AC 2009-2441: TEACHING REAL-TIME EMBEDDED SYSTEMS NETWORKING AND ASSESSMENT OF STUDENT LEARNING

Zaydoun Rawashdeh, Wayne State University

Zaydoun Rawashdeh has received his Master's degree from the University of Michigan. He is currently a Ph.D candidate in the Department of Electrical and Computer Engineering at Wayne State University. Since 2007 he has been working as a Teaching Assistant in the department. His research area is Vehicular Ad hoc Networks.

Syed Masud Mahmud, Wayne State University

Syed Masud Mahmud received the Ph.D. degree in electrical engineering from the University of Washington, Seattle, in 1984. Since 1988, he has been with Wayne State University, Detroit, Michigan. Currently, he is an Associate Professor of Electrical and Computer Engineering Department. During the last 20 years, he has been working in the areas of hierarchical multiprocessors, hierarchical networks, performance analysis of computer systems, digital signal processing, embedded systems, in-vehicle networking, performance analysis of networking protocols, secure wireless communications, and privacy protected vehicle-to-vehicle communications and simulation techniques. He has supervised a number of projects from Ford Motor Company and other local industries. He also served as a Co-PI on two NSF funded projects. He has published over 100 peer-reviewed journal and conference proceeding papers. He has supervised four Ph.D. dissertations and 8 MS theses.

Dr. Mahmud received the President's Teaching Excellence Award of Wayne State University in 2002. He also received several other teaching excellence awards within the college of engineering. He has served as a Technical Reviewer for many conferences, journals, and funding agencies. Currently, he is the Editor of the SAE Transactions on Passenger Cars: Electrical and Electronic Systems. Since 2008 he has also been serving as an ABET program evaluator. He is a senior member of IEEE. He is also a member of SAE, ASEE, Sigma Xi and Tau Beta Pi. He has been listed in the Who's Who in Science and Engineering Empowering Executives and Professionals, and many others.

Abstract

Today, embedded systems and embedded networking are common in manufacturing, complex vehicles, medical equipment, and home appliances, but few undergraduate engineering and technology curricula teach courses devoted to real-time embedded systems networking. Not having appropriate educational experiences risks a decline in U.S. technical expertise. Various companies such as Vector Corporation and Dearborn Group have developed commercial software packages for analysis, diagnosis and simulation of real-time embedded systems networking protocols. These companies also offer short training courses in embedded systems networking. The intended audiences of these training courses are the experienced engineers working in the industry but not the inexperienced undergraduate students. Although the commercial software packages are very complex and time consuming for the students to learn on their own, they can be taught (delivered) to the students in a simple way. This can be done by focusing on the major components that are considered the base to build any real-time embedded systems network. In this paper, we present an instructional material for CAN Open Environment (CANoe) software. This software can be used to develop, test, and simulate CAN networks. The instructional material basically summarizes the main components that are needed to build CANoe application in five labs. These labs represent the major phases in the CANoe application development cycle. Students, after learning this material, can move forward to develop more sophisticated applications on their own. We have also done assessment analysis for student learning. Based on our analysis it is concluded that our semester-long teaching and hands-on laboratory exercises greatly enhanced student learning. Detailed descriptions of our teaching materials along with our teaching methodologies are presented in the paper. The paper also presents the assessment of student learning. Our work may motivate other instructors around the country to develop similar courses to teaching real-time embedded systems networking.

Introduction

We are living in a world today in which rapid technological change is occurring faster than even our universities can keep up with. If we look back 50 years, we see an automobile industry in which almost all systems are mechanically based. Fast-forward to today, and industry projections are showing that by 2010, 40% of the total cost of a car will be dedicated to electronics and distributed electronic control systems. During this time span, individual manufacturers have developed their own processes and their own proprietary electronic architectures and protocols. Many of these standards did not make it out of their company's doors, but we finally have a situation where certain protocols are used globally. This global industry standardization did not come about until around 10 years ago. After a decade of preferred usage across Europe, Bosch's CAN protocol [1] finally won widespread acceptance in the US auto industry during the late 1990s. Worldwide usage brings certain advantages with it. Standardization of components drives down manufacturing costs; it also reduces maintenance costs when replacements are easy to obtain. In addition, auto mechanics and repair personnel only have to learn one electronics communication protocol to diagnose and repair faulty systems. In this entire situation, the only weak link is the educational system.

Many schools around the country teach a course in embedded design, but very few [2,3,4] focus on the networking of embedded systems. With the globalization of our workplace, and jobs moving to other countries, it is important to train US workers on these new technologies to remain competitive with the rest of the world.

The main issue with teaching a course in embedded systems networking is that new software and hardware tools are necessary. Vector [5] and Dearborn Group [6] have developed software tools that can be used to build and simulate a whole CAN network. Although these software packages are very complex and time consuming for the students to learn on their own, they can be taught (delivered) to the students in a simple way. This can be done by focusing on the major components that are considered the base to build any real-time embedded systems network. Vector [5] has developed a software package called CANoe which can be used to do real-time simulations of a large CAN-based. This software can also be used to develop and test CAN networks. Real external nodes can also be connected to the CANoe software system to monitor bus traffic and control external nodes. The system can run a system with both simulated and real nodes working together. Both Vector and Dearborn Group offer short courses to teach their software systems. However, these short courses are very expensive especially for undergraduate students. Thus, we have decided to introduce CANoe software in our undergraduate senior design course so that the students can gain hands-on experience of real-world systems. As a result, the students can be well prepared before they enter the workforce. In this paper, we present an instructional material for CANoe software. The instructional material covers the main components that are needed to build CANoe application in five labs. These labs represent the major phases in the CANoe application development cycle. Students, after learning this material, can move forward to develop more sophisticated applications on their own.

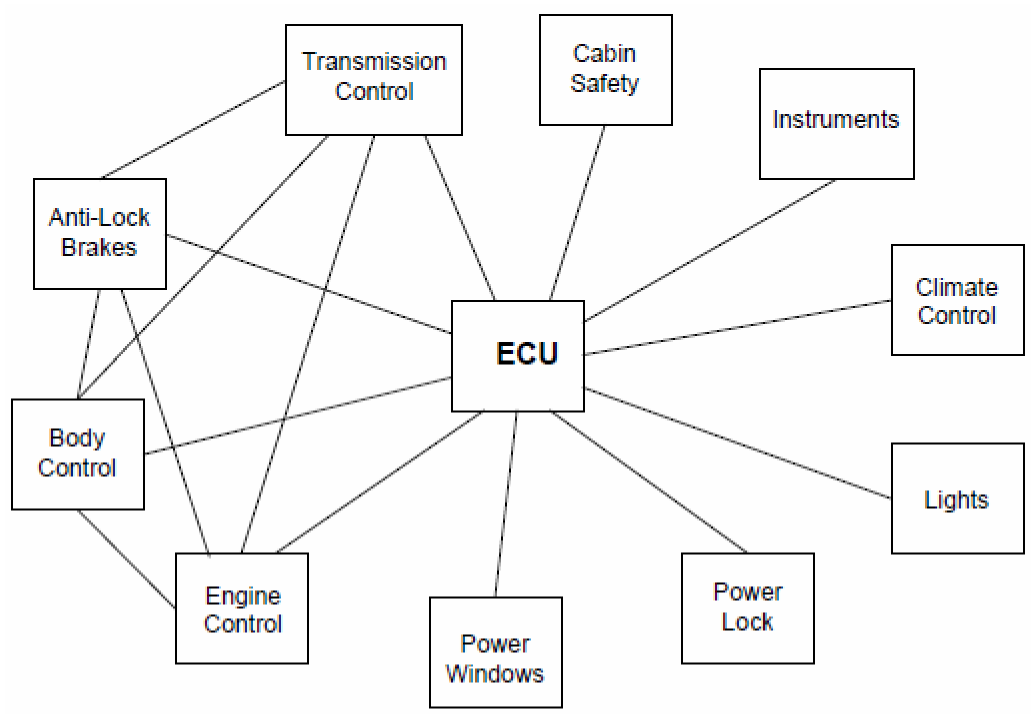


Figure 1: Simplified control network for average automobile without CAN.

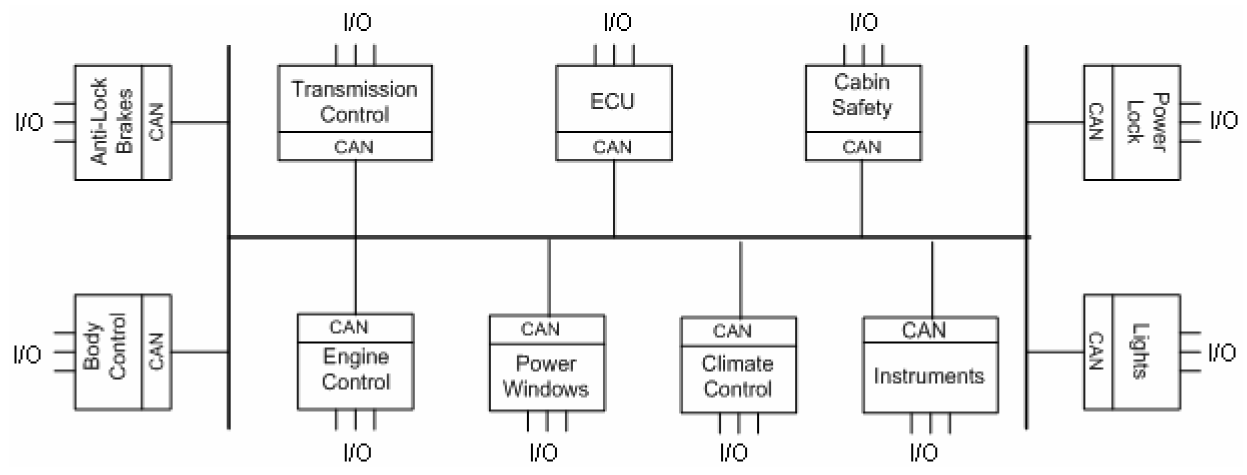


Figure 2: Simplified control network for an average automobile using CAN.

