

IMPLEMENTATION OF CONTROLLER AREA NETWORK (CAN) BUS IN AN  
AUTONOMNOUS ALL-TERRAIN VEHICLE

by

Sunil Kumar Reddy Gurram

A thesis submitted to the faculty of  
The University of North Carolina at Charlotte  
in partial fulfillment of the requirements  
for the degree of Master of Science in  
Electrical Engineering

Charlotte

2011

Approved by:

---

Dr. James M. Conrad

---

Dr. Ronald Sass

---

Dr. Yogendra Kakad

©2011  
Sunil Kumar Reddy Gurram  
ALL RIGHTS RESERVED

## ABSTRACT

SUNIL KUMAR REDDY GURRAM. Implementation of controller area network (CAN) bus in an autonomous all-terrain vehicle. (Under the direction of DR. JAMES M. CONRAD)

The University of North Carolina Charlotte is provided with a Honda Four Trax Ranch ATV by Zapata Engineering (a small business firm located in Charlotte, NC) which is intended to tow a trailer of ground sensing equipment. The mechanical control system of the all-terrain vehicle (ATV) is converted to an electronic control system and is interfaced to a wireless radio system. The control system of the ATV is designed to run autonomously with the help of LIDAR, GPS and camera and can also be controlled over the wireless radio system. In order to improve the control system design and reduce the wiring, a Controller Area Network (CAN) control system has been implemented which is very flexible and reliable.

A CAN control system contains electronic control units (ECU) which communicate over CAN protocol. CAN protocol is a serial communication protocol which is internationally standardized by ISO and it creates a two line differential bus for communication. It is a widely used real time communication protocol designed mainly for in vehicle networking but also gained popularity in many embedded applications. This thesis presents a design and implementation of a prototype CAN control system for the ATV with the LIDAR, GPS and IMU connected to the ECU.

## ACKNOWLEDGEMENTS

I would like to be thankful to all the people who are involved directly or indirectly in the completion of my thesis. First and foremost I would like to express my grateful thanks to my advisor Dr. James M Conrad for his encouragement and guidance throughout my Masters program.

I would also like to thank Richard McKinney for providing valuable ideas and support towards completion of my thesis. Last but not the least, I would like to thank my parents, family and friends for providing me all kind of support to achieve my masters.

## TABLE OF CONTENTS

LIST OF FIGURES	viii
LIST OF TABLES	x
LIST OF ABBREVIATIONS	xi
CHAPTER 1: INTRODUCTION	1
1.1 UNC Charlotte Background	2
1.2 Past of the Zapatabot	2
1.3 Thesis Contribution	3
1.4 Organization of thesis	4
CHAPTER 2: CAN BASICS	5
2.1 Overview	5
2.1.1 Characteristics of CAN protocol	5
2.2 Structure of a node in CAN network	7
2.3 Types of Frames and their Architectures	8
2.3.1 Data and Remote Frame	8
2.3.2 Error Frame	12
2.3.3 Overload frame	12
2.4 Bus Arbitration	13
2.5 Message Broadcasting	14
2.6 Data Transfer Synchronization	15
2.7 Error detection and Fault confinement	16
2.8 Benefits of using CAN	18

CHAPTER 3: LITERATURE REVIEW	19
CHAPTER 4: DESIGN OF THE ZAPATABOT	21
4.1 Mechanical Design	21
4.1.1 Steering	22
4.1.2 Throttle	22
4.1.3 Braking	23
4.2 Electrical Design	23
4.2.1 Light Detection And Ranging (LIDAR)	25
4.2.2 Global Positioning System (GPS)	26
4.2.3 Inertial Measurement Unit (IMU)	27
CHAPTER 5: IMPLEMENTATION OF CAN ON ZAPATABOT	29
5.1 Renesas RX62N Evaluation Board	29
5.2 Implementation of CAN protocol on RX62N	30
5.2.1 Self Testing of CAN module on RX62N	31
5.2.2 Baud Rate of CAN protocol	34
5.2.3 Interrupts of CAN protocol on RX62N	34
5.2.4 Algorithm of CAN protocol on RX62N	34
5.3 CAN Control System for the ATV	36
5.3.1 Requesting Data from LIDAR	38
5.3.2 Requesting Data from GPS	41
5.3.3 Requesting Data from IMU	42
CHAPTER 6: PROTOTYPE DESIGN	43
6.1 Hardware implementation of RS232-CAN Communication Bridge	43

6.2	Software Implementation of RS232-CAN Communication Bridge	45
6.3	Final Prototype	48
6.4	Data received over CAN bus	50
CHAPTER 7: CONCLUSION AND FUTURE SCOPE		53
7.1	Conclusion	53
7.2	Future Scope	54
BIBLIOGRAPHY		55

## LIST OF FIGURES

FIGURE 2.1: ISO/OSI Reference model for CAN protocol.	8
FIGURE 2.2: Architecture of Data and Remote frame.	9
FIGURE 4.1: Zapatabot ATV.	21
FIGURE 4.2: DC steering motor [11].	22
FIGURE 4.3: Servo motor mounted to vehicle's throttle body [11].	23
FIGURE 4.4: H-bridge motor control mounted to linear actuator for braking [11].	23
FIGURE 4.5: Design of the ATV [16].	24
FIGURE 4.6: SICK LMS 200 LIDAR [16].	25
FIGURE 4.7: Garmin GPS25HVS receiver with the antenna.	27
FIGURE 4.8: 9 DOF Razor IMU [17].	28
FIGURE 5.1: Renesas RX62N evaluation board.	30
FIGURE 5.2: CTx0 and CRx0 connections for listen only mode [4].	32
FIGURE 5.3: CTx0 and CRx0 connections for self test mode 0 [4].	33
FIGURE 5.4: CTx0 and CRx0 connections for self test 1 mode [4].	33
FIGURE 5.5: Control System design of the ATV with CAN bus.	37
FIGURE 5.6: Commands to change the baud rate [16].	38
FIGURE 5.7: Commands to change the angular range and resolution [16].	38
FIGURE 5.8: Number of data values based upon the settings [16].	39
FIGURE 5.9: Command to start the continuous stream of data [16].	39
FIGURE 5.10: Description of the output data string [16].	40
FIGURE 5.11: Packets of output string [16].	40



FIGURE 5.12: Command to stop the continuous stream of data [16].	40
FIGURE 6.1: RS232-CAN communication bridge execution flow.	46
FIGURE 6.2: GUI for RS232-CAN module.	47
FIGURE 6.3: Prototype of the control system for the ATV.	48
FIGURE 6.4: Graphic display on RX62N.	49
FIGURE 6.5: LIDAR data received over CAN bus.	50
FIGURE 6.6: GPS data received over CAN bus.	51
FIGURE 6.7: IMU data received over CAN bus.	52

## LIST OF TABLES

TABLE 2.1: Setting of DLC field for required amount of data.	10
TABLE 2.2: Bus arbitration on CAN bus	14
TABLE 6.1: Baud rate selection with DIP switch.	44

## LIST OF ABBREVIATIONS

AHRS	Attitude and Heading Reference System
ASCII	American Standard Code for Information Interchange
ATV	All Terrain Vehicle
CAN	Controller Area Network
CCD	Charge Coupled Device
CRC	Cyclic Redundancy Check
DIP	Dual In-line Package
DLC	Data Length Code
ECU	Electronics Control Units
EOF	End Of Frame
EPS	Electric Power Steering
FIFO	First In First Out
GPS	Global Positioning System
HEW	High-performance Embedded Workshop
IDE	Integrated Development Environment
IMU	Inertial Measurement Unit
INS	Inertial Navigation System
LIDAR	Light Detection And Ranging
LIN	Local Interconnect Network
NMEA	National Marine Electronics Association
PWM	Pulse Width Modulation

RTR	Remote Transmission Request
SOF	Start Of Frame
SONAR	SOund Navigation And Ranging
TCP/IP	Transmission Control Protocol/ Internet Protocol
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

## CHAPTER 1: INTRODUCTION

The recent technology trends in the automobile industry are bringing more safety and comfort in a vehicle by incorporating automation techniques like collision avoidance, air bag deployment and entertainment devices. In the process of making an automated vehicle, there was a rapid increase in the use of electronic control units (ECU) in the vehicle. Therefore, there was a need for a special communication system for achieving the communication between the ECUs in a vehicle. Initially, multiplexed communication was implemented which decreased the interconnections (cables) between the ECUs. The main problem with the multiplexed communication system was it could not communicate data in real time. In 1980's, BOSCH corporation designed a multi master serial communication protocol called Controller Area Network (CAN) protocol for robust and real time for in-vehicle networking.

The first vehicle with the CAN protocol was implemented in 1986 and reduced 2km of wiring and 50kgs of weight in the vehicle. Since then, CAN protocol has become the most widely used communication protocol for in-vehicle networks. Today, the CAN protocol is advancing into many automated systems like vehicle automation, home automation and medical equipment.

## 1.1 UNC Charlotte Background

In the Embedded Systems Research lab at UNC Charlotte, researchers had the task of building an autonomous all terrain vehicle (ATV) which can be used to tow a trailer, carrying sensitive ground scanning equipment. The ATV was designed to run autonomously with the help of a Light Detection and Ranging (LIDAR) unit, camera and GPS. A PC is used to give instructions through a microcontroller to the control circuitry of the vehicle based upon the data from sensor equipment (LIDAR and GPS). The project was named as Zapatabot.

## 1.2 Past of the Zapatabot

The main aim of the Zapata bot project is to design an autonomous robotic vehicle which can tow a trailer in diversified terrains. The University of North Carolina at Charlotte decided to use Honda “Four Trax Rancher AT” ATV and the project was named Zapata bot as it was funded by Zapata Engineering, a small business firm. The project was divided between five design teams:

- Localization team – This team was responsible for gathering data from GPS and other sensors which provide localization information.
- Environmental sensing team – This team was responsible for gathering and analyzing environmental data from different sensors like LIDAR, SONAR and a camera. A PC is used on the ATV to analyze the data.
- Mapping team – This team was responsible to create a map of the surrounding area with the data acquired from the localization and environmental sensing teams.

- Path planning team – This team was responsible to move the ATV in the desired path avoiding obstacles.
- ATV control circuitry team – This team was responsible for designing the control circuitry for the ATV such that it can be controlled wirelessly. The control circuitry has been designed in such a way that it can be interfaced to any microcontroller to run the ATV autonomously.

