connection.
  - Keep it running in the background
  - It reads data from shared memory and writes it to the serial port and vice versa in a continuous loop until stopped manually or loss of connection because of hardware



# Serial Port Settings (Windows)

- **Windows:**
  - Connect USB to serial device or FTDI device to the computer
  - Find its address in device manager, install drivers if required
  - If drivers detected then address will be like COMx where x=2,3,4...
  - Go to directory Windows/InterProcessCommunication/Resources/Include/ and find the c file Serial_SHM.c in it
  - Compile the program using any C compiler
  - Execute the Serial_SHM.exe
  - Fill in the device ID (e.g. COM5) and baud rate (e.g. 115200) when prompted to do so. It will establish the connection.
  - Keep it running in the background
  - It reads data from shared memory and writes it to the serial port and vice versa in a continuous loop until stopped manually or loss of connection because of hardware



# Save Librt.So File

- This step is to be followed only by Linux and Raspberry Pi sytem users and only one time for a model. librt.so is a system specific file thus needs to be changed for each system.
- **Linux**
  - Go to root directory
  - Search librt* and copy the librt.so and librt.a files those show up in search results.
  - Paste those files in the directory: Linux/InterProcessCommunication/Resources/Library/linux64/

- **Raspberry Pi**
  - Go to root directory
  - Search librt* and copy the librt.so and librt.a files those show up in search results.
  - Paste those files in the directory: RaspberryPi/InterProcessCommunication/Resources/Library/



# Making A New Model

1) Build a complete model in OpenModelica using OMEdit
2) Compile and run it to observe the ideal behavior
3) Split the model into two parts and build them on different systems, as per your requirements
4) Include the essential files and functions as mentioned in previous pages into both the models
5) Use SharedMemoryRead() and SharedMemoryWrite() functions for the data transfer to shared memory and run the Serial_SHM.c executable for serial data transfer.
6) Compile and run the programs at same time on both the systems
7) Observe the output and validate it with the ideal output



# Files In Examples Package

- **Following are the packages used for demonstration of IPC shared memory:**

- **InterProcessExamples**
  - **DCMotor**.mo //DC Motor Example Model. Models Torque, Inertia and SpeedSensor from Modelica library are used and modified.

  - **DiscretePID**.mo //Discrete PID Example Model. Models Step, Feedback, PID and Filter from Modelica library are used and modified.

- **CombinedExamples** //This package is the combination of DiscretePID and DCMotor models and contains models for the validation of the output of DiscretePID and DCMotor models
  - **PIDandMotor**.mo //Combination of PID and DC motor model for verifying results. Models Torque, Inertia, SpeedSensor, Step, Feedback, PID and Filter from Modelica library are used and modified.
  - **SinPIDandMotor**.mo //PIDandMotor model with sinusoidal input and different inertia. Models Torque, Inertia, SpeedSensor, Sine, Feedback, PID and Filter from Modelica library are used and modified.



# Inter Process Communication Examples Package

- The overall objective of the examples package in InterProcessCommunication library is to demonstrate the closed loop speed tracking of DC motor using discrete PID controller, with the help of shared memory and serial communication.
- Let us assume that DiscretePID model is being run on system 1 and DCMotor model on system 2.
- The DCMotor model contains the DC motor, along with a speed sensor. The speed sensor measures the speed of the DC motor, which is written into the shared memory of system 2. A serial communication executable Serial_SHM running on system 2 reads this value from shared memory and transfers it to serial port and then to system 1 via serial communication device. Similar Serial_SHM executable running on system 1 reads this serial data and writes it to the shared memory of system 1.
- The DiscretePID model running on system 1 reads this data from the shared memory and based on the difference between set-point and measured speed, the Discerete PID controller generates control signal. This control signal is written into the shared memory of system 1, which is read and transmitted serially to system 2. The control signal acquired by the DCMotor model, acts as an input to the DC motor. Therefore, the complete closed loop speed tracking of DC motor can be achieved.



# DiscretePID.mo

- The DiscretePID model contains discretized PID controller along with filter. Sample rate is 0.05s by default. It is essentially a PID controller as given in Modelica library with K, Ti, Td values tuned to suit the application.
- The **DiscretePID** model demonstrates how the components of a discrete PID can be used to establish InterProcess Communication with DCMotor model using shared memory and serial communication.
- For intersystem communication run DiscretePID model on PC-1 and DCMotor model on PC-2.