Background

Today, CAN is the standard for mission critical real-time electronic control networks. Developed in the 1980s by Robert Bosch GmbH, with first silicon from Intel Corporation, CAN was created to meet a specific need in the new fleet of automobiles on the road. Microprocessors started to become economical enough and small enough to be implemented in cars. Fuel was now being electronically injected, sensor data was being collected for temperatures, fluid levels, and emissions, and we still had all of the wiring from power windows, doors, and other systems. In this scenario, just as shown in Figure 1, the Electronic Control Unit (ECU) in the car would need to be directly connected to all devices it would interface with. This would include analog or digital connections to every sensor it would be processing. With dedicated links between every processing control center and their external devices, massive amounts of wiring were needed in addition to ever growing wiring harness sizes to accommodate these wires. This not only increases the cost of the vehicle, but adds a noticeable amount of weight to the vehicle.

CAN eschews this structure and replaces it with a serial communication bus between all devices in the vehicle. For the cost of adding a CAN processing chip on each node in the network we are gaining many advantages. We now have to send only one set of wires (most likely two) throughout the vehicle and every device can tap into the same bus. As can be seen in Figure 2, instead of needing numerous analog and digital ports on the ECU, we can now reduce it down to one CAN connection. This severely reduces the number of wires in the car, and the number of inputs needed on the ECU, but it can also increase the intelligence of each node on the network. Every node is connected to the same bus, and therefore, every node sees all of messages sent on the bus. If multiple nodes need a specific sensor’s data, they can pull that data out simultaneously from the same sensor when the data travels over the bus. This is a large improvement versus having to make multiple requests to the ECU for the sensor data, or having to build in local redundant sensors for each node if the ECU cannot handle the traffic. Another great feature is expandability. If the vehicle manufacturer wanted to add a new system to the car, the electronics would have to be placed, the wiring laid out, the harnesses changed to accept the new wiring, and the ECU changed to one that had enough ports. In a CAN network, they would only need to add the electronics and tap into the CAN bus. All other changes would be made to the software firmware of the ECU so that the unit knew the new system's CAN messages and what to do with them. From a protocol level, CAN is already a mature standard [7, 8]. CAN became an ISO standard in the early 1990s, and has seen 20 years worth of testing and applications so far. The protocol was implemented in silicon very soon after introduction, so the CAN node controllers in use can operate at very high speed, and low power consumption with the error handling and fault confinement capabilities handled automatically. The message format and communication methods are also designed with error detection in mind. The robust format allows easy detection of invalid message strings, and quick and easy notification of the transmitter that there was a problem. The CAN protocol also has very good handling of faulty nodes. Instead of a faulty node being able to consume all bandwidth on the network and block all transmissions, the CAN protocol has features built in that allow the network to disable that faulty node. Since we are not 100% reliant on a single node in the network, any transactions that do not involve the faulty node can continue as if nothing bad happened. Even though it was originally applied to automobiles, the dependability and advantages of CAN have attracted other industries to utilize the protocol. The textiles industry was the first to make a major move toward redesigning all of their machines to utilize CAN networks. CAN based weaving, knitting, and sewing machines are in widespread use today. CAN networks can also be found in train systems, airplanes, hospital room controls, and even large and small home appliances [9]. Learning this protocol for embedded networking will help prepare students for a large number of possible industries.

On the development side of CAN, Vector and Dearborn Group both make tools to aid in design and production. Vector's CANoe [10] is a simulation and measurement software that allows us to simulate in-vehicle networks. It empowers us by allowing us to: 1) Simulate nodes in the network, 2) Choose network protocols, 3) Create messages and signals, 4) Monitor simulated and real bus, 5) Log all information required. Using CANoe, various simulation conditions can be created, for example, you can choose to work on real buses or simulated buses (up to 32 buses can be simulated). Network can have simulated and real nodes at the same time. Also, CANoe can be configured to function as a gateway between two networks with different bandwidth e.g., High Speed and Low Speed CAN networks.

Although CANoe software is very complex, it can be presented and delivered to the students in a simplified way. The methodology used is to introduce the basic components at the first stage, and then present the major functionality components that are needed to develop and simulate simple network scenario. For this purpose, we developed an instructional material that is composed of five labs. The material is organized such that the students learn the basic part at early stages and in each new lab; the students advance their knowledge and become familiar with more details of CANoe. At the end, students will be able to simulate simple network scenario.

Instructional Material

The purpose of the CANoe software is to provide us with a development environment for CAN systems. CANoe system also requires us to write codes for real-time networking applications using a high-level language called CAPL which is similar to C language. The instructional material on CANoe has been developed for our senior design class. Since all students come to the senior design class with background in C language, writing codes in CAPL was not an issue.

Since the use of CANoe software requires knowledge of the CAN protocol, lecture notes were developed on CAN protocol to provide the students with the basic background in CAN protocol. The other instructional materials were developed to help students build a standard CANoe model by focusing only on the software part of the CANoe setup. The material is partitioned into five labs. A high-level representation of the labs is shown in Figure 3.

At the end of each Lab, students learn how to operate CANoe to accomplish the goals of the current lab. The labs are organized in an incremental way of complexity. The experiments build upon the previous to add more functionality. In each new stage, students learn new features that enhance their knowledge about the system. Eventually, when students finish all labs, they will be able to simulate a CAN network consisting of multiple nodes.

Learning Outcomes of the Laboratory

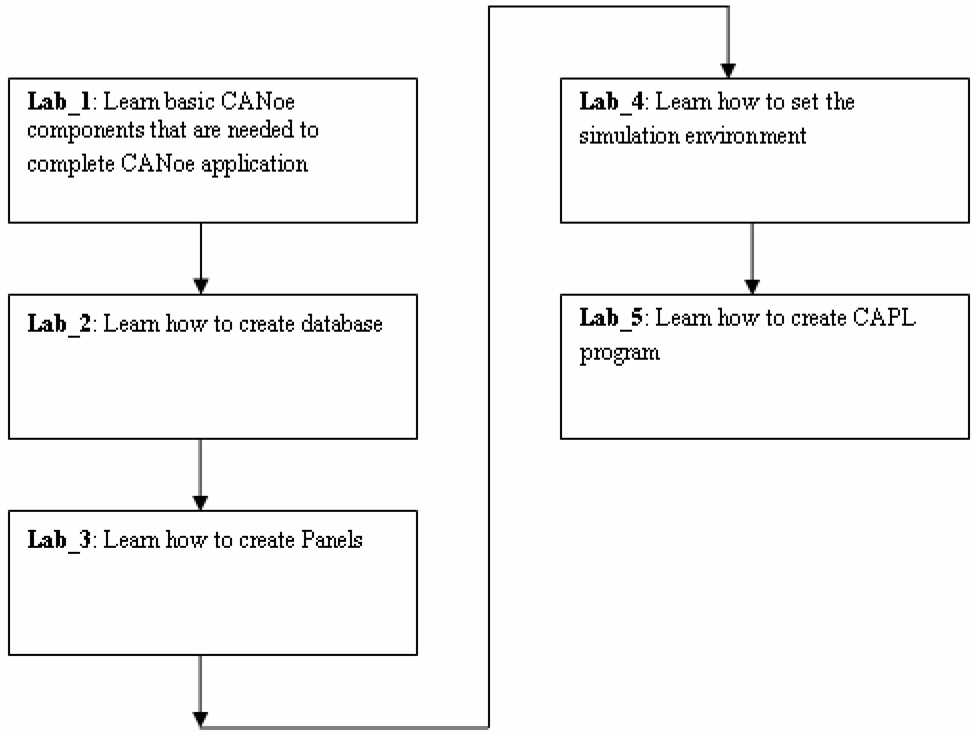

Lab_1: Learn basic CANoe components that are needed to complete CANoe application
Lab_2: Learn how to create database
Lab_3: Learn how to create Panels
Lab_4: Learn how to set the simulation environment
Lab_5: Learn how to create CAPL program

Figure 3: High level description of the labs and their order

After completing the laboratory assignments, students should be able to:

1. Construct CANoe applications.
2. Create database to store all objects needed to complete a CANoe application.
3. Create panels and identify different types of controls that can be placed on the panels.
4. Configure nodes to use environment variables to pass data of external events to the network.
5. Configure the simulation environment and add nodes to the network.
6. Write code using CAPL to implement various types of events.

7. Use CAPL to simulate node behavior.
8. Program nodes to send and receive messages.

Brief Descriptions of Labs

So far we have developed instructional materials for five labs. At the beginning of each lab, students learn the material (content) of that lab, and at the end, they implement the lessons learned from that lab. In order to help students grasp the important lessons of the lab, we developed an assignment for practicing purposes. Students must implement the assignment in parallel with the labs. The solution of the assignment can be done in several phases. Students, in each lab, finish the phase of the solution that is related to the material of that lab. The assignment along with the solution is included in the Appendix. You can see that, each phase of the solution is named after the lab that is used to accomplish that part (phase). Brief descriptions of the five labs are presented below. All these labs together will help the students accomplish the above learning objectives.

Lab1: Exploring Main Functions of CANoe.