The QSK30P evaluation board with a Renesas M16C/30P group microcontroller controls the power steering, throttle and the braking system using a stepper motor and a servo motor. The microcontroller on the evaluation board is programmable over USB through an IDE developed by renesas called “High-performance Embedded Workshop”. The microcontroller uses the on chip peripheral devices like timers for controlling the motors with a PWM pulse and a UART for communicating with the PC. The data accumulated from different sensors are sent to the PC. The PC analyzes the data and issues corresponding instructions to the control circuitry through the microcontroller. The ATV is wirelessly controlled using the Spectrum DX6 remote with a Spectrum BR6000 receiver.

### 1.3 Thesis Contribution

In order to make the Zapatabot design and performance more effective, researchers had an idea of implementing an advanced microcontroller which could create a communication bus for the LIDAR and GPS data. They have decided to implement CAN bus on the ATV as it was the widely used communication bus for in -vehicle networking. The other advantages of using CAN bus are:

- Reduction in the amount of I/O ports used in a microcontroller.
- Reduction in the interconnections
- Real time communication
- Error detection and fault confinement

This thesis focuses on the development of a CAN network for the ATV. The Renesas RX62N is the CAN controller which creates a CAN bus for the communication of data. The CAN module on RX62N group implements one channel of CAN protocol according to the specification of ISO 11898-1. Each CAN controller on the bus is called a node. Each node is configured to receive data from the sensor equipment (LIDAR, GPS and IMU). The data is finally transmitted to the PC over CAN bus.

#### 1.4 Organization of thesis

The rest of thesis is organized as follows. Chapter 2 describes the basics of CAN protocol and its operation. Chapter 3 presents the previous implementations of the CAN protocol in robotics. Chapter 4 explains the implementation of the CAN protocol on Zapatabot. Chapter 5 provides the results of the implementation. Finally, Chapter 6 provides the conclusion and future work of the thesis and the references used for this thesis are provided in the end.



## CHAPTER 2: CAN BASICS

### 2.1 Overview

Controller Area Network (CAN) is an asynchronous serial communication protocol which follows ISO 11898 standards and is widely accepted in automobiles due to its real time performance, reliability and compatibility with wide range of devices. CAN is a two wire differential bus with data rates up to 1Mbps and offers a very high level of security. Its robust, low cost and versatile technology made CAN applicable in other areas of applications where inter processor communication or elimination of excessive wiring is needed. Some of the areas it is widely used are industrial machinery, avionics, medical equipments and home automation etc.

#### 2.1.1 Characteristics of CAN protocol

The main characteristics of CAN protocol are

- Multi master hierarchy
- Priority based bus access
- Baud rate up to 1Mbits/sec
- Error detection and fault confinement

A CAN bus is a half duplex, two wire differential bus. The two lines, CAN\_L and CAN\_H, form the communication bus for the nodes to transmit data or information. The logic levels used on the bus are dominant and recessive levels, where dominant level is referred when TTL = 0V and recessive level is referred when TTL = 5V. The dominant level always overrides recessive level and this concept is used to implement the bus arbitration.

The voltage levels on the CAN bus varies from 1.5 volts to 3.5 volts. The logic levels are calculated as the voltage difference between the two lines.

$$V_{diff} = V_{CAN\_H} - V_{CAN\_L} \quad (2.1)$$

If the difference voltage ( $V_{diff}$ ) is 2 volts, it is considered as a dominant level and if it is 0 volts, it is considered as a recessive level.

In the CAN protocol, nodes communicate data or information through messages termed as frames. A frame is transmitted on to the bus only when the bus is in idle state. There are four different types of frames which are used for communication over CAN bus.

- Data Frame – Used to send data
- Remote Frame – Used to request data
- Error Frame – Used to report an error condition
- Overload Frame – Used to request a delay between two data or remote frames.

The frames transmitted from one node will be received by all the other nodes on the network using message broadcasting. The message filtering which is provided by the CAN controller hardware, decides whether the received frame is relevant to that node or

not. If any error occurs due to reception or transmission, an error frame will be transmitted on the bus to let the network know of the error. Since the error frame starts with a 6 dominant bits, it will have highest priority when the bus is idle. As soon as the error is detected, the CAN protocol implements the fault confinement techniques to overcome the error. The fault confinement feature in the CAN protocol differentiates between a temporary error and a permanent failure of a node. If the error is due to permanent failure of the node, it automatically detaches the defective node from the bus without causing any problems to the network.

## 2.2 Structure of a node in CAN network

The CAN controller implements only three layer of the ISO/OSI Reference model in a node. It creates a bridge from Data link layer to Application layer (as shown in FIGURE 2.1) in order to limit the resources and to improve the performance. The other layers i.e. Layer 3 to Layer 6 are implemented in higher layer protocols like CANopen, J1939 and DeviceNet. The physical layer and data link layer are integrated on the CAN controller chips and the libraries for the connection between the data link layer and the application layer are provided by the CAN chip manufacturers.

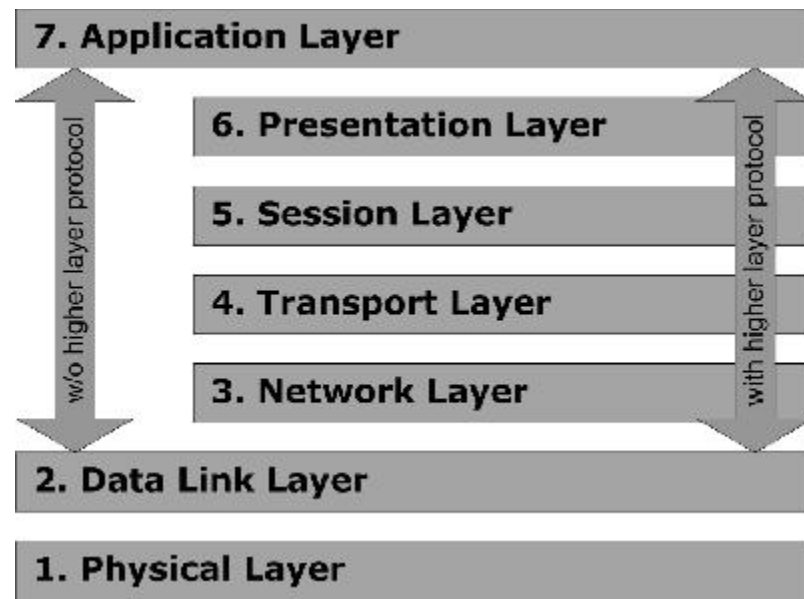


FIGURE 2.1: ISO/OSI Reference model for CAN protocol.

### 2.3 Types of Frames and their Architectures

As mentioned earlier CAN provides four different types of message frames for communication, the architecture of each frame is discussed in this section.

#### 2.3.1 Data and Remote Frame

The architecture of the data and the remote frame are exactly the same. A data frame has higher priority than a remote frame. Each data and remote frame starts with a Start Of Frame (SOF) field and end with an End Of Frame (EOF) field. The FIGURE 2.2 gives architecture of data and remote frames.

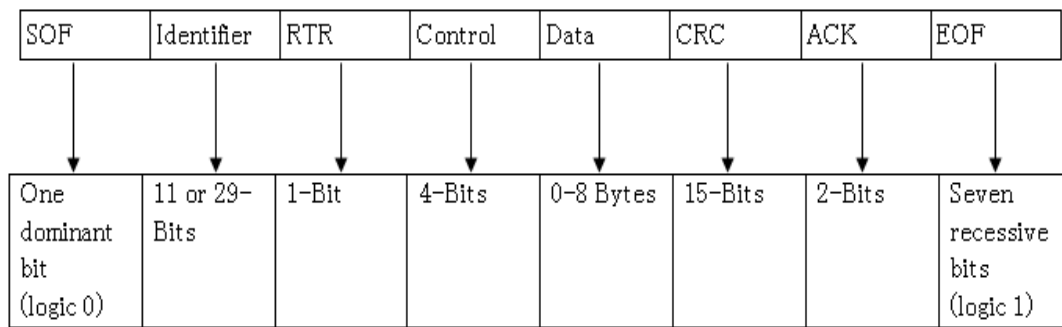


FIGURE 2.2: Architecture of Data and Remote frame.

The following are the fields in data and remote frame:

- SOF field (1 bit) – Indicates the beginning of the frame. A single dominant bit represents a start of a frame. It is also used for data transfer synchronization.
- Arbitration Field – This contains two sub fields, Message Identifier and RTR field.
  - Message Identifier (11/29 bits) – This field contains a message ID for each frame which is either 11 (standard ID) or 29 (Extended ID) bits. No two message frames in the CAN network should have the same message ID. A message ID which has a low decimal value is considered as a high priority message.
  - Remote Transmission Request (RTR) (1 bit) – The RTR field distinguishes a data frame from a remote frame.
- Control Field (6 bits) – This contains two sub fields, IDE and DLC field
  - Identifier Extension (IDE) Bit (1 bit) – This bit indicates the format of the message ID in the frame, either a standard 11-bit format or extended 29-bit format.

- Data Length Code (DLC) field (4 bits) – This field is used to set the amount of data being transferred from one node to other node. In a remote frame, these bits represent the amount of data it is requesting. The following are the values of the DLC bits for the corresponding amount of data.

TABLE 2.1: Setting of DLC field for required amount of data.

DLC (3 bits)	Amount of data
dddd	0 bytes
dddr	1 byte
ddrd	2 bytes
ddrr	3 bytes
drdd	4 bytes
drdr	5 bytes
drrd	6 bytes
drrr	7 bytes
rddd	8 bytes

Where “r” represents recessive level.

“d” represents dominant level.

- Data Field – This field contains the actual data and it is not applicable for remote frame.

- CRC field (16 bits) – The CRC field consists of the CRC Sequence and a CRC Delimiter bit.
  - CRC Sequence field (15 bits) – This 15 bit field contains the frame check sequence without the stuffing bits.
  - CRC Delimiter bit (1 bit) – This bit is used to provide processing time for the CRC Sequence field.
- ACK field (2 bits) – The ACK field consists of a 1 bit Acknowledgement Slot field and Acknowledgment Delimiter bit (which is always recessive).
- EOF field (7 bits) – Indicates the end of the frame. A seven bit continuous recessive bit represents the end of frame.

A node uses data frame to transmit data to any other node on the network. The RTR field determines whether the message frame should act as data frame or a remote frame. When the RTR bit is set to dominant level, then the message frame will act as a data frame. A maximum of 8 bytes of data can be transferred using a single data frame. Each data frame will be assigned a unique message ID using which the node decides whether the data is relevant or not.