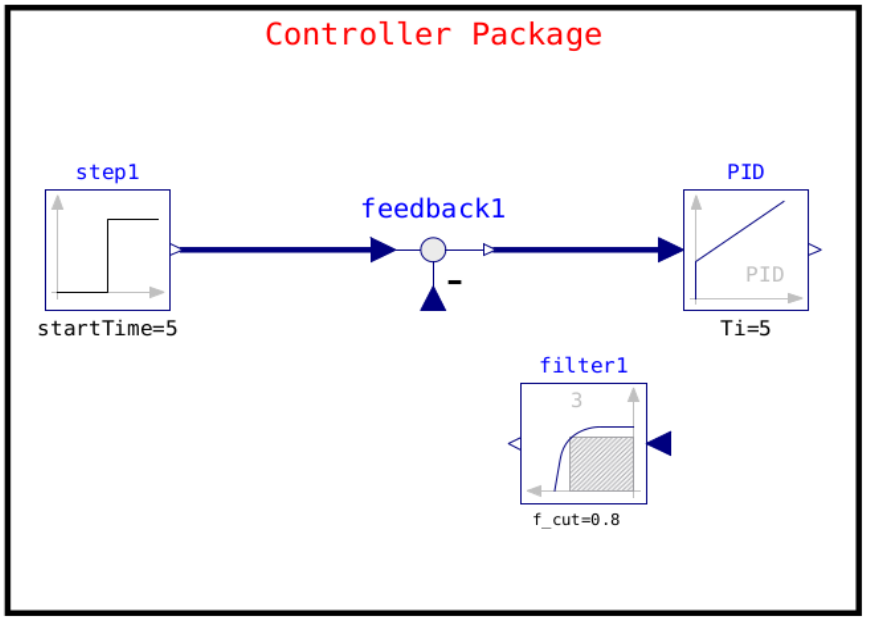



Fig.4 Discrete PID Controller

# DiscretePID Model

- **Input:**
  - A unit step signal with amplitude 20 and offset of 5s
- **Feedback:**
  - Value measured by speed sensor in DCMotor model
  - This value comes via serial port and is stored in shared memory space
  - Modelica model reads it from the memory using SharedMemoryRead() function. Address of memory goes as argument to the function. Address can be a number between 1 to 10 (dependent on address provided to Output of the other model)
  - This value is filtered using critical filter and then given as feedback
- **Output:**
  - Discrete PID controller value
  - SharedMemoryWrite() function writes the output to the shared memory space. Address (1-10) and variable name are passed as argument to the function
  - Serial_SHM.c reads those values and transmits it over the serial port



# DCMotor Model

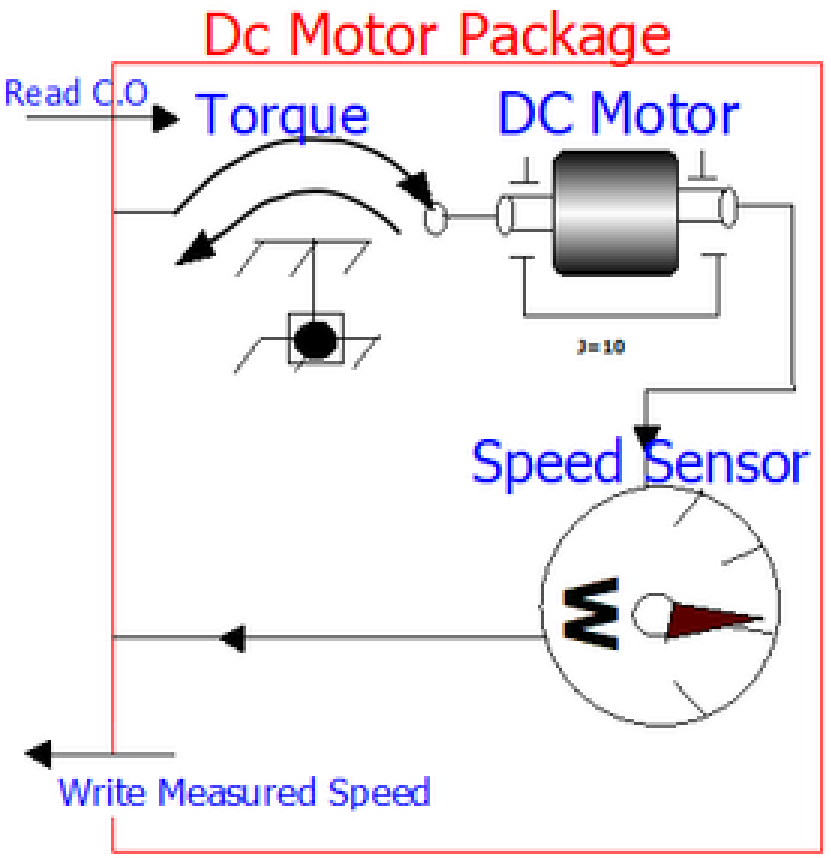



Fig.5 DC Motor Package

- The DCMotor model contains an inertia, which is attached with the speed sensor.
- The **DCMotor** model demonstrates how the it can be used to establish Inter Process Communication (IPC) using shared memory with DiscretePID model.
- For inter-system communication run DiscretePID_SM_Example model on PC-1 and DCMotor_SM_Example model on PC-2.