The goals of Lab 1 are: to introduce the CANoe Software and its graphical interface, to understand the structure of CANoe, to become familiar with CANoe major components needed to implement CANoe application at the introductory stage, to understand the sequence of developing CANoe applications, to be able to open, load, and run a demo configuration. Since CANoe is complex and huge, we begin by exploring how CANoe looks like, what do various windows have to offer us, and how can we open a new project / configuration in CANoe. At the beginning of this lab, we show students how to start CANoe and then we go over the windows that help students shorten the time to learn whatever they need at this stage. The following windows along with their functionalities have been presented to the students:

1. Windows offering the most generic functionalities, e.g., open and load a configuration file.
2. The database window that is used to create and modify databases.
3. The measurement setup window that is used to configure data statistics we need to monitor.
4. The Simulation setup window, which provides us with an environment to choose the type of the bus, to add and configure nodes for simulation, to associate messages to the configuration setup, etc.
5. A set of windows that can be used to view the data in different formats and display system messages. These windows are:
    a. Trace window: Used to show the content of the message and some other related information like message ID, time stamp, and etc.
    b. Data window: In this window, users can select what data items of the message to be shown.
    c. Graphics window: This is used to show the content of the messages graphically.
    d. Bus Statistics window: This is to show the statistical information for the network, e.g., the number of messages exchanged, the rate of the messages, and so on.

    e. Write window: This is used to output text data during program execution, output the starting and end time of program execution.

6. Start window where we can start, stop, break and step through the simulation.
7. The mode window where we can switch between the online and offline modes of operations.
8. The Panel editor work place which provides us with an environment to create panels that can serve as a front end of the program.

This lab basically gives the directions to the students to focus on the most important components of CANoe that are needed to develop simple CANoe application at this stage. Instead of start learning on their own, students can save time and focus their efforts on learning these components. After that, students can start focusing on more advanced features of the software.

Lab2: Introduction to CANdb++

The CAN system is basically a set of nodes that exchange messages and react upon their reception. Therefore, to make the development of CANoe application simple, all objects e.g., nodes, messages, signals, etc, can be created, configured, and stored in a database so that they can be used at any time during system development. Lab 2 has been designed to introduce the CANdb++ and its graphical editor where objects needed for system development can be created and stored. Not only this, but, students will also be familiar with the characteristics of the objects being used. Therefore, we start the lab by describing the following objects:

- Network Nodes, each network node consists of one CAN controller and a transceiver in an ECU.
- Messages, each message is a container holding a block of data transmitted onto the bus and has a unique ID.
- Signals, which represent the actual data objects to be exchanged among nodes, and are encoded in the data field of the CAN message.
- Value tables, which can be used to define symbolic values.
- Environment variables that are used to describe external events and can be used as input and output variables.

After that and by using the CANdb++, students will be performing the following steps to build the database:

- Create and configure (define) network nodes.
- Create and configure (define) messages.
- Create signals and position them inside the message.
- Represent signal values symbolically.
- Establish object association:
    - Signals to messages association.
    - Messages to network nodes association.
- Define environment variables.

At the end of this lab, students will be able to create their own databases by defining and adding objects from the object tree. They learn how to create network nodes, how to create a complete CAN message, how to define message signals and how to position signals within a CAN message, and how to associate messages to the network nodes. The “Constructing Database” in the appendix shows how students can define the database; how they can define three network nodes, three messages, four signals, and three environment variables, and finally how to assign signals to messages and messages to network nodes. This lab also helps students understand the object hierarchy of the CANdb++ and how objects are organized as shown in Figure 4.

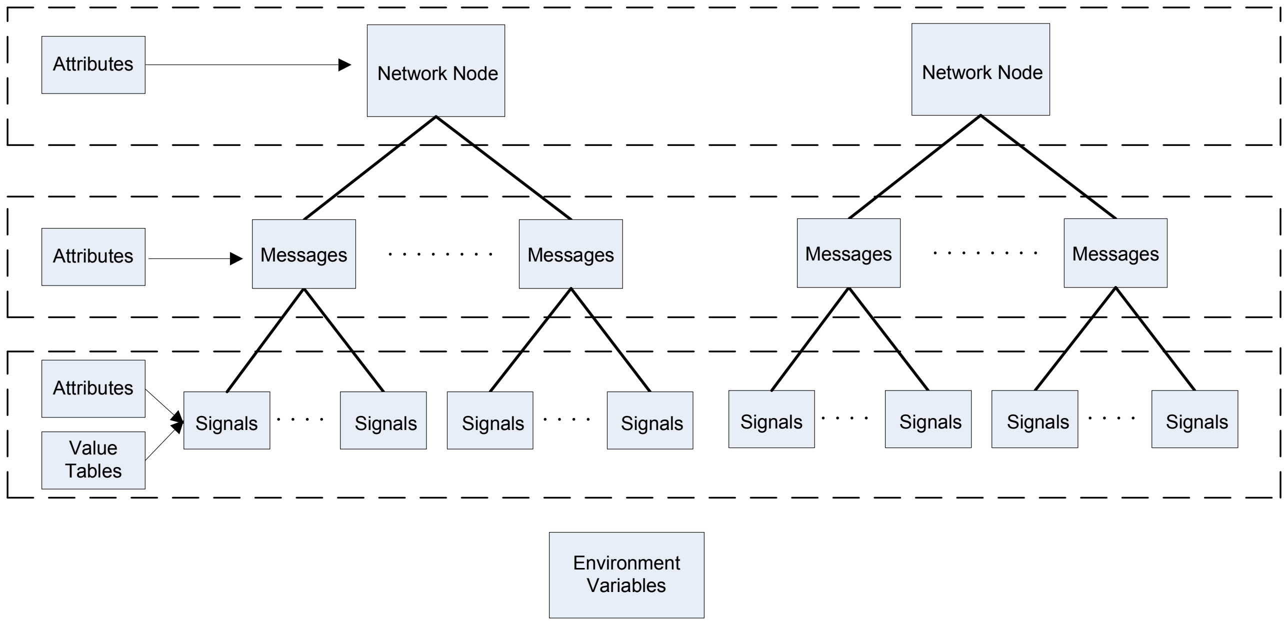


Figure 4: Object hierarchy in the database

Lab3: Introduction to Panel Editor

Panels are basically user-defined graphical interfaces used to control and display node behavior. They are created and assigned to nodes to make it easy to run and operate the system. Panles can react to external events like; changing the state of the switch, or internal events like, receiving messages from CAN bus. This lab has been designed to help students be familiar with the Panel editor, which is basically a tool that can be used to create and arrange panels. But, In order for the students to be able to use Panel editor to create and configure panels, they need to understand the different types of controls that can be placed on the panels. These controls are called elements and can be classified into:

- Control elements: they are used to represent the values from sensors. These values can be modified by the user. Push buttons, Switches, are an example of this type.
- Display elements, they are used to represent values from actuators. The changes of these values can be displayed. Meters, Multi-control, and gauges are an example of this type.
- Static elements where no environment variables can be assigned, e.g., Bitmap, Text (Label), and so on.
- Input/Output Control, which regulate the input values resulting from our programmed actions, and display the resulting output values, e.g., Input/Output control.

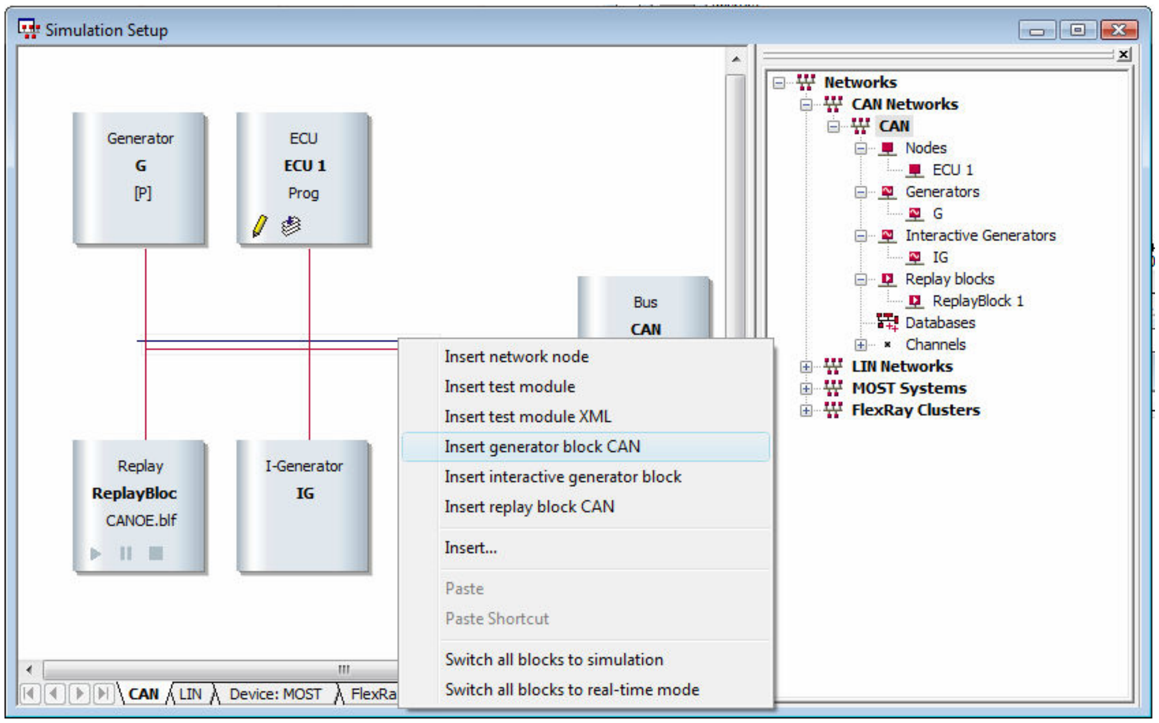
In this lab, students learn how to run the Panel editor, how to choose the predefined panels' elements, how to configure nodes to use environment variables to pass data of external events to the network, and let the receiving nodes display the data embedded in the messages being received. Students will also learn how to configure elements as input, output or both. In addition to that, they will also learn how to associate elements with either an environment variable or signal defined in the database. Finally, students learn how to associate the created panels to the CANoe configuration. The "Creating Panels" in the appendix shows how students can create one Control element (Switch) and two Display elements (LCD). The students then associate the switch to one of the environment variables, and the LCDs to the other environment variables. After the elements (one switch and two LCDs) have been configured, students perform the Panels to CANoe association.