A remote frame is used to request a data frame from any node on the network. When the RTR bit is set to recessive level, then the message frame will act as a remote frame. While requesting data from a node, the length of the data field in control field (DLC bits) of the remote frame should be same as the requesting data frame otherwise a bus collision occurs. As soon as the remote frame is accepted by a node, a data frame will

be transmitted on to the bus with the requested data. When two or more nodes on the network request the same message at the same time a bus collision occurs.

### 2.3.2 Error Frame

An error frame is transmitted onto the bus whenever a transmission or reception error occurs due to a faulty node or bus problems. An error frame consists of three fields

- Error flag (6 bits) – It is 6 dominant bits which indicates the transmitting or receiving error on the bus.
- Error Delimiter (8 bits) – It is represented as a sequence of 8 recessive bits. After transmitting the error flag each node transmits a single recessive bit and waits for the bus level to change to recessive. Only after the bus level is recessive, the remaining 7 recessive bits will be sent onto the bus.
- Interframe Space (3 bits) – It is represented as minimum space between any type (data, remote, error, overload) of frame and a following data or remote frame. It contains of 3 recessive bits.

The error delimiter and the Interframe space are used to synchronize the nodes to the error frame transmitted on to the bus.

### 2.3.3 Overload frame

The overload frame takes the same form of the error frame but the overload frame is used to request a delay between the transmission of the next data or remote frame. It consists of two fields, Overload flag and an Overload delimiter.



- Overload flag (6 bits) – It contains 6 dominant bits which indicates the transmitting or receiving error on the bus.
- Overload Delimiter (8 bits) – It is represented as a sequence of 8 recessive bits. After transmitting the error flag each node transmits a single recessive bit and waits for the bus level to change recessive. Only after the bus level is recessive, the remaining 7 recessive bits will be sent onto the bus.
- Interframe Space (3 bits) – It is represented as minimum space between frames of any type (data, remote, error, overload) of frame and a following data or remote frame. It transmits 3 consecutive recessive bits on to the bus. When the Interframe space is being transmitted, no node on the network is allowed to transmit any of the frames except the overload frame.

## 2.4 Bus Arbitration

In a single bus communication protocol, when two or more nodes request access to the bus then the bus arbitration technique comes in. Usually bus access will be given to a node with a high priority. The bus arbitration technique also reduces data collisions.

CAN protocol provides a non-destructive bus arbitration mechanism. It assigns a recessive level to the bus only if all the nodes on the bus output a recessive level and it assigns a dominant level if any one of the nodes on the network output a dominant level. When a dominant bit and recessive bit request access for the bus, the dominant bit is given the access as it is considered as the high priority. So the bus arbitration on CAN network follows an AND gate logic as shown in table.

TABLE 2.2: Bus arbitration on CAN bus

Node1	Node 2	Bus logic level
Dominant	Dominant	Dominant
Dominant	Recessive	Dominant
Recessive	Dominant	Dominant
Recessive	Recessive	Recessive

When transmitting a frame on to the bus, the bus access to a node will be given based upon the message ID of the frame. As the dominant level is considered as high priority, the message ID with more dominant bits is considered as a high priority message. When the bus is idle, the bus access will be assigned to the node which transmits a message with a higher priority.

## 2.5 Message Broadcasting

CAN protocol is based on message broadcasting mechanism, in which the frames transmitted from one node is received by every other node on the network. The receiving nodes will only react to the data that is relevant to them. Messages in CAN are not acknowledged due to unnecessary increase of traffic. But the receiving node checks for the frame consistency and acknowledges the consistency. If the acknowledge is not received from any or all the nodes of the network, the transmitting node posts an error message to the bus. If any of the nodes are unable to decode the transmitted message due to internal malfunction or any other problem, the entire bus will be notified of the error and the node re-transmits the frame. If there is an internal malfunction in a node, that

particular node reports an error for each frame it receives. Due to this most bandwidth of the network will be allocated to error frames as they have higher priority (starts with 6 consecutive dominant bits). To overcome this problem the CAN protocol supports a bus off state in a node, in which the node will be detached from the bus if it reports an error for more than a pre defined value. The bus off state of a node is implemented to avoid the breakdown of the network due to a single node.

While broadcasting data frames on the bus, each node on the bus receives every data frame transmitted on to the bus. As CAN protocol does not support IDs for the nodes and the receiver does not know the information of the transmitter of the frame, each data frame goes through an acceptance filtering process at the receiving node, which is dependent on the message ID (standard or extended) of the frame.

The process of data requesting in CAN protocol is carried out by the remote frame. The RTR bit in a frame decides whether the frame is a remote frame or data frame. When the RTR bit is set to recessive level the frame will act as a remote frame. When a node is requesting a data frame from another node, the message identifier section (ID bits) and data length section (DLC bits) in the remote frame should be of same value of that in the data frame that is requested otherwise an error will be reported on the bus.

## 2.6 Data Transfer Synchronization

Each node in the CAN network will have different oscillators running at different frequencies, so to make all the nodes work synchronously while transferring data, the CAN protocol uses the falling edge of the SOF bit (transition from recessive to dominant bus level).

The bit coding used in CAN bus is Non-Return-to-Zero principle in which the bit level remains constant during the entire bit time, which creates a node synchronization problem during the transmission of larger bit blocks of same polarity. To overcome this problem CAN protocol uses bit stuffing mechanism.

*Bit Stuffing:* The CAN protocol allows only 5 consecutive bits of same polarity between the SOF bit and Data field. If more than 5 consecutive bits of same polarity are transmitted on to the bus it will be considered as an error condition. So to transmit data with more than 5 consecutive bits of same polarity the CAN protocol inserts a complementary bit of opposite polarity at the transmitter end and at the receiver end the filtering should be performed to get rid of the stuffed bit. Bit stuffing is applied only in data and remote frames and it is not applicable after CRC field.

## 2.7 Error detection and Fault confinement

The CAN protocol implements a series of error detection mechanisms which contributes to the high level of reliability and error resistance. It also implements fault confinement mechanisms for proper function of the network. The error detection mechanisms implemented are

- Bit monitoring – Transmitter compares each bit that is transmitted on to the bus with the data it is transmitting and reports an error if there is change in the data transmitted.
- Checksum check – Every data and remote frame has a 15-bit CRC field which carries the checksum of the frame and is used to detect errors at the receiver.

- Bit stuffing – CAN protocol allows only 5 consecutive bits of same polarity. Bit stuffing is implemented during transmission of more than 5 consecutive bits of same polarity. If more than 5 consecutive bits of same polarity are transmitted, the bus takes it as an error frame as the first 6 bits of the error frame are dominant bits.
- Frame check: Each transmitting and receiving node checks for the consistency of the frames.
- Acknowledge check: Each receiving node transmits frame consistency acknowledge to the transmitting node.

Whenever an error occurs on the bus, each node on the network receives the error frame and the transmitting node serves the error by re-transmitting the frame.

The CAN protocol implements fault confinement techniques to ensure that the communication on the network never fails. Consider a situation in which a node has an internal malfunction caused due to electrical disturbances and transmits error frame for every frame it receives. For serving these kind of errors, CAN protocol is supplied with two counters, a transmit error counter and a receive error counter. The corresponding counter is incremented each time a failure in the transmission/reception occurs. The counter is decremented whenever there is a successful transmission/reception. The counter value does not decrement when the value is zero. Based upon the values of two counters the CAN nodes will have three states, Error active, Error passive and Bus off state.

- Error active state – Every node after reset starts with this state in which it transmits an error flag (6 consecutive dominant bits) whenever it receives an error frame.
- Error passive state – A node enters into this state when a receive error counter or transmit error counter value is equal to or greater than 127. When the node is in error passive state, it transmits an error flag with 6 consecutive recessive bits.
- Bus off state – A node enters into this state when a transmit error counter value increases more than 255.

## 2.8 Benefits of using CAN

Controller Area Network is a serial communication protocol which is mainly used for reducing wired interconnections in a vehicle. Some of the benefits in implementing CAN protocol in automobiles are

- Reduced wired interconnections
- Low cost implementation
- Speed, reliability and error resistance
- Worldwide acceptance

## CHAPTER 3: LITERATURE REVIEW

This section discusses about some of the CAN protocol designs used to build an autonomous robot. The following are some of the designs of CAN protocol implemented in robotics

- In a paper titled “Application of Controller Area Network to Mobile Robots”, architecture of the mobile robot with CAN protocol is discussed. Architecture of the robot contains obstacle avoidance, contour following, dead reckoning, planning, vision, regulation and user interface modules [10].
- In a paper titled “A CAN architecture for an intelligent mobile robot”, an intelligent autonomous robot using CAN bus is designed. It mainly focuses on the real time analysis of sensor systems, data fusion algorithms and field buses [5].
- In a paper titled “Localization of a Mobile Robot using Images of a Moving Target”, design of an autonomous robot which tracks a particular object using a CCD camera is discussed. In this design, control of the motor movement for the wheels and camera are acquired over CAN bus [2].
- In a paper titled “Development of a Home service Robot ISAAC”, a home service robot used for vacuum cleaning and home security is designed. The robot consists

of sensors and a USB camera. The sensor data is communicated over CAN bus and for the rest of the data TCP/IP protocol is implemented [3].

From the literature review, each CAN node designed carries a different firmware which limits the use of it only to that particular application. Keeping in mind the reusability of the device, a design of an RS232-CAN communication bridge has been implemented. The RS232-CAN communication bridge can be used in any application where a CAN bus needs to be implemented and the devices which give digital data can be programmed to output data on a serial port. The module implementation has been tested as a vehicle network bus for the ATV.



## CHAPTER 4: DESIGN OF THE ZAPATABOT

The FIGURE 4.1 shows the ATV used for the design of the Zapatabot.



FIGURE 4.1: Zapatabot ATV.

### 4.1 Mechanical Design

The main task of the mechanical design of the ATV is to interface the vehicle controls to a motor or actuator for electrical interface to a motor driver and a microcontroller. The mechanical design is based on a previous project done by Richard McKinney and the UNC Charlotte Senior Design Program [11]. The following are the mechanical components of the ATV that need to be interfaced to a microcontroller.

#### 4.1.1 Steering

A DC motor is provided to the steering system with the Honda Fourtrax EPS. In order to interface the steering system to a microcontroller, the power assist system has been detached and the DC motor is used as the steering motor. An analog rotary encoder sensor is mounted to the bottom of the steering shaft as a feedback for the directional position of the steering column and the current angular heading of the vehicle.



FIGURE 4.2: DC steering motor [11].

#### 4.1.2 Throttle

The throttle body of the Honda ATV has a spring return and it returns to an idle position when it is not engaged. The vehicle does not move or slowly comes to stop when the throttle is in an idle position. A standard servo from parallax has been interfaced to control the throttle. The FIGURE 4.3 shows the servo installed to the throttle body with the help of a bracket (white in color).