# DCMotor.mo

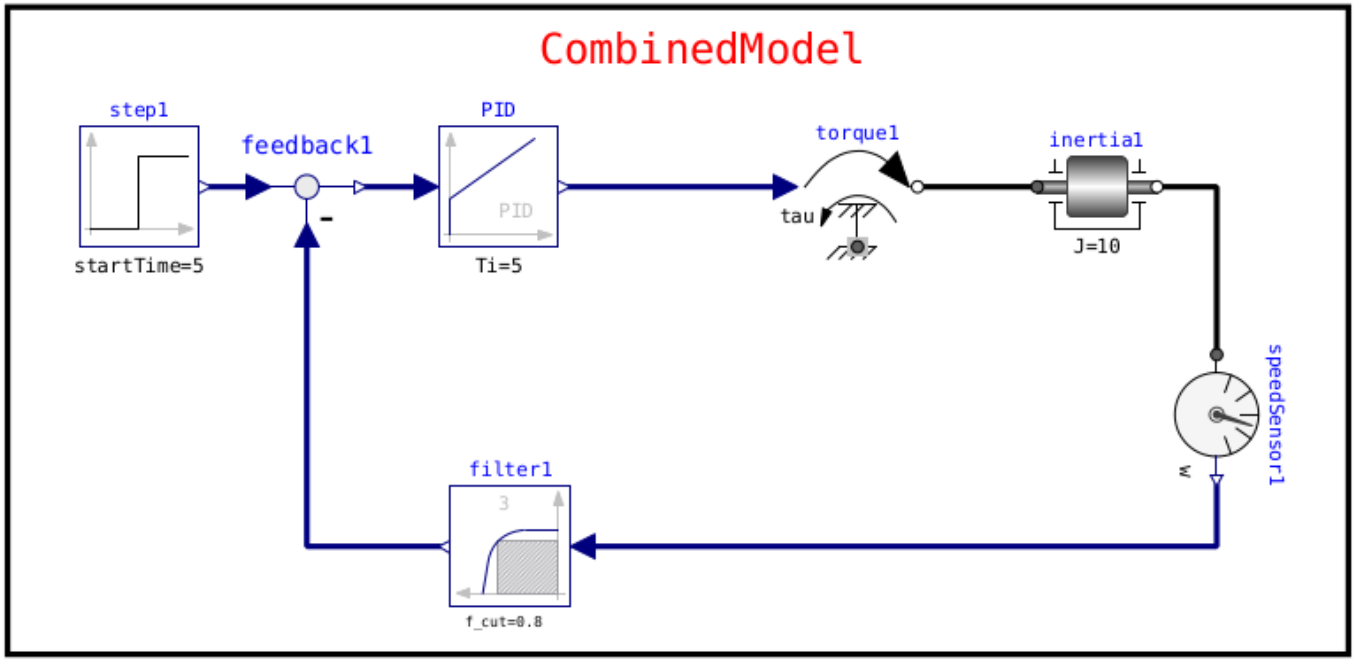
- **Input:**
  - Discrete PID controller value from DiscretePID model
  - This value comes via serial port and is stored in shared memory space
  - Modelica model reads it from the memory using SharedMemoryRead() function. Address of memory goes as argument to the function. Address can be a number between 1 to 10 (dependent on address provided to Output of the other model)

- **Output:**
  - Value measured by speed sensor
  - SharedMemoryWrite() function writes the output to the shared memory space. Address (1-10) and variable name are passed as argument to the function
  - Serial_SHM.c reads those values and transmits it over the serial port



# CombinedExamples Package

- This package contains combined DiscretePID and DCMotor models.
- It has all the components of both the models
- It does not require shared memory or serial port programming
- It is used for validation of the DiscretePID and DCMotor models by superimposing its results on to the results obtained after implementing inter process and inter system communication.
- It has two models namely: PIDandMotor and SinPIDandMotor. Difference lies in the source PIDandMotor model has step signal source and SinPIDandMotor has sinusoidal signal source.