Lab4: CANoe Simulation Setup

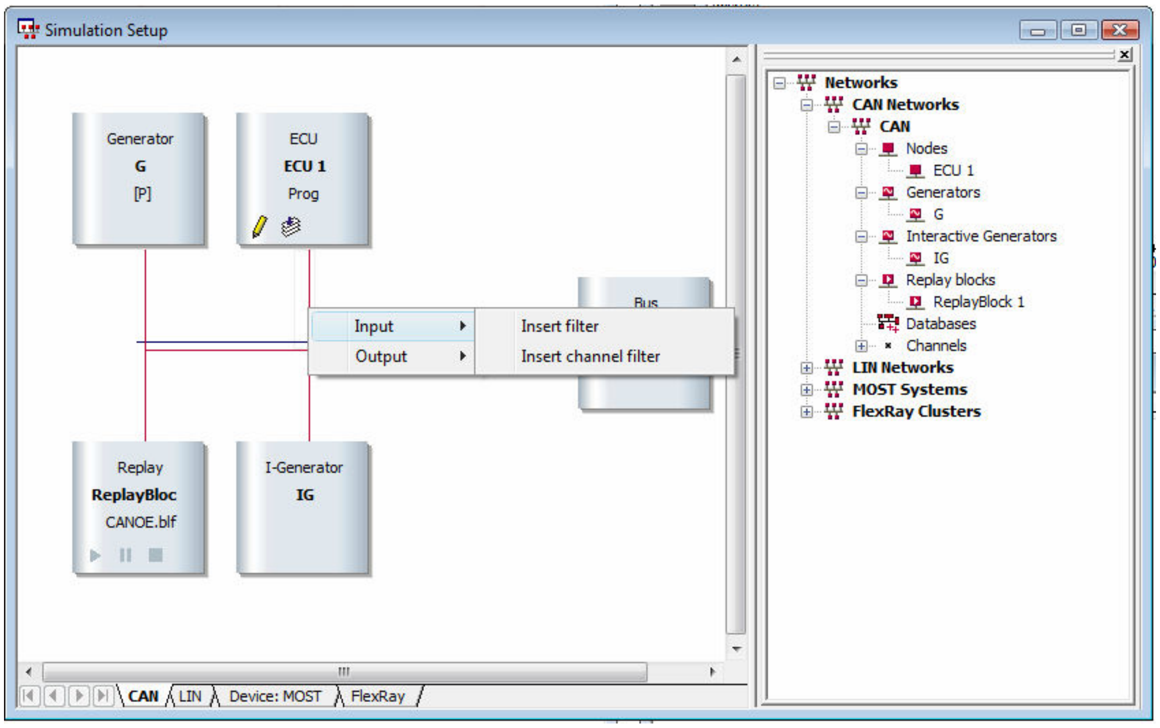
Up to this point, students have already learned how to create databases, and how to define nodes, messages, and signals. The time now is to learn how to add these nodes to the CAN network, and to understand the environment that provides us with the infrastructure where the communication among nodes can take place. This lab is designed to introduce the simulation setup tool where students can visualize the distribution of network nodes on CAN bus. In this lab, students learn how to configure network nodes to communicate with each other. In addition to that, students learn how to associate the database and CAN bus to the network. But, in order for the students to be able to use the simulation setup, they need to understand the various types of nodes that can be added to the network, these types can be summarized as follows:

- Generator Block: The purpose of this type of nodes is to send messages configured in advance and organized in a transmit list based on some triggering conditions.
- Interactive Generator Block: this is similar to the Generator Block, but the difference is that signals, messages, and the triggering conditions can be adjusted during the run time.
- Replay Block: this is used to capture messages from a log file and resend them.
- Network Node: this is the most important type of nodes that can be used to simulate complex nodes
- Test Module: This type is not defined in the network database. It can be implemented in two ways: using CAPL or using an XML file. This type is good for large number of test scenarios generated from existing data.

Once students become familiar with the different types of nodes, they start performing simulation setup. They start by opening the simulation setup window and then associating the database to the network. In the next step, students learn how to add nodes to the network, Figure. 5 (a) shows the process of adding a node to the network. In addition to that, students learn how to add filtering elements by choosing the filter type (Input / Output). The filtering can be done by right clicking on the line connecting the node to the bus and inserting a filter for message, signal, and nodes as shown in Figure. 5 (b). The whole channel can also be filtered. Using this technique, we can block messages and prevent some nodes from receiving or sending them. Students also learn how to deactivate and reactivate simulated nodes and filtering blocks. The "CANoe simulation setup" in the appendix shows how students can open the simulation setup tool add the three Network Nodes defined earlier in the database to the CAN bus.



(a)



(b)

Figure 5: (a) Adding a node to the network. (b) Insert filtering block

Lab 5: Configuring Network Nodes Using CAPL.

In the previous labs, students learn how to create database, create panels, and setup the simulation environment by adding nodes to the network. But, the system can not be operational unless we program the network nodes and simulate their behavior. In this lab, we introduce CAPL that is used to program nodes and simulate their behavior. We try to teach the most important components of CAPL that are sufficient to complete CANoe application at this level.

The organization of this lab is as follows:

1. Presenting the features of CAPL.
2. Configuring nodes by opening CAPL browser and assigning CAPL programs to them.
3. Accessing database objects and interacting with panels.
4. Presenting CAPL events that are needed to develop simple CANoe application.
5. Defining messages and signals, and creating functions in the CAPL browser.

At the beginning, we introduce CAPL, which is based on C language, and present its unique features and the main differences from C as shown in table I. We also show how to construct CAPL program. In this lab, students learn how to open CAPL browser, by double clicking on the Network Node shown in Figure 5, to configure network nodes. Figure 6 shows the CAPL browser along with the events used to simulate nodes' behavior. We show students how to declare global variables in the global variable declaration area, how to access the database to locate messages and signals for declaration. The students also learn how to add, define, and write event procedure, they learn how to define and call function, and finally, they learn how to compile the code.

Since CAPL is an event–based language, Students should learn that CAPL applications are developed to respond to the occurrence of the predefined events. These events are classified into:

- Input/Output events.
- Timing events.
- Communication events.

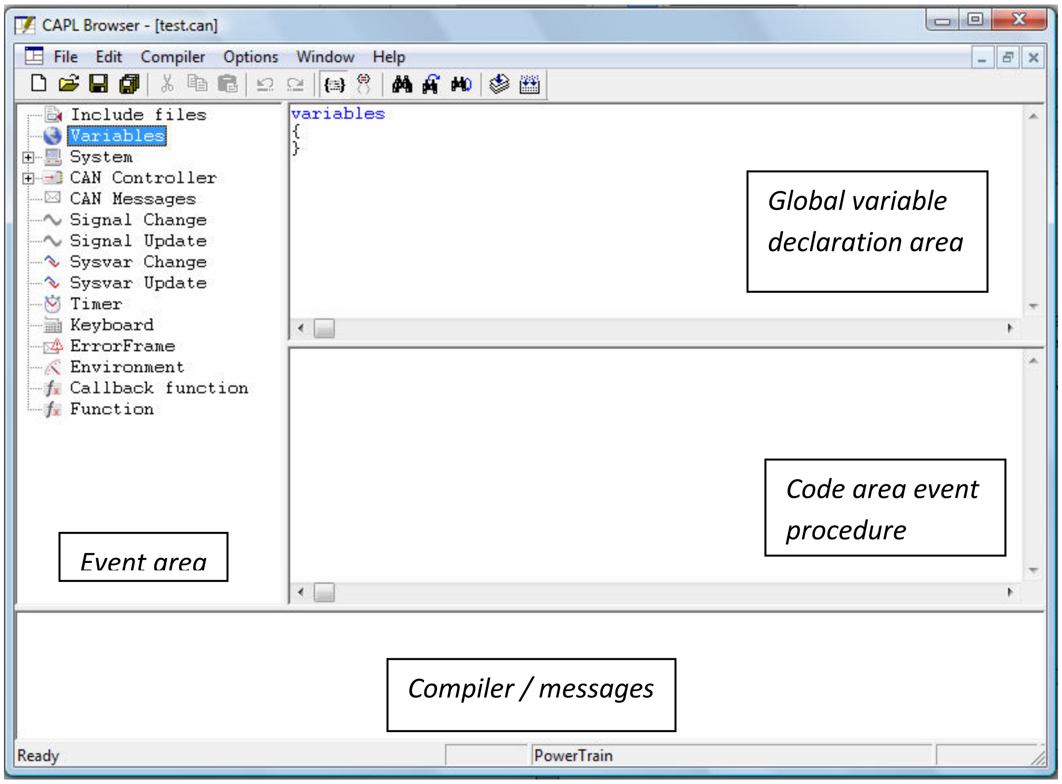
We train students to add and write code for the events. We show them that events are not executed in a sequential order, although the instructions within the event are executed sequentially. Students are able to understand that the code of the event is executed only when it occurs. In this lab, we let students focus on the following events:

1. CAN message: This event is called when a message is received, e.g., whenever a node receives a message with a certain ID it starts a timer to send periodic messages.
2. Keyboard: This event is called and executed after pressing a key, e.g., a message can be sent to the bus when pressing a certain key.
3. Timer: This event is called when a timer expires. Two types of timers are supported, the mstimer that have a millisecond resolution, and the timer having resolution in seconds.
4. Environment: This event is called when an environment variable is changed in the CAPL node or in the Panel.