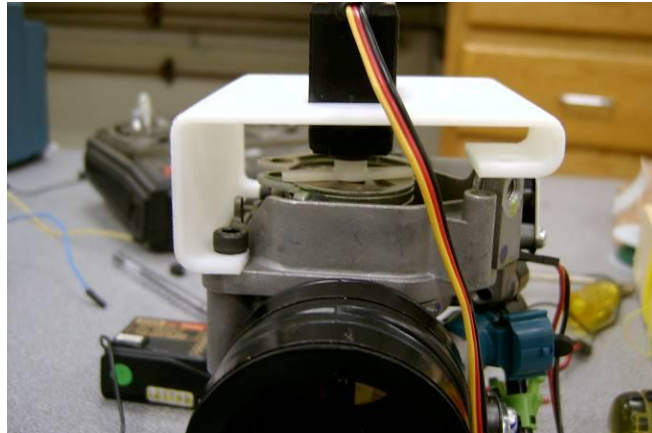


FIGURE 4.3: Servo motor mounted to vehicle's throttle body [11].

#### 4.1.3 Braking

The Honda “Four Trax Rancher EPS” ATV is provided with a hand brake, which is linked to a brake on one of the front wheels. The foot brake is linked to a brake on the rear axle. A bracket has been designed to attach the electromechanical linear actuator to the foot brake and the linear actuator is controlled by a motor.



FIGURE 4.4: H-bridge motor control mounted to linear actuator for braking [11].

## 4.2 Electrical Design

The following (FIGURE 4.5) block diagram shows the design of the Zapatabot

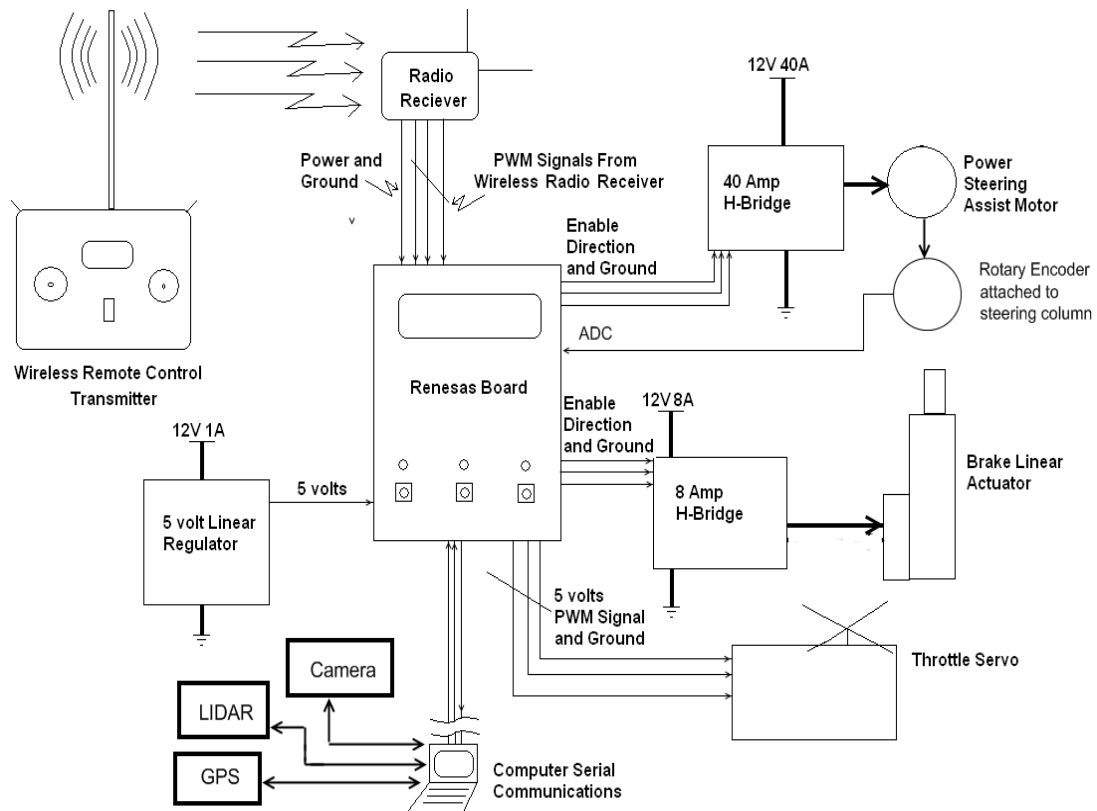


FIGURE 4.5: Design of the ATV [16].

A Renesas QSK/62P microcontroller is used for controlling the motors corresponding to the data from the sensors. The sensors are connected to the PC and the motors are controlled through H-bridges. The PC sends data to the microcontroller through a serial RS232 interface. In autonomous mode of the ATV, the PC analyzes the data from different sensors interfaced to it and sends corresponding instructions to the microcontroller which moves the motors accordingly. The sensors used for localization and path planning are LIDAR, camera and GPS. Though the camera part is not fully implemented, the LIDAR and GPS have been implemented and tested. An IMU is also in plan to be implemented on the ATV. A wireless radio system was used to control the

ATV remotely when not in autonomous mode. A Spektrum 'DX6' transmitter and 'BR6000' receiver are used a wireless radio system.

#### 4.2.1 Light Detection And Ranging (LIDAR)

The first and foremost task in an autonomous robot is detecting the obstacles in its path and re routing the robot away from the obstacle. In Zapatabot, a SICK LMS 200 LIDAR is being used for detecting the obstacles. LIDAR is also known as laser range finder which measures distance of an object based upon the principle of "time of flight". The FIGURE 4.6 shows the LIDAR used on the ATV.



FIGURE 4.6: SICK LMS 200 LIDAR [16].

The SICK LMS 200 LIDAR covers a distance up to 80 meters with a 180 degree field of view, which gives a two dimensional view of the world. It uses a rotating mirror to sweep a 180 degrees field of view and has a refresh rate of 80 times a second. The data from the LIDAR is transferred on to the user interface through RS-232 serial communication. The RS-232 data contains the distance the laser beam has travelled (i.e.

distance of the object from the LIDAR) at 180 different points as the LIDAR is configured for 180 degrees field of view. If we need less precise data we can also configure LIDAR for only 100 degree field of view which gives you only 100 data points. The angular resolution can also be switched between 0.25 or 0.5 or 1 degree. For example if you setup the LIDAR 180 degrees field of view and a angular resolution of 0.5 degrees, we get  $360(180/0.5)$  data points.

#### 4.2.2 Global Positioning System (GPS)

The Global Positioning System (GPS) is a satellite based navigation system which receives information from the satellites and uses triangulation method to determine the exact location. For a GPS receiver to get an exact 2D position (latitude and longitude) it should receive data from at least three satellites and for 3D position (latitude, longitude and altitude) it should receive data from four or more satellites. The GPS on the ATV is used to obtain the positional information of the vehicle.

Most of the GPS receivers available in the market provide accuracy between 1.5 meters and 10 meters. For the ATV, as the positional information is mainly used to calculate the speed and the distance travelled, a GPS with accuracy up to 5 meters is acceptable. So the researchers chose a Garmin GPS 25LP series GPS sensor board, which tracks up to twelve satellites providing every one second navigation updates. The GPS25-HVS receiver operates from 6 volts to 40 volts DC voltage and outputs the data on a RS232 serial port. It provides a position accuracy of 15 meters with non differential GPS and less than 5 meters accuracy with a differential GPS. The FIGURE 4.7 shows the GPS 25HVS receiver with the antenna (black wire) attached to it.





FIGURE 4.7: Garmin GPS25HVS receiver with the antenna.

#### 4.2.3 Inertial Measurement Unit (IMU)

The Inertial Measurement Unit (IMU) is a device which measures the velocity, orientation and gravitational forces by using the accelerometers and gyroscopes. The IMU with a group of inertial sensors (gyroscopes and accelerometers) can be used in two ways, an IMU which transfers raw data from the inertial sensors and an Inertial Navigation System (INS) which transfers data to a navigation system which calculates position, velocity and attitude of a vehicle.

A 9 Degrees Of Freedom Razor IMU is used on the ATV for the inertial measurement of the vehicle. It provides 9 degrees of inertial measurement with two gyros (single axis and dual axis), one triple axis accelerometer and one triple axis digital

magnetometer. It is Attitude and Heading System (AHRS) compatible, which takes the raw data and estimates only the attitude of the vehicle. The IMU device comes with an on board ATmega328 which collects the data from the sensors and the outputs the data on a serial port. The output pins are compatible to be interfaced with FTDI Basic Breakout, Bluetooth Mate and XBee Explorer. The FIGURE 4.8 shows the IMU used on the ATV.



FIGURE 4.8: 9 DOF Razor IMU [17].



## CHAPTER 5: IMPLEMENTATION OF CAN ON ZAPATABOT

The main goal of the design is to distribute the control over the CAN bus. The initial design of the autonomous ATV is as shown in **Error! Reference source not found..** As the design shows a lot of I/O ports are used to interface each device to the microcontroller and more interconnections (wires) would make the hardware look clumsy. Instead we can substitute all the interconnections by using a single two wire CAN bus. The advantages of implementing CAN bus on the ATV would be

- Decreased wire harnesses
- Easy installation of devices on to the bus
- Error detection and fault confinement
- Does not affect the operation of the bus if a particular node breaks down.
- Real time performance
- Robust to noisy environments

### 5.1 Renesas RX62N Evaluation Board

The Renesas RX62N group has a RX family/RX600 series 32-bit CPU which features high performance and high speed. The rx62n group is equipped with two channels of Usb 2.0, one channel of Ethernet, one channel of CAN bus protocol, timers,

independent watchdog timer and brown out detectors (power on reset and low level voltage detection). The FIGURE 5.1 shows the Renesas RX62N evaluation board.