# Execution Steps

1) Connect two systems via serial devices
2) Change serial port settings
3) Launch OpenModelica on both the systems
    1) Windows: Double click on OpenModelica Editor icon
    2) Linux/ Pi: type “sudo OMEdit” in the terminal
4) Open the example model (DiscretePID on one system and DCMotor on other system)
5) Change the simulation flags to real time (-rt=1) and change the start and stop times
6) Press simulate
7) Plot output
8) Validate results against PIDandMotor models output



# Results (Step Input)

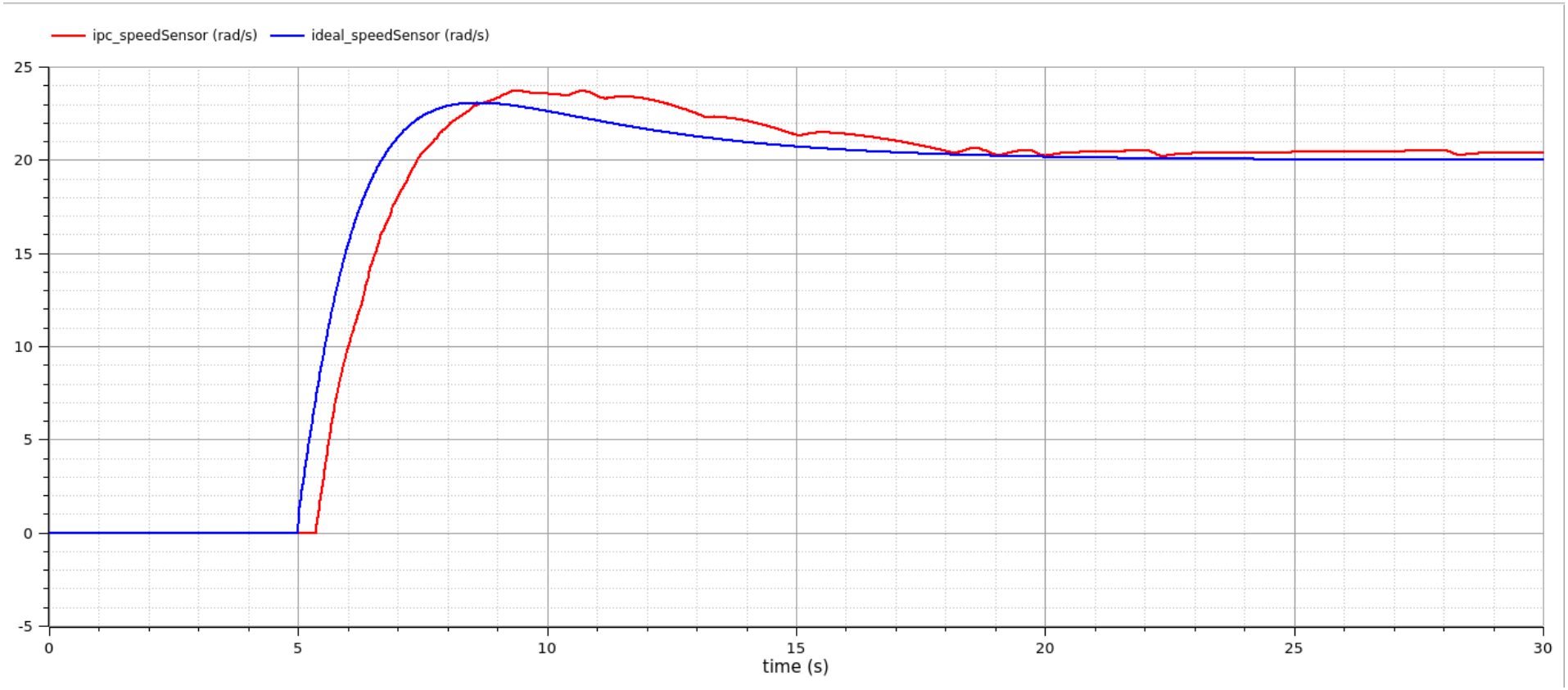



Fig. 6 Results with step input
Red represents the speed sensor value for interprocess communication model
Blue represents the speed sensor output for combined model