We also introduce some embedded functions, e.g., the output function that is used to send messages to the bus and the write function to display system messages on the screen. At the end of this lab, students learn how to program nodes to react to the occurrence of some external and internal events.

Table I: Main Features of CAPL

CAPL	
Main Features	**Main differences from C language**
Event- based language (events are uninterruptible)	Doesn't support user defined structures
Interfacing with the database, which provides us with an easy way to access the database objects like Environment variables, Messages, and Signals	Support function overloading
Utilizing the environment variables and signals to interact with panels.	No Pointers are used



```
variables
{
}
```

Figure 6: CAPL browser

The "Programming nodes using CAPL", presented in the appendix, shows how students declare the global variables like the messages, timers, the expiration period of the timer, and other variables. They also define event procedures like environment variable that will be executed after the switch is turned on, the timer event that can be used as an indication to send a message using output function or increment the numbers to be displayed on the LCD. The code for all nodes is included in the appendix. By finishing this phase, the students are able to complete a simple CANoe application that can be considered as a base to start more sophisticated tasks.

Table II: Survey results of the learning outcomes.
(Numerical values in the cells of the following table indicate the number of students.)

Learning Outcomes	No Ability	Some Ability	Adequate Ability	More than Adequate	Ability High Ability
1. Construct CANoe applications.		2	4	10	8
2. Create database to store all objects needed to complete a CANoe application.		3	5	10	6
3. Create panels and identify different types of controls that can be placed on the panels.		3	6	9	6
4. Configure nodes to use environment variables to pass data of external events to the network.		2	5	10	7
5. Configure the simulation environment and add nodes to the network.		1	4	11	8
6. Write code using CAPL to implement various types of events.		4	5	9	6
7. Use CAPL to simulate node behavior.		4	6	8	6
8. Program nodes to send and receive messages.		3	7	8	6

Assessment of Student Learning

Our teaching materials on Embedded Systems Networking were introduced to the students during the Fall-2008 semester in our senior design class. Altogether 10 hours of lecture (five 2-hr lectures) were presented to cover the materials related embedded systems networking. The students were divided into groups of three. The students of each group worked together to achieve the goals of each laboratory assignment. The students were required to present a demo of each assignment. Different student of a group was required to take the lead in showing the demo of different assignment. This way, we could make sure that all students participated equally and learning experience among the entire student population was similar. Most students were able to do all five assignments without any major difficulties. However, a few students who did not have strong programming skills had some difficulty. There were 24 students in the class and they were divided into 8 groups. Six groups completed all labs with 100% accuracy. One group achieved 80% accuracy and the other group achieved 75% accuracy. Thus, we believe that our goals of teaching embedded systems networking materials were met. We also conducted a survey to determine how the learning outcomes of the laboratories were met. The survey results are presented in Table II.

The above survey results indicate that the students gained significant experience in designing embedded system networks. As a result, we believe that our goals have been achieved. We are also using our teaching materials during the Winter-2009 semester. We will also do another assessment for the current semester.

Conclusion

This paper presents the description of the instructional materials that have been developed to teach CANoe software system. Using this system, students quickly learn how to construct and simulate CAN networks. We believe student learning about embedded systems networks is enhanced due to the use of our instructional materials. The lecture notes and related instructional materials can be made available to any interested instructors for teaching embedded networking system at their institutions.

Bibliography

1. Bosch, "CAN Specification Version 2.0," Robert Bosch GmbH, Stuttgart, Germany, 1991.
2. Oklahoma State University – Advanced Embedded Systems Design - http://biosystems.okstate.edu/Home/mstone/5030_04/5030_04index.htm
3. Carnegie Melon University – Embedded Systems Design - http://www.ece.cmu.edu/~ece549/index.html
4. Wayne State University – Capstone Design –
5. http://ece.eng.wayne.edu/~smahmud/ECECourses/ECE4600/ECE4600.htm
6. Vector Group Worldwide – http://www.vector-worldwide.com/
7. Dearborn Group - http://www.dgtech.com/
8. International Standards Organization, "Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signaling," ISO 11898-1, 1993.
9. International Standards Organization, "Road vehicles – Controller area network (CAN) – Part 2: Highspeed medium access unit," ISO 11898-2, 1993.
10. CANopen Application Examples – http://www.canopen.us/applications.htm
11. Vector CANoe – http://www.vector-worldwide.com/vi_canoe_en,,223.html

Appendix

Detailed step by step to construct simple CANoe application

The goals of this practice are as follows:

- To let students apply what they learn in the five labs mentioned earlier.
- To construct a complete CANoe application by passing through labs in the way they are organized.

The assignment:

Step1:

Create a CANoe Database that contains the following three objects:

- Three network nodes (N_1, N_2, N_3)
- Three messages (msg_1, msg_2, msg_3)
- Six signals (sig_1, sig_2, sig_3, sig_4).

Assign sig_1 and sig_2 to msg_1, sig_3 msg_2, and sig_4 to msg_3. Then associate msg_1 to N_1, msg_2 to N_2, and msg_3 to N_3.

Step2:

Use environment variables and panel editor. You need to use the Switch button and the LCD (1 switch button and 2 LCDs).

Step3:

When you change the state of the switch to "ON" (by pressing on it), the following procedure starts and continues running as long as the state of the switch remains "ON".

1. N_1 generates 2 random numbers. The first Random Number (RN1) should be between 6 and 20, while the second Random Number (RN2) is either 1 or 2. The node stores RN1 and RN2 in Sig_1 and Sig_2 respectively. The node must wait for 1 second and then send the message to the can bus. The first generated number represents the maximum value of a counter, and the second number represents the increment step.
2. Nodes (N_2 & N_3) receive the message and extract its contents. Based on the content of the message, the two nodes take the following actions:
   - If RN1 is between 6 and 13 ($6 \leq RN1 \leq 13$), only N_2 reacts as follows:
      i. N_2 starts counter and shows the counting process on the LCD. The maximum counter value is defined by the received Sig_1, while the counting steps (each increment step) are defined by Sig_2.
      ii. If RN1 is odd and RN2 is even, then the maximum count can be up to RN1-1.
      iii. When the counter reaches its maximum value. N_2 waits 1500 msec. and replies by sending a message to the bus. The message contains the generated random number (RN1) incremented by 1.

iv. Only N_1 is allowed to respond to this message. N_1 generates new random numbers, stores them in Sig_1 and Sig_2 respectively, and sends its message to the bus. If the new generated random number remains between 6 and 13 ($6 \le RN \le 13$), we need to repeat the previous step (step i).
v. This process will be repeated as long as the state of the switch is **ON**.

- If RN1 is between 14 and 20 ($14 \le RN1 \le 20$), only N_3 reacts as follows:
  i. N_3 starts counter and shows the counting process on the LCD. The maximum counter value is defined by the received Sig_1, while the counting steps (each increment step) are defined by Sig_2.
  ii. If RN1 is odd and RN2 is even, then the maximum count can be up to RN1-1.
  iii. When the counter reaches its maximum value. N_3 waits 1500 msec. and replies by sending a message to the bus. The message contains the generated random number (RN1) incremented by 1.
  iv. Only N_1 is allowed to respond to this message. N_1 generates new random numbers, stores them in Sig_1 and Sig_2 respectively, and sends its message to the bus. If the new generated random number remains between 14 and 20 ($14 \le RN \le 20$), you need to repeat the previous step (step vi).
  v. This process will be repeated as long as the state of the switch is **ON**.

<u>Step4:</u>

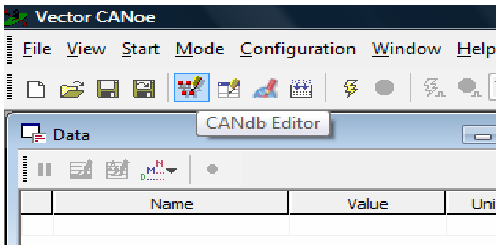
To exit from the above procedure, you need to press on the switch again to change its state to "**OFF**".

<u>Solution:</u>

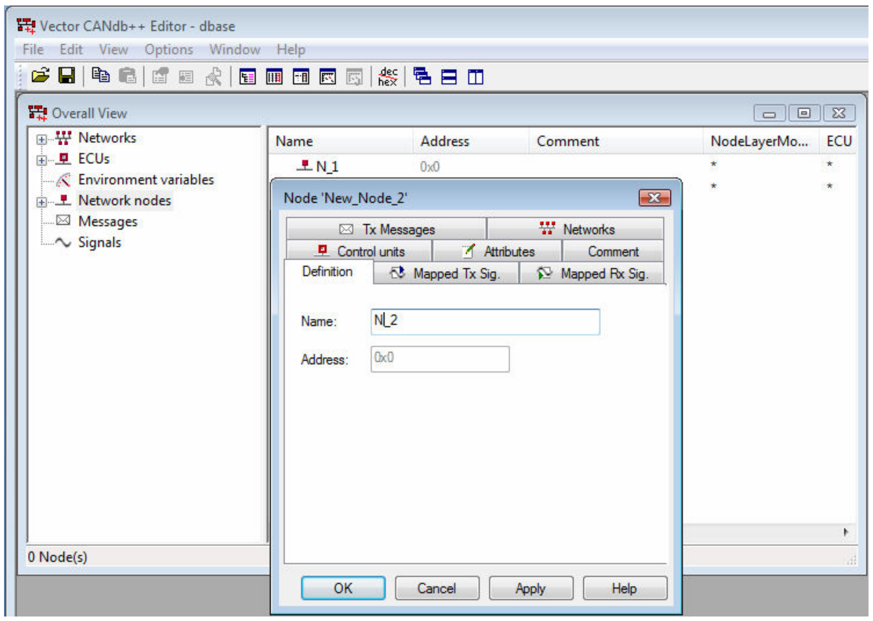
<u>Constructing Database:</u>

The first thing that needs to be done is creating the objects (Nodes, messages, signals, and environment variables), assigning signals to messages, and associating the database to the configuration. In the following steps we show how to accomplish this.