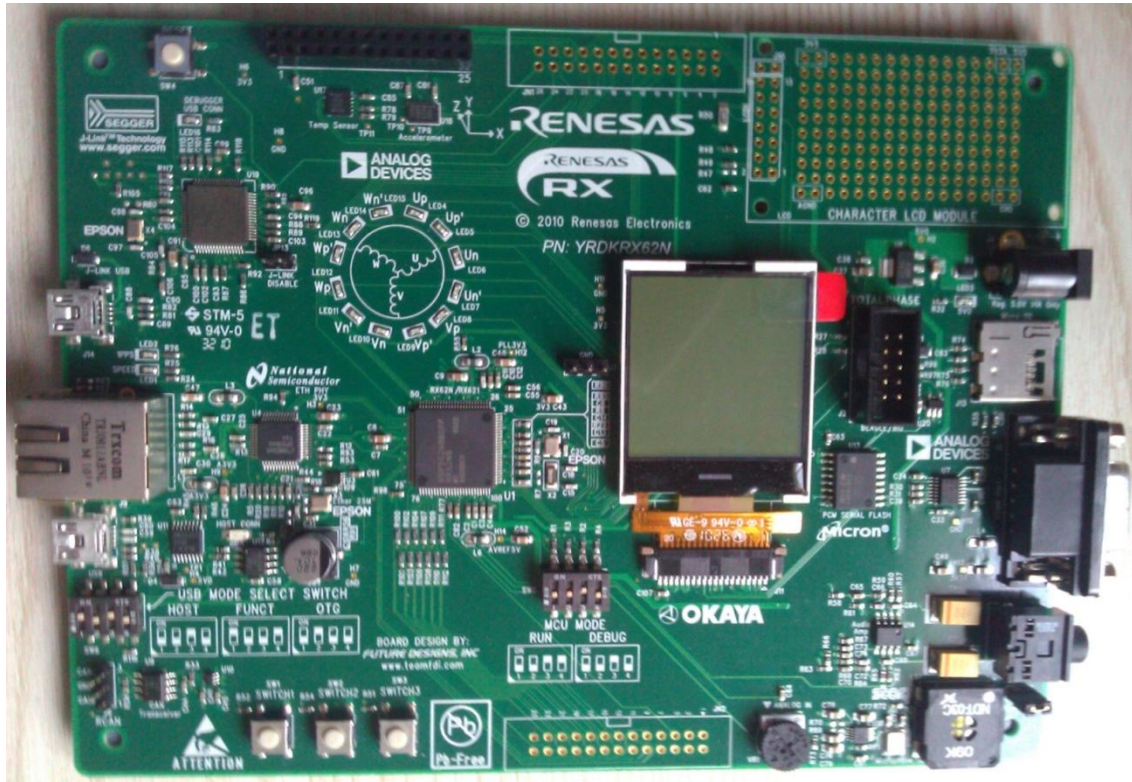


FIGURE 5.1: Renesas RX62N evaluation board.

The CAN module on RX62N group implements one channel of CAN bus protocol according to the specification of ISO 11898-1. This allows communication of messages in both standard identifier (11 bits) and extended identifier (29 bits) and allows data rates up to 1 Mbps.

## 5.2 Implementation of CAN protocol on RX62N

In the implementation part of the CAN protocol, a CAN bus has been created where all the nodes can be connected and tested for correct operation. An algorithm has been developed for CAN communication between RX62N microcontrollers. The

discussion in this section would be about the CAN communication on RX62N and an in detail explanation of the algorithm.

The CAN protocol on RX62N is configured using some set up registers and the mailboxes are used for transmission and reception of data. It has totally 32 mailboxes that can be configured in two different modes

- Normal mailbox mode: In which all 32 mailboxes can be configured to either transmission or reception mailboxes.
- FIFO mailbox mode: In which 24 mailboxes can be configured to either transmission or reception mailboxes. In the remaining 8 mailboxes, first four mailboxes can be configured as FIFO transmission and the other four mailboxes are configured as FIFO reception mailboxes.

A transmission mailbox carries the data to be transmitted onto the bus and the reception mailbox stores the received data. A status register is available for RX62N which records the status of all the events occur in a particular node. Bus off state of a node can also be checked by reading this register.

### 5.2.1 Self Testing of CAN module on RX62N

The test modes on RX62N microcontroller allow us to test the CAN module without connecting the external bus. When a test mode is enabled, the CAN transmit pin (CTx0) is virtually connected to the CAN receive pin (CRx0) and checks for the transmission errors. In all the test modes the CTx0 pin outputs only recessive bits. There are three different kinds of test modes available each one featuring different forms of testing the CAN bus.

- Listen only mode: In this mode, the node can receive valid data and remote frames and the node just monitors the bus. This mode is mainly used for baud rate detection. The FIGURE 5.2 shows the connections of CTx0 and CRx0 pins in listen only mode

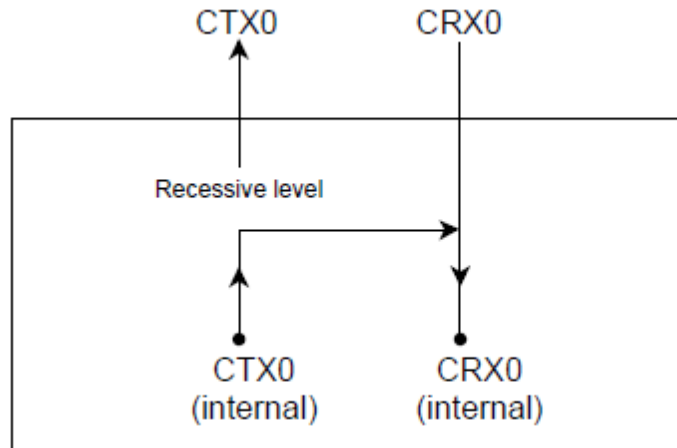


FIGURE 5.2: CTx0 and CRx0 connections for listen only mode [4].

- Self test mode 0 (External loopback): In self test mode 0 the microcontroller receives its own transmitted messages, stores them in a mailbox and sends an ACK bit. This mode is used to test the CAN transceiver so the CAN transmit pin (CTX0) and CAN receive pin (CRX0) should be connected to the transceiver. The FIGURE 5.3 shows the connections of CTx0 and CRx0 pins in self test 0 mode.

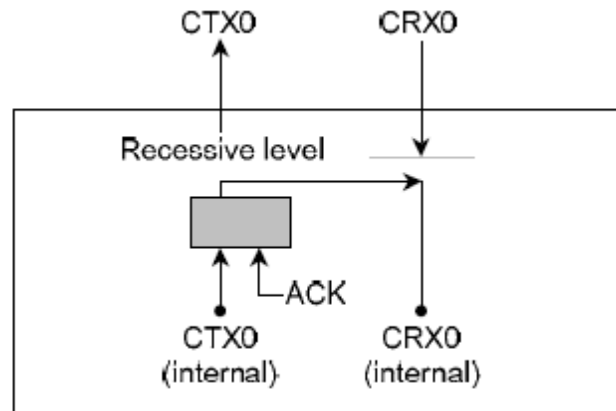


FIGURE 5.3: CTx0 and CRx0 connections for self test mode 0 [4].

- Self test mode 1 (Internal Loopback): This mode is used for self test functions. In self test mode 1 the microcontroller receives its own transmitted messages, stores them in a mailbox and sends an ACK bit. In this mode the CTX0 and CRX0 pins are internally connected and the input from the CRX0 pins is ignored. The **Error! Reference source not found.** shows the connections of CTx0 and CRx0 pins in self test 1 mode.

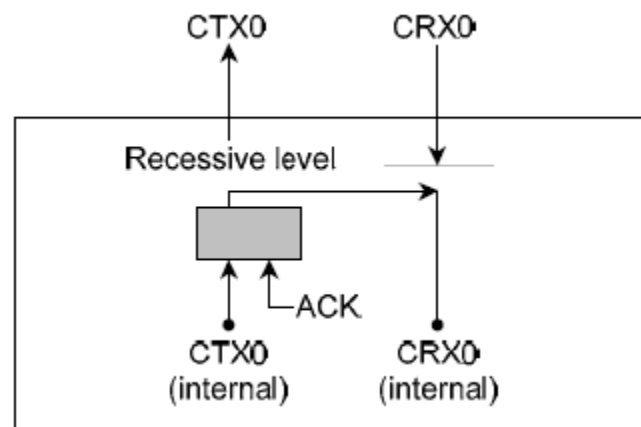


FIGURE 5.4: CTx0 and CRx0 connections for self test 1 mode [4].

These modes can be set up using the CAN0 test control register in RX62N microcontroller.

### 5.2.2 Baud Rate of CAN protocol

The baud rate of the CAN protocol can be set to a maximum of 1Mbps. It can be varied using the external clock source, settings of the internal clock source and prescaler values in the set up registers. The formula used to calculate the baud rate of CAN bus is

$$Bit\ Rate[Bps] = F_{CANCLK} / Segment\ Length \quad (5.1)$$

Where Segment Length is used to synchronize data between two nodes as each node may or may not have the same clock frequency. The default baud rate on the CAN bus used in this project is 500Kbps.

### 5.2.3 Interrupts of CAN protocol on RX62N

The CAN module on RX62N provides with the following interrupts.

- Transmission complete interrupt
- Reception complete interrupt
- Error interrupt
- Transmit FIFO interrupt
- Receive FIFO interrupt

The first two interrupts are used in normal mailbox mode of the CAN protocol whereas the last two interrupts are used in FIFO mailbox mode. The error interrupt is enabled for all the mailboxes in either normal mailbox mode or FIFO mailbox mode. In this project only the reception complete interrupt is being used.

### 5.2.4 Algorithm of CAN protocol on RX62N

The following is the pseudo code for the transmission of data on to the CAN bus using polling

- Enable CAN module.
- Enable the ports for transmission and reception.
- Switch CAN module to reset mode.
- Select the type of mailbox (normal or FIFO), ID (standard or extended).
- Set up required clock speed and corresponding baud rate.
- Select the required test mode if needed.
- Switch CAN module to halt mode or operation mode.
- Select a mailbox for transmission and clear the mailbox.
- Select the length of the data. Set the id and enter the required transmitting data into the mailbox.
- Clear the transmission enable bit of the CAN bus.
- Select the type of transmission (one shot or continuous).
- Set the transmission enable bit of the CAN bus.
- When the sending of data is successful, the sent data status flag will be enabled.
- Clear the sent data status flag for the next transmission.
- Clear the transmission enable bit and set it again for the next transmission.

The following is the pseudo code for receiving of data on the CAN bus using polling

- Enable CAN module.
- Enable the ports for transmission and reception.
- Switch CAN module to reset mode.
- Select the type of mailbox (normal or FIFO), ID (standard or extended).

- Set up required clock speed and corresponding baud rate.
- Select the required test mode if needed.
- Switch CAN module to halt mode or operation mode.
- Select a mailbox for receiving and clear the mailbox.
- Set the required id in the mailbox to receive data only with that particular id.
- Switch CAN module to halt mode.
- Enable the mask for message filtering with a particular id.
- When the data is received the new data status flag will be enabled and the reception complete interrupt occurs.
- When the reception complete interrupt occurs, the received data is saved in a queue.