# Results (Sine Input)

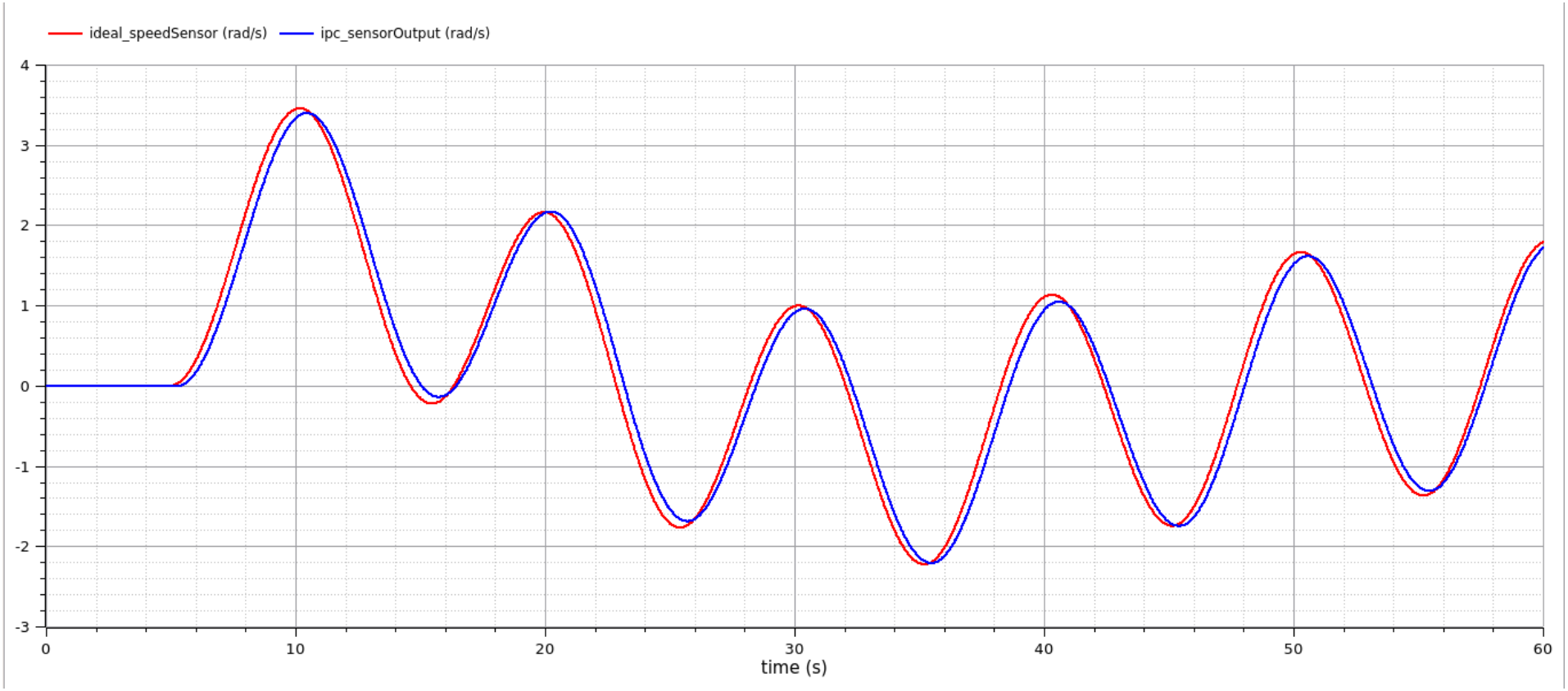



Fig. 6 Results with sine input

Blue represents the speed sensor value for interprocess communication model

Red represents the speed sensor output for combined model

# Development Details

- This library is built on basic libraries provided in OpenModelica such as:
  - PID, Step Source, Sine Source, Filter
  - Torque, Inertial, Speed Sensor
- In first phase library for sharing data using Shared Memory was made using external function. With that solution inter process communication was realized. Data was shared between two instances of OpenModelica running on same system or between OpenModelica and other HMI softwares.
- In second phase library for serial communication was added. At first it was appended to the shared memory library. But it did not work fine. Later we made separate executable for serial communication which is to be executed independent of OpenModelica and is not called by external functions.



# Limitations And Future Development

- At lower baud rates (<9600) we encountered huge drop of data even for lower sampling rate (1 sample per second). We experimented with baud rate and made output better and current implementation is working at a baud rate of 115200 and sample time of 0.05s with very less data loss.
- For further improving the response filter was added to filter out the feedback value and reduce effect of data loss.
- Next phase in development is to provide clock synchronization between two systems. This addition will enable diagnosis of the received data more accurately.
- Modbus-RTU will be used for establishing communication between two systems.
- More use cases will tested for the library.



# Credits

- **ModeliCon Infotech Team**
  - Shushmita (Intern)
  - Namrata (Intern)
  - Ankur Gajjar
  - Shubham Patne
  - Jal Panchal
  - Ritesh Sharma
  - Pavan P

- **License:**





For any queries write in to info@modelicon.in