1- Click on the database icon to launch CANdb++ in CANoe as shown below:



2- Add nodes as show below

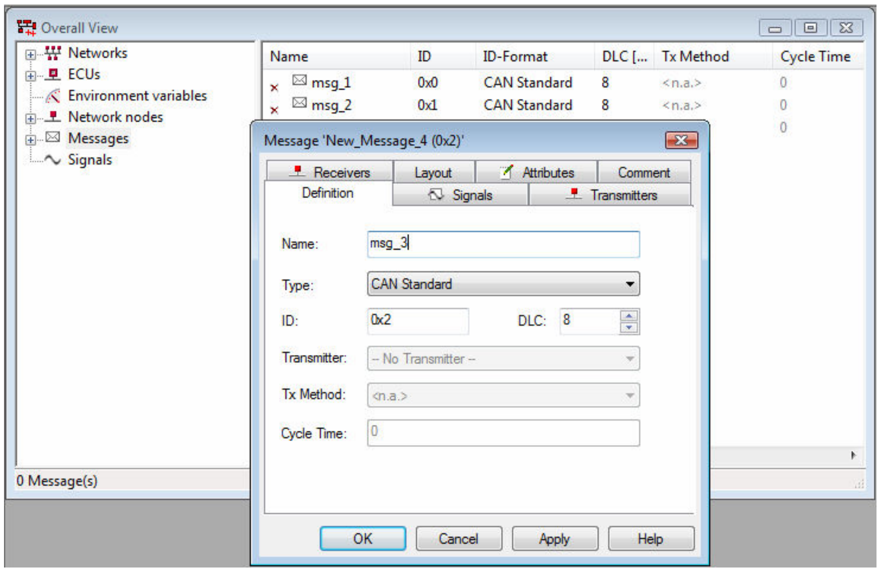


Name	Address	Comment	NodeLayerMo...	ECU
N_1	0x0		*	*
			*	*

Name: N_2

Address: 0x0

3- The following figure shows how to add messages. Signals and Environment variables can be add in the same way



Name	ID	ID-Format	DLC [...	Tx Method	Cycle Time
msg_1	0x0	CAN Standard	8	<n.a.>	0
msg_2	0x1	CAN Standard	8	<n.a.>	0
					0

Name: msg_3

Type: CAN Standard

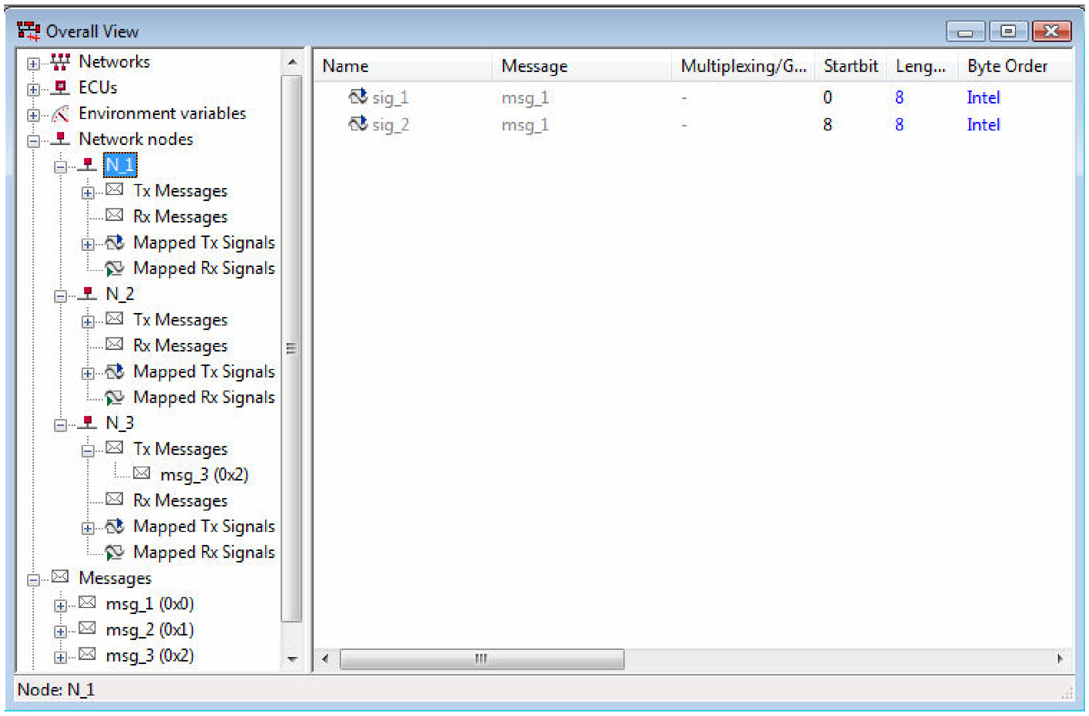
ID: 0x2 DLC: 8

Transmitter: -- No Transmitter --

Tx Method: <n.a.>

Cycle Time: 0

4- Assign signals to messages and messages to nodes. The figure below shows all objects after the association process:



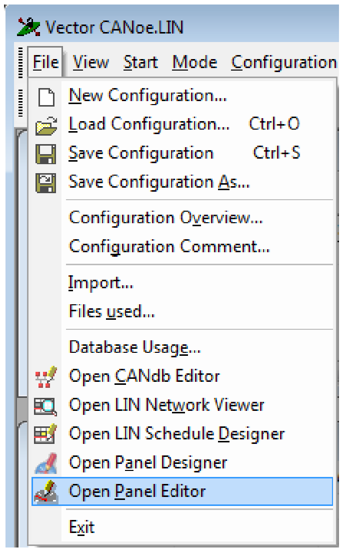
Name	Message	Multiplexing/G...	Startbit	Leng...	Byte Order
sig_1	msg_1	-	0	8	Intel
sig_2	msg_1	-	8	8	Intel

5- Associate the database to the configuration.

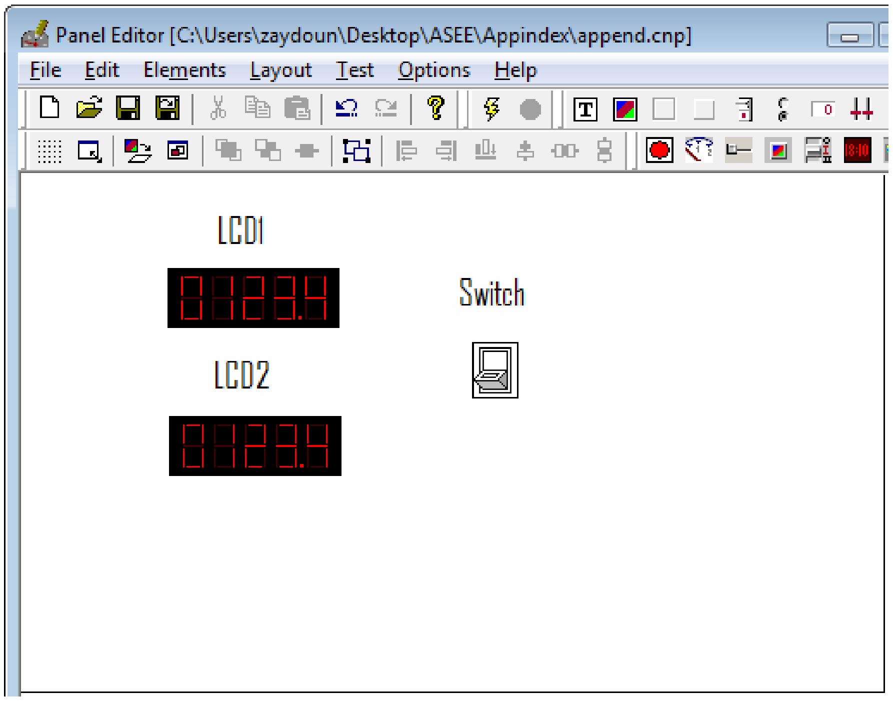
Creating Panels

Since the assignment requires us to use switch to initiate the communication process and to use the LCD to show the content of the messages, we start the second step by creating panels, configuring them to react to the external event (changing the state of the switch) and internal event (receiving messages), and associate the panels to the configuration file. In the following steps, we show how to create and configure panels.