### 5.3 CAN Control System for the ATV

The FIGURE 5.5 shows the control system design of the ATV with the CAN bus.



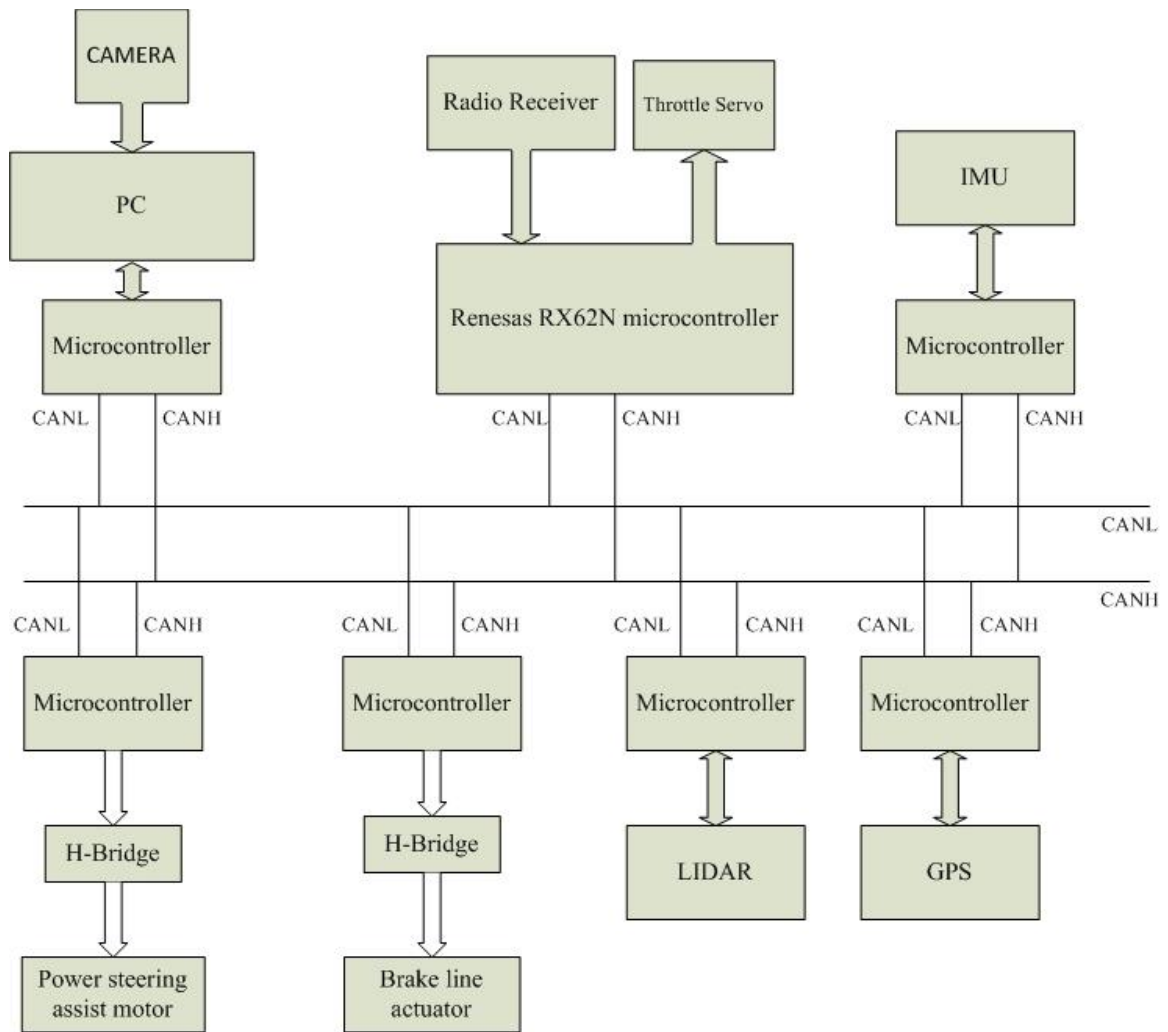


FIGURE 5.5: Control System design of the ATV with CAN bus.

Each node on the CAN bus uses the above transmission and reception algorithms to communicate data over the CAN bus. Each node is termed as an ECU which communicates data with the device connected to it through RS232 serial protocol. The data from the ECU is sent on the CAN bus which is received by all the nodes on the bus and the message filtering technique decides whether to act upon the data or not. In this thesis, the ECU's with LIDAR, GPS and IMU have been implemented and tested. This section discusses about how the data is controlled and transmitted on to the CAN bus.

### 5.3.1 Requesting Data from LIDAR

As mentioned earlier the LIDAR used in this design is a SICK LMS 200. The data from the LIDAR is communicated over RS232 serial protocol with default settings of 9600 baud rate, 180 degrees angular range, 0.5 degrees angular resolution and measurements in mm. Commands should be sent to the LIDAR to change the settings and the LIDAR goes back to default settings each time it is powered down.

The commands used for changing the settings are listed out. The following are the commands to be sent to the LIDAR to change the baud rate

<b>LMS baudrate setting</b>	<b>Telegram code in hex PC → LMS</b>	<b>Reply telegram in hex LMS → PC</b>
9600 baud	02 00 02 00 20 42 52 08	06 02 81 03 00 A0 00 10 36 1A
19200 baud	02 00 02 00 20 41 51 08	06 02 81 03 00 A0 00 10 36 1A
38400 baud	02 00 02 00 20 40 50 08	06 02 81 03 00 A0 00 10 36 1A

FIGURE 5.6: Commands to change the baud rate [16].

The following are the commands to be sent to the LIDAR to change the angular range and resolution

<b>LMS mode</b>		<b>Telegram code in hex PC → LMS</b>	<b>Reply telegram in hex LMS → PC</b>
<b>Angular range</b>	<b>Angular resolution</b>		
0°..100°	1°	02 00 05 00 3B 64 00 64 00 1D 0F	06 02 81 07 00 BB 01 64 00 64 00 10 4A 3F
0°..100°	0.5°	02 00 05 00 3B 64 00 32 00 B1 59	06 02 81 07 00 BB 01 64 00 32 00 10 12 92
0°..100°	0.25°	02 00 05 00 3B 64 00 19 00 E7 72	06 02 81 07 00 BB 01 64 00 19 00 10 BE C4
0°..180°	1°	02 00 05 00 3B B4 00 64 00 97 49	06 02 81 07 00 BB 01 B4 00 64 00 10 5E B2
0°..180°	0.5°	02 00 05 00 3B B4 00 32 00 3B 1F	06 02 81 07 00 BB 01 B4 00 32 00 10 06 1F

FIGURE 5.7: Commands to change the angular range and resolution [16].

Depending on the selected combination of angular range and angular resolution, the following are the amount of data values received in the output string

Angular range	Angular resolution	Number of data values
0° .. 100°	1°	101
0° .. 100°	0.5°	201
0° .. 100°	0.25°	401
0° .. 180°	1°	181
0° .. 180°	0.5°	361

FIGURE 5.8: Number of data values based upon the settings [16].

The LIDAR acknowledges each command with a reply command which lets us know whether the change of the setting was successful or not. There is also a special command for the LIDAR to tell it to start the distance measurement and the distance measurement data would be received in continuous mode.

LMS continuous data output	Telegram code in hex PC → LMS	Data stream in hex LMS → PC
Start	02 00 02 00 20 24 34 08	06 02 81 03 00 A0 00 10 36 1A <output string header> < LMS data > (refer to section D.2.)

FIGURE 5.9: Command to start the continuous stream of data [16].

The data from the LIDAR starts with a header which contains the information of the data being sent. The FIGURE 5.10 shows format of the data from the LIDAR.

Designation	Data size (number of bits)	Remarks
STX	8	Start byte (STX = 02 hex)
ADR	8	Address of the subscriber (in this case the PC) addressed. Typically, the value is (81 hex).
Len	16	Length of the total LMS output data string. Number of following output data string bytes excluding the checksum (CRC = 2 bytes)
CMD	8	Command byte, in this case (B0 hex), which is the command for continuous data output.
DataLen	16	Number of measurement data bytes (depending on measurement mode, refer to section C.8.)
Data ...	n x 16	Measurement data values (2 bytes each) according to the measurement mode settings. Please refer to the note below )*
Status	8	Status byte Indication of system error, pollution etc. Refer to telegram listing for exact information.
CRC	16	CRC Checksum

FIGURE 5.10: Description of the output data string [16].

As shown in the FIGURE 5.10, the data will be accompanied with a 7 byte header and a CRC checksum field at the end of the string. The data comes in the form of packets and the received packets will be in the format as shown in FIGURE 5.11.

STX	ADR	LenL Low byte	LenH High byte	CMD	Data LenL	DataL enH	Data 0° Low byte	Data 0° High byte	Data 1° Low byte	Data 1° High byte	..... (more data)	Status	CRC Low byte	CRC High byte
-----	-----	---------------------	----------------------	-----	--------------	--------------	------------------------	-------------------------	------------------------	-------------------------	-------------------------	--------	--------------------	---------------------

FIGURE 5.11: Packets of output string [16].

As the LIDAR sends a continuous stream of data, there is also a special command to stop the continuous data.

LMS continuous data output	Telegram code in hex PC → LMS	Reply telegram in hex LMS → PC
Stop	02 00 02 00 20 25 35 08	06 02 81 03 00 A0 00 10 36 1A

FIGURE 5.12: Command to stop the continuous stream of data [16].

The main task of the RX62N microcontroller is to save the distance measurement data along with the header and an ID which differentiates LIDAR data from the other data. The ID to be set for the LIDAR data can be selected by the user and the selection of ID for the data from the sensors will be discussed in further chapters.

### 5.3.2 Requesting Data from GPS

The Garmin GPS 25HVS used on the ATV is configured to the default settings of NMEA Version 2.0 ASCII output, 4800 baud rate and an output frequency of 1Hz. It outputs data on a RS232 serial port. For polling of data on the GPS, the NMEA 0183 standard provides an Output Sentence Enable/Disable. The format of the sentence is as follows

\$PGRMO,<1>,<2>\*hh<CR><LF>

Where <1> Target sentence description (e.g., PGRMT, GPGSV, etc.)

<2> Target sentence mode, where:

0 = disable specified sentence

1 = enable specified sentence

2 = disable all output sentences

3 = enable all output sentences (except GPALM)

This sentence can be used to request any required NMEA sentence from the GPS. The requested sentence will be transmitted every second in ASCII format. For example to enable transmission of all the output sentences the format would be

\$PGRMO,,3<CR><LF> and to stop the continuous transmission of sentences the format of the sentence would be \$PGRMO,,2<CR><LF>.

### 5.3.3 Requesting Data from IMU

The IMU used on the ATV is a 9 degree of freedom Razor IMU which has four sensors.

- Single axis gyro
- Dual axis gyro
- Triple axis accelerometer
- Triple axis magnetometer

The IMU contains an onboard ATmega328 which collects the data from the sensors and outputs the data on a RS232 serial port. The IMU is configured to output data at a rate of 19200 baud. The ATmega328 triggers the continuous transmission of data at the reception of “Ctrl + z” command.

## CHAPTER 6: PROTOTYPE DESIGN

As the modules (LIDAR, GPS, IMU and H-Bridge) being implemented on the ATV are RS232 based devices, a communication bridge between RS232 and CAN would make interfacing of new devices on to the CAN bus easier. Software has been developed in a way that the end user can just plug any device which communicates over RS232 and get the data over CAN. This chapter would discuss about the hardware and software implementation of RS232-CAN bridge module using Renesas RX62N.

### 6.1 Hardware implementation of RS232-CAN Communication Bridge

Each CAN node on the bus is given an ID which will be used by the CAN protocol as a standard ID for message transfer. An 8 bit DIP switch has been provided through which ID to the CAN node can be selected. Among the 8 bits, the first 3 MSB bits are used to select the baud rate of the RS232 protocol and the rest of the bits are used to select the ID. The node with DIP switch value of 0xFF would be considered as a base node and communicates with the PC at 115200 baud rate. The TABLE 6.1 lists the baud rate values for the corresponding values of the DIP switch.

TABLE 6.1: Baud rate selection with DIP switch.

DIP switch value	Baud rate
001	2400
010	4800
011	9600
100	19200
101	38400
110	57600
111	115200

Apart from the 3 bits of DIP switch for the RS232 baud rate selection, 5 bits are left out for the ID selection but only 4 bits are being used. Consider a case where a IMU is transferring data from the IMU node to the base station, if the base station wants to stop the transmission of data it needs to send a command on to the CAN bus. If the transmit ID of the IMU node has a higher priority (lesser value) than the receive ID of the IMU node, then the higher priority data is transmitted on to the bus and the command from the base station will not have access to the bus till the transmission of data stops due to which the continuous transmission of data never stops. To avoid this dead lock situation, the 5<sup>th</sup> bit of the DIP switch is used to assign lower priority for the RS 232 data being transmitted and higher priority to the commands from the base station. As only 4



bits are being utilized for the ID, maximum of 16 devices can be connected on a single CAN bus. If more devices are to be interfaced, the baud rate of all the RS232 devices can be kept same, so that the 3 bits used for the selection of the baud rate can be utilized to select the ID by which the number of devices that can be connected to the bus increases to 128 ( $2^7$ ).

## 6.2 Software Implementation of RS232-CAN Communication Bridge

The softwares used for the development of this application are High-performance Embedded Workshop (HEW) and Netbeans. HEW is a GUI based development environment used for debugging and development of embedded applications for Renesas microcontrollers. It is a powerful and an easy to use interface which has C/C++ compilers and debugger elements for all the available Renesas evaluation boards. In this application, HEW is used to compile, debug and program the firmware for Renesas RX62N.

Netbeans is an Integrated Development Environment (IDE) used for developing Java, JavaScript, PHP, Python, Ruby, Groovy, C and C++ based applications. In this application it is used to develop a Java based Graphical User Interface (GUI) for serial communication with Renesas RX62N microcontroller.

The firmware for RS232-CAN bridge module on Renesas RX62N microcontroller is developed in C language. It collects the data from serial RS232 bus and transmit it on to the CAN bus. Two FIFO queues have been created to store the data received from the RS232 and CAN protocols. Receive interrupts have been set up for both the protocols, where the ISR saves the data in the corresponding queue. The **FIGURE 6.1: RS232-CAN**

communication bridge execution flow. FIGURE 6.1 shows the execution flow of the program implemented.

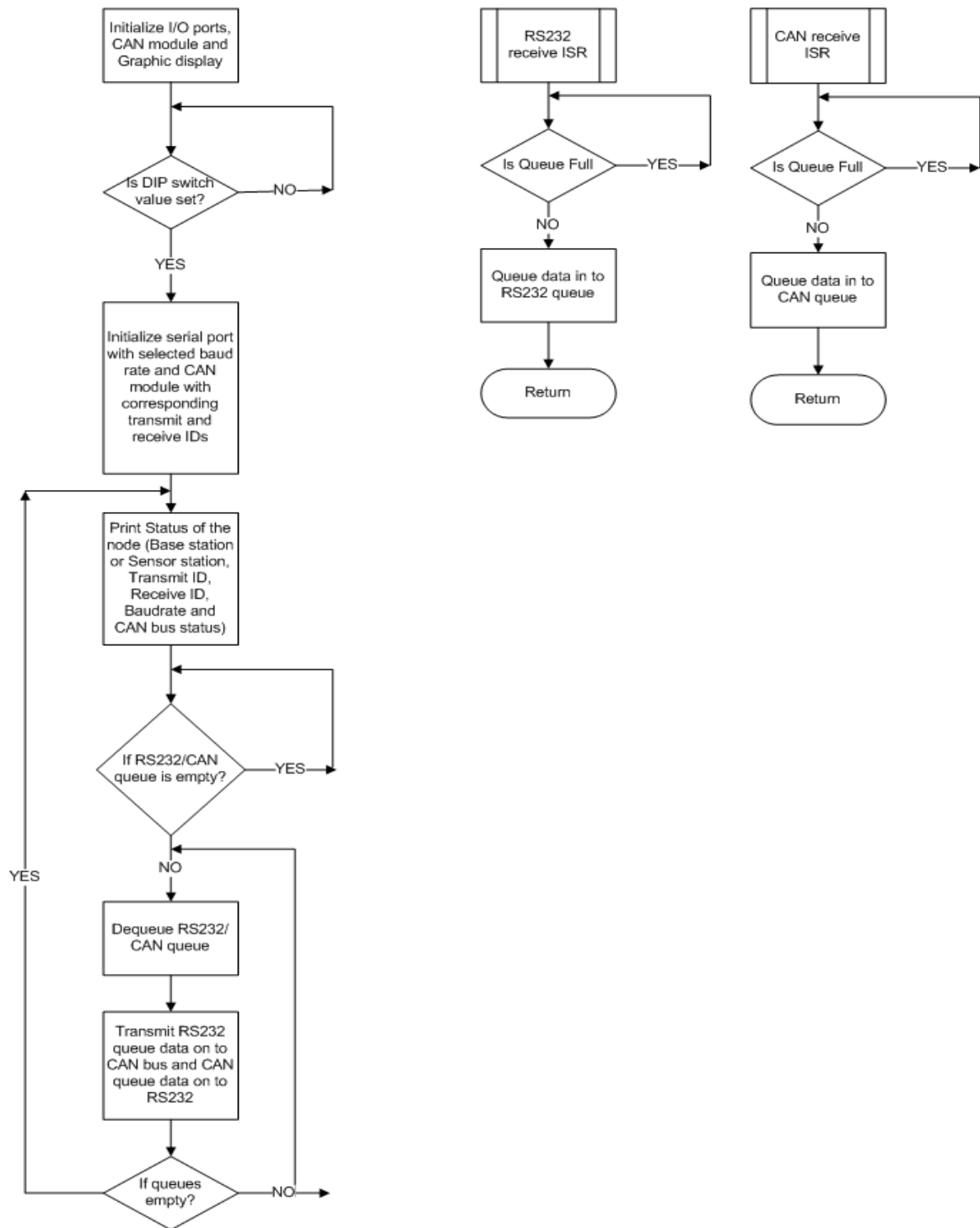


FIGURE 6.1: RS232-CAN communication bridge execution flow.

The Java based GUI created using Netbeans is used to trigger data from the sensor nodes. A serial port with a baud rate of 115200 has been configured to communicate with Renesas RX62N microcontroller. The data received on the serial port is saved in an array. The data in the array starts with a one byte header which represents data from a particular sensor node. For every byte of data on the CAN bus, two bytes of data is sent to the PC with a header as an extra byte which is used to differentiate data from different sensor nodes. Using the header, data is sorted and written in to corresponding text files. For every new header value, a new text file will be created and the data related to this header will be logged in to the corresponding text file. The working of the RS232-CAN module was tested with three nodes (LIDAR, IMU and GPS) connected to the CAN bus. The data received in the text files was consistent without any errors even though all the three nodes were transmitting data at the same time.

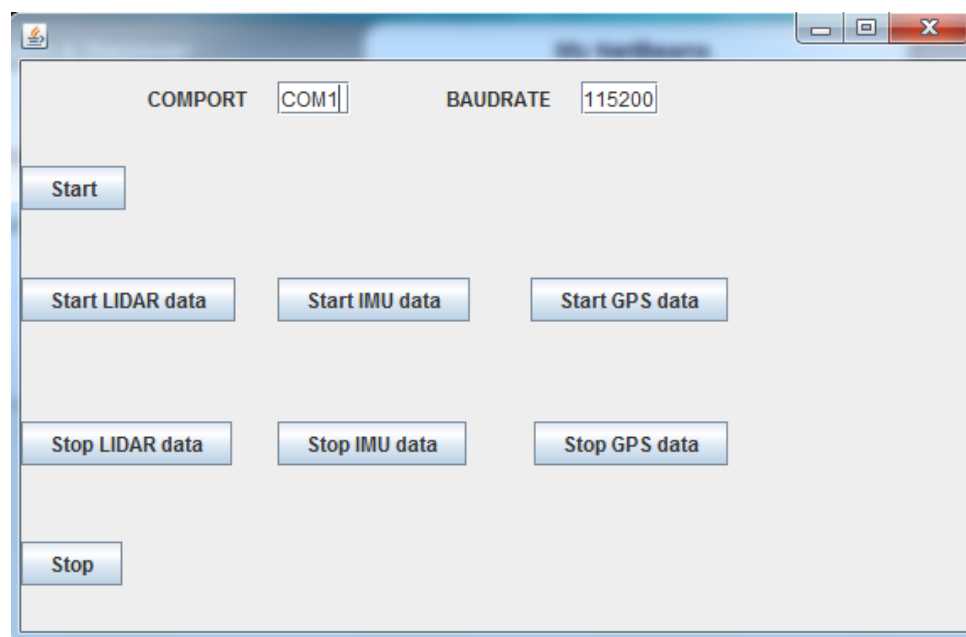


FIGURE 6.2: GUI for RS232-CAN module.