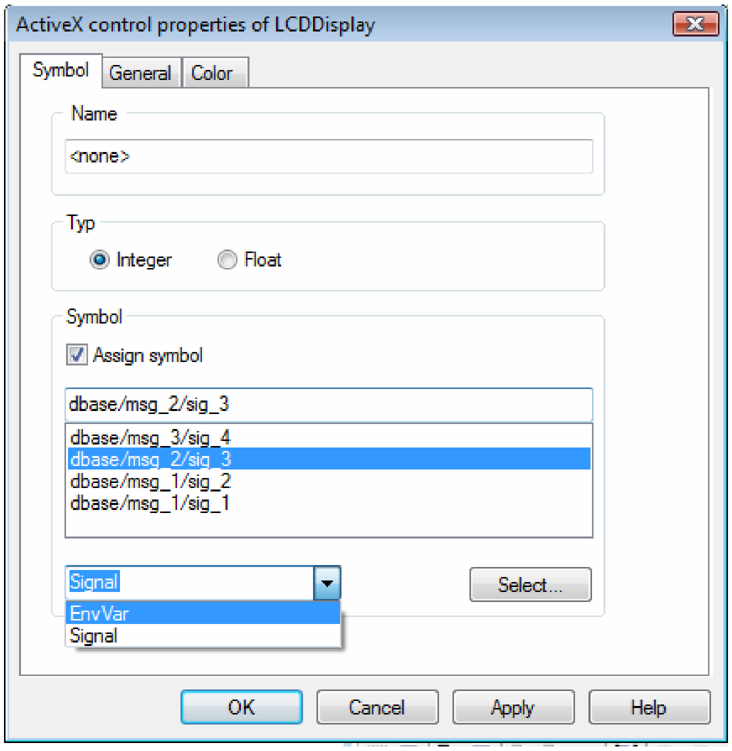
1- Open Panel editor as shown below



2- Select input and output elements and add them to the panel as shown below:


Panel Editor [C:\Users\zaydoun\Desktop\ASEE\Appindex\append.cnp]
File Edit Elements Layout Test Options Help
LCD1
LCD2
Switch

3- Configure these elements by right click on the element and select configure, then associate the element either with the signal or environment variable. The figure below shows how to configure the LCD



Name

<none>

Typ

[✓ Integer] Float

Symbol

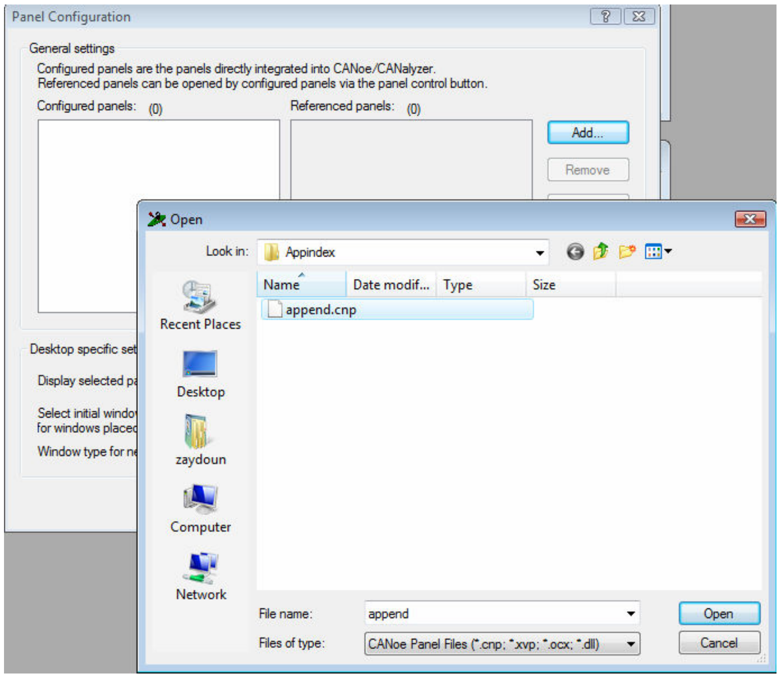
[✓] Assign symbol

dbase/msg_2/sig_3

dbase/msg_3/sig_4
dbase/msg_2/sig_3
dbase/msg_1/sig_2
dbase/msg_1/sig_1

Signal
EnvVar
Signal

4- After configuring all elements, we need to associate the panels to the configuration. This can be done by selecting the Panel configuration under configuration in the main menu. The figure below shows how associate panels.


Panel Configuration

General settings

Configured panels are the panels directly integrated into CANoe/CANalyzer.
Referenced panels can be opened by configured panels via the panel control button.

Configured panels: (0)

Referenced panels: (0)

Open

Look in: Appindex

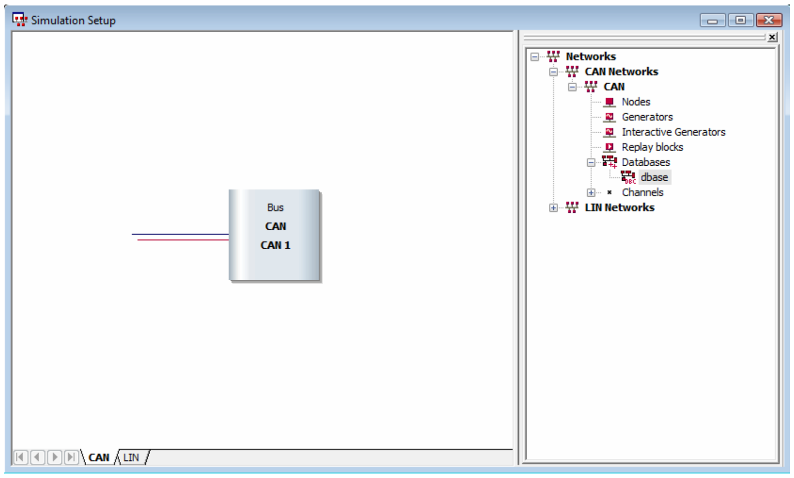
File name: append

Files of type: CANoe Panel Files (*.cnp; *.xvp; *.ocx; *.dll)

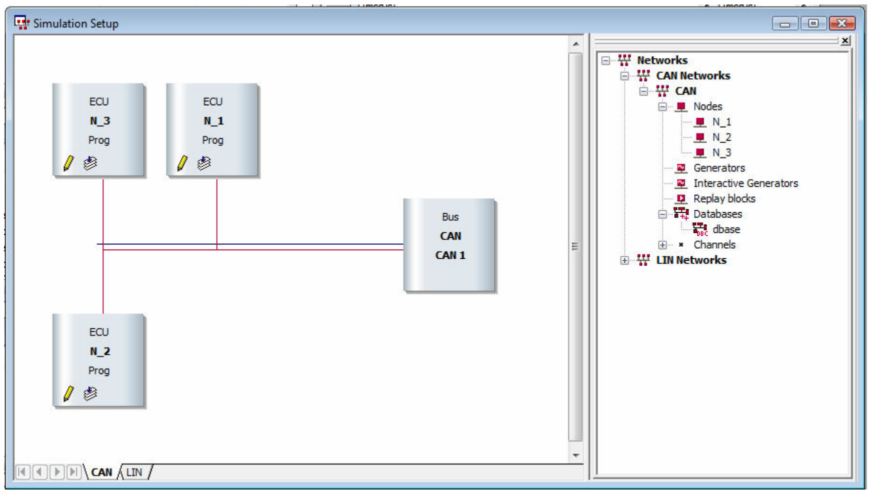
CANoe Simulation Setup

Before writing codes to simulate nodes' behavior, we need to build the network that provides us with the infrastructure where nodes can exchange information. Therefore we start this phase by adding the nodes in the database to the CAN bus. In the following steps, we show how to accomplish this.

1- Open the simulation setup window under view in the main menu. The figure below shows the CAN bus without nodes being connected to it.



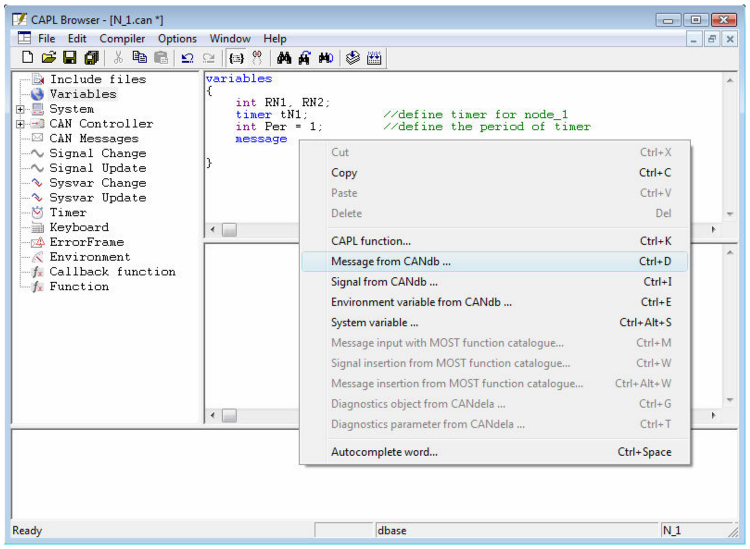
2- Add the three nodes to the bus. There are more than one way to add these nods, we choose to the nodes by right click on the database, select node synchronization, add nodes one by one to the network, and click finish to complete the process. The figure below shows all three nodes being added to the network.



Programming nodes using CAPL

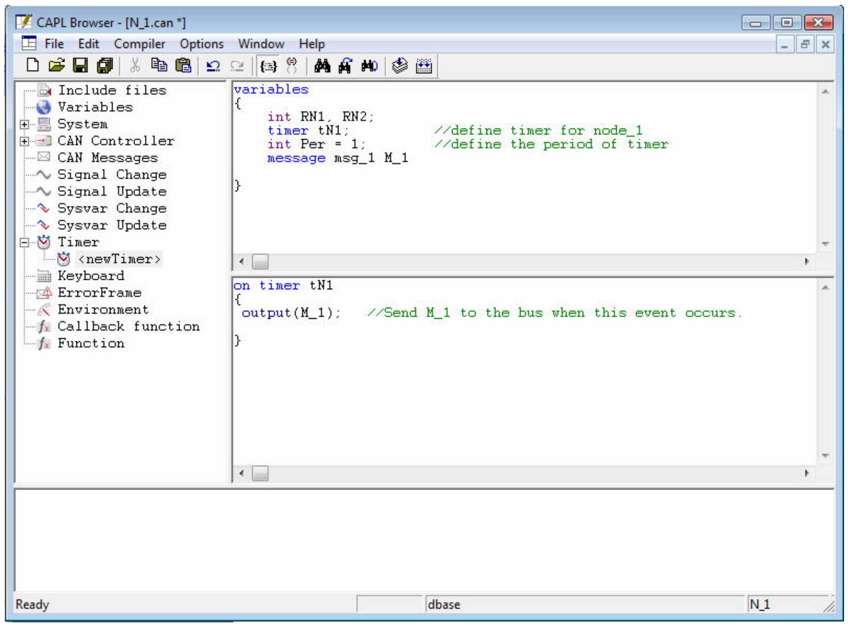
The last phase to complete CANoe application is to program nodes using CAPL. In the following steps, we show how to open CAPL browser, how to add events and write procedures to describe them, and how to make nodes react upon the occurrence of these events.