The GUI for RS232-CAN module shows the GUI designed for initializing the data communication with the PC and also for triggering continuous data transmission of the sensors. The COMPORT text field and the BAUDRATE text field are used to select comport of the PC with required baud rate. The START button starts the serial communication between the base station and the PC. The STOP button closes the serial port and deletes the text files created. The other buttons are used to trigger the continuous data transmission of the sensors available.

### 6.3 Final Prototype

The final prototype of the control system for the ATV is as shown in the figure below.

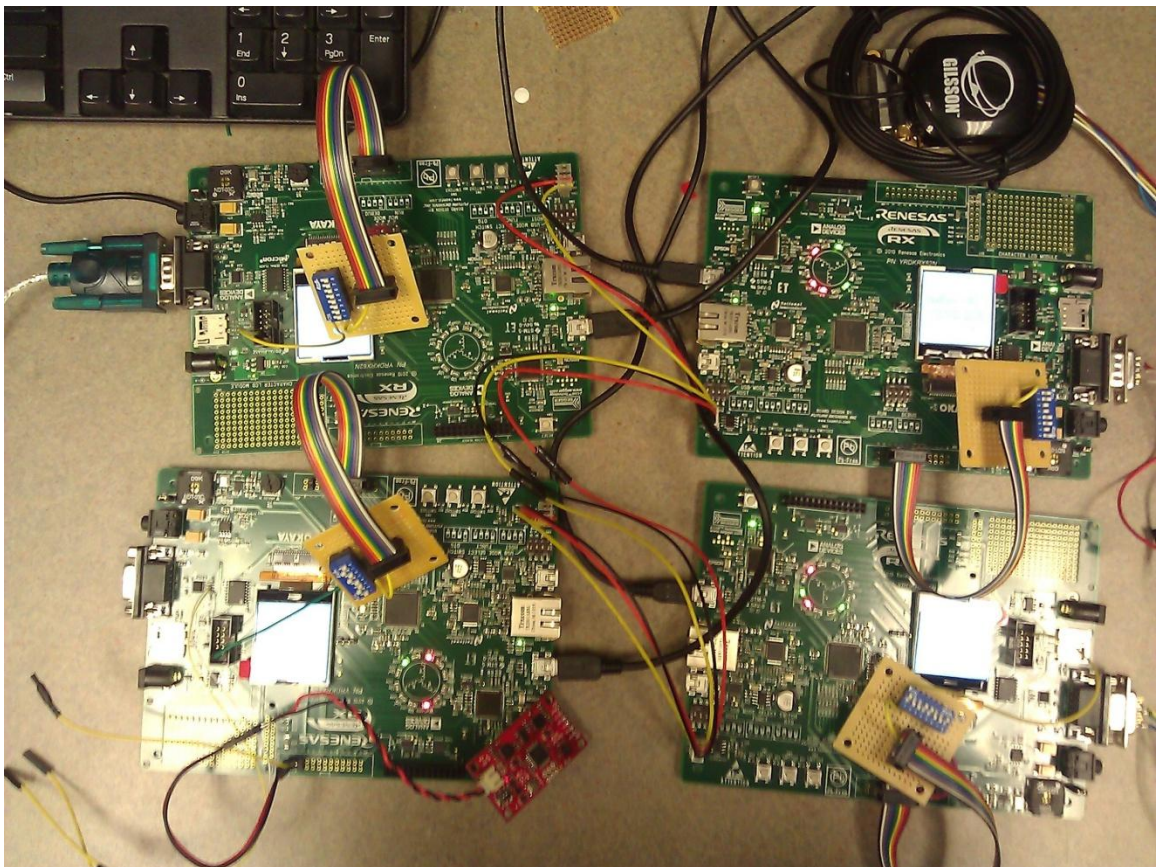


FIGURE 6.3: Prototype of the control system for the ATV.

The FIGURE 6.3 shows the four CAN nodes with DIP switches. The yellow, red and black wires form the CAN bus connecting the four nodes.

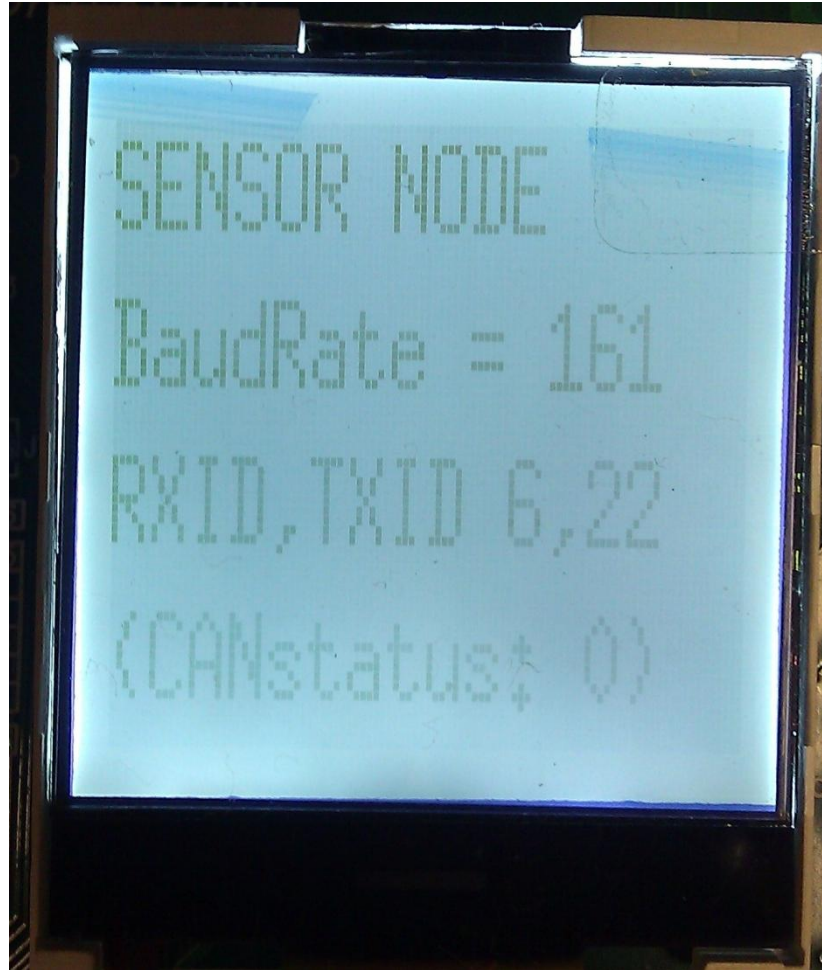


FIGURE 6.4: Graphic display on RX62N.

The FIGURE 6.4 shows the messages printed on the graphic display while the firmware is running. The first line displays whether the node acts as a sensor node or a base station. The second line shows the value put into the baud rate generator register of RX62N for calculating the baud rate for RS232 serial protocol. The third line shows the transmit and receive ID's of the node. The fourth line shows the value of the CAN status register in RX62N which depicts the status of CAN bus.



## 6.4 Data received over CAN bus

The following are the screenshots of the text files that show the data of LIDAR, GPS and IMU received over CAN bus.

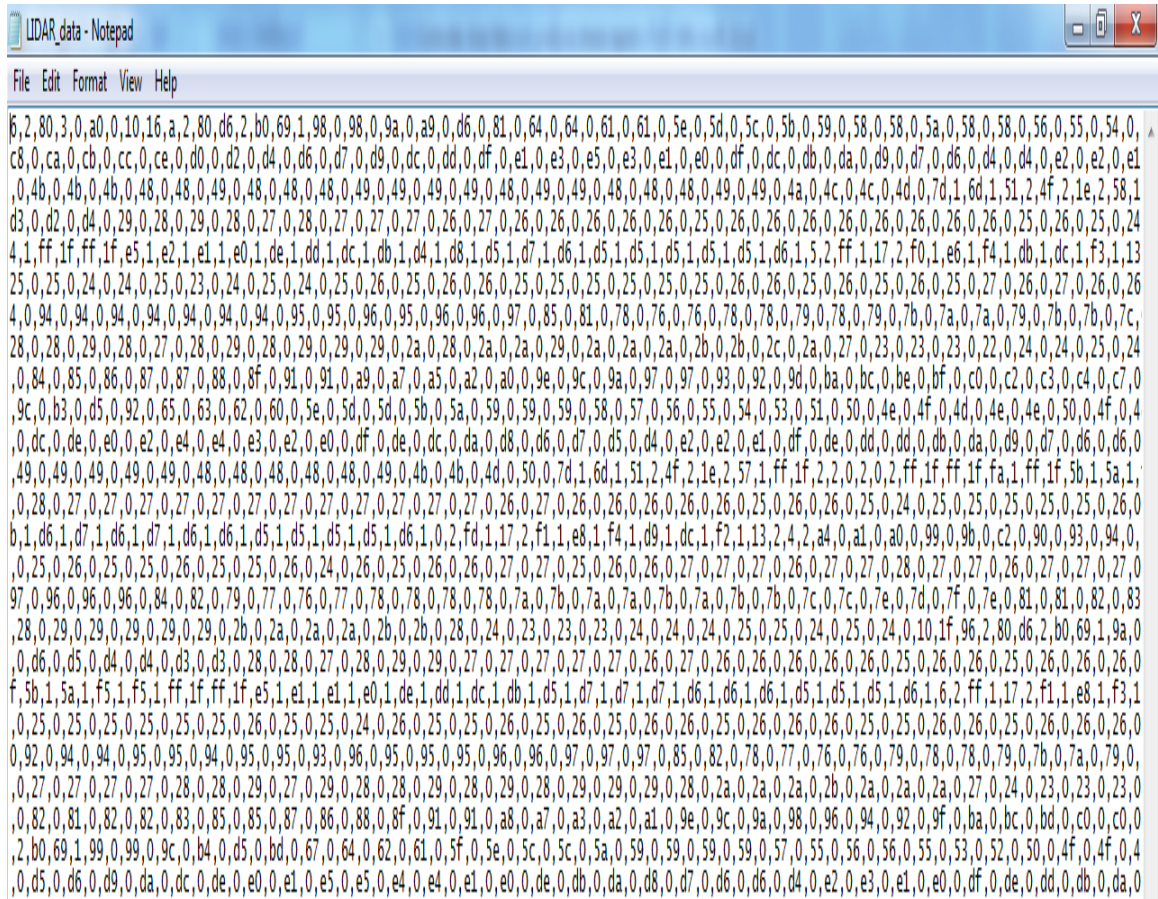