1- A single click on the pencil icon on the node opens the CAPL browser. We start by declaring the global variables. The figure below shows global variable declaration for Node_1. We also show how to access the database from CAPL to locate messages for declaration. To do this, just right click on the blank space after the keyword message, and then select "Message from CANdb".


CAPL Browser - [N_1.can *]
File Edit Compiler Options Window Help
Include files
Variables
System
CAN Controller
CAN Messages
Signal Change
Signal Update
Sysvar Change
Sysvar Update
Timer
Keyboard
ErrorFrame
Environment
Callback function
Function
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Del
CAPL function... Ctrl+K
Message from CANdb ... Ctrl+D
Signal from CANdb ... Ctrl+I
Environment variable from CANdb ... Ctrl+E
System variable ... Ctrl+Alt+S
Message input with MOST function catalogue... Ctrl+M
Signal insertion from MOST function catalogue... Ctrl+W
Message insertion from MOST function catalogue... Ctrl+Alt+W
Diagnostics object from CANdela ... Ctrl+G
Diagnostics parameter from CANdela ... Ctrl+T
Autocomplete word... Ctrl+Space
Ready
dbase
N_1

```
variables
{
    int RN1, RN2;
    timer tN1;          //define timer for node_1
    int Per = 1;        //define the period of timer
    message
}
```

2- To add events, select the particular event from the event tree, right click on it and select "New". In the figure below, we show the procedure of the "Timer" event that will be executed whenever the timer of Node_1expires.



```
variables
{
    int RN1, RN2;
    timer tN1;          //define timer for node_1
    int Per = 1;        //define the period of timer
    message msg_1 M_1
}
```

```
on timer tN1
{
 output(M_1);   //Send M_1 to the bus when this event occurs.
}
```

3- Similar to 2, we can add events and then define them. All events along with the code to simulate nodes' behavior is:

<u>Node_1</u>

```
                              /*Global variable declaration*/
variables
{
   int RN1, RN2;
   timer tN1;          //define timer for node_1
   int Per = 1;        //define the period of timer
   message msg_1 M_1;
}

  /* The procedure that will be executed whenever Node_1 receives a message from Node_2*/
on message msg_2
{
   RN1 = random(14) + 6;                  //generate random number between 6 and 20
   RN2 = random(2) + 1;                   //generate a number that is either 1 or 2
   M_1.sig_1 = RN1;                       //Store RN1 in signal_1 of msg_1
   M_1.sig_2 = RN2;                       //Store RN2 in signal_2 in msg_1
   write("RN1 = %d", M_1.sig_1);          //Display RN1 on write window
   write("RN2 = %d", M_1.sig_2);          //Display RN2 on write window
   output(M_1);  //Send the message to the bus
}

   /* The procedure that will be executed whenever Node_1 receives a message from Node_3*/
on message msg_3
{

   RN1 = random(14) + 6;              //generate random number between 6 and 20
   RN2 = random(2) + 1;               //generate a number that is either 1 or 2
   M_1.sig_1 = RN1;                   //Store RN1 in signal_1 of msg_1
   M_1.sig_2 = RN2;                   //Store RN2 in signal_2 in msg_1
   write("RN1 = %d", M_1.sig_1);      //Display RN1 on write window
   write("RN2 = %d", M_1.sig_2);      //Display RN2 on write window
   output(M_1);                       //Send the message to the bus
}

                /*The procedure that will be executed whenever "Timer tN1" expires*/
on timer tN1
{
   output(M_1);                        //Send M_1 to the bus when this event occurs.
}

            /*The procedure that will be executed when the state of the switch is changed*/
on envVar En_Var_1
{
 //Check the state of the Switch, and take the appropriate action
 switch(@En_Var_1)
   {
    case 0:                            //The switch is OFF
      write("Switch os OFF");
      canceltimer(tN1);
      break;

    case 1:                            //The switch is ON
      write("Switch is ON");
      RN1 = random(14) + 6;            //generate random number between 6 and 20
      RN2 = random(2) + 1;             //generate a number that is either 1 or 2
      M_1.sig_1 = RN1;                 //Store RN1 in signal_1 of msg_1
      M_1.sig_2 = RN2;                 //Store RN2 in signal_2 in msg_1
      write("RN1 = %d", M_1.sig_1);    //Display RN1 on write window
      write("RN2 = %d", M_1.sig_2);    //Display RN2 on write window
      settimer(tN1, Per);              //Set timer
      break;
    }
}
```

Node_2

```
                              /*Global variable declaration*/
variables
{
   int vLCD;
   int RCVd_RN1, RCVd_RN2, RN1_count;
   message msg_2 M_2;
   mstimer tN2;                    //this timer is used to introduce delay of 1500msec
   int ms_per = 1500;              // the period of tN1
   timer LCDcnt;                   //this timer is used for incrementing the number that
                                   //will be displayed every second
   int per = 1;                    //the period of the LCDcnt
}

   /* The procedure that will be executed whenever Node_2 receives a message from Node_1*/
on message msg_1                   //upon the receiption of message 1
{
   RCVd_RN1 = this.sig_1;
   RCVd_RN2 = this.sig_2;

// if RCVd_RN1 is odd and RCVd_N2 is even, then decrement RCVd by 1
// and store it in RN1_count, RN1_count will be the target to count towards

   if(RCVd_RN1 >=6 && RCVd_RN1 <=13)
    {
      if(RCVd_RN1 % 2 !=0 && RCVd_RN2 ==2)
         RN1_count = RCVd_RN1 - 1;
      else
         RN1_count = RCVd_RN1;
      settimer(LCDcnt, per);                  //activate timer so that the increment
                                              //occurs when the timer expires
      vLCD = 0;
      @En_Var_2 = vLCD;                       //initialize the value to be displayed on LCD

    }
}

               /*The procedure that will be executed when Timer LCDcnt expires*/
on timer LCDcnt
{
   vLCD = vLCD + RCVd_RN2;     //increment vLCD by RCVd_RN2 and display it on LCD
   @En_Var_2 = vLCD;

//Continue counting as long as the target value is not reached
   if(vLCD < RN1_count)
      settimer(LCDcnt, per);
//if the target value is reached, then stop counting by deactivating timer LCDcnt
//Also activate timer tN2 which will send message_2 to the bus after 1500 msec.
   else
   {
      canceltimer(LCDcnt);
      settimer(tN2, ms_per);
   }

}

                    /*The procedure that will be executed when Timer tN2 expires*/
on timer tN2
{
   //increment the received number by 1 and save it in the message
   M_2.sig_3 = RCVd_RN1 + 1;
   output(M_2);                          // Send the message to the bus
}
```

Node_3

```
                                        /*Global variable declaration*/
variables
{
   int vLCD3;
   int RCVd_R1, RCVd_R2, R1_count;
   message msg_3 M_3;
   mstimer tN3;                //this timer is used to introduce delay of 1500msec
   int ms_per = 1500;          // the period of tN1
   timer LCDcntr;              //this timer is used for incrementing the number that
                               //will be displayed every second
   int per = 1;
}

   /* The procedure that will be executed whenever Node_3 receives a message from Node_1*/
on message msg_1
{
   RCVd_R1 = this.sig_1;
   RCVd_R2 = this.sig_2;

// if RCVd_RN1 is odd and RCVd_N2 is even, then decrement RCVd by 1 and store
// it in RN1_count, RN1_count will be the target to count towards

   if(RCVd_R1 >=14 && RCVd_R1 <=20)
   {
      if(RCVd_R1 % 2 !=0 && RCVd_R2 ==2)
```

```
          R1_count = RCVd_R1 - 1;
       else
          R1_count = RCVd_R1;
       settimer(LCDcntr, per);             //activate timer so that the increment
                                           //occurs when the timer expires
       vLCD3 = 0;
       @En_Var_3 = vLCD3;                  //initialize the value to be displayed on LCD

   }
}

                /*The procedure that will be executed when Timer LCDcntr expires*/
on timer LCDcntr
{
 vLCD3 = vLCD3 + RCVd_R2;        //increment vLCD by RCVd_RN2 and display it on LCD
   @En_Var_3 = vLCD3;

//Continue counting as long as the target value is not reached
   if(vLCD3 < R1_count)
       settimer(LCDcntr, per);

//if the target value is reached, then stop counting by deactivating timer LCDcnt
//Also activate timer tN2 which will send message_2 to the bus after 1500 msec.
   else
   {
       canceltimer(LCDcntr);
       settimer(tN3, ms_per);
   }
}

                   /*The procedure that will be executed when Timer tN3 expires*/
on timer tN3
{
   M_3.sig_4 = RCVd_R1 + 1;   //increment the received number by 1 and save it in the message
   output(M_3);                      // Send the message to the bus
}
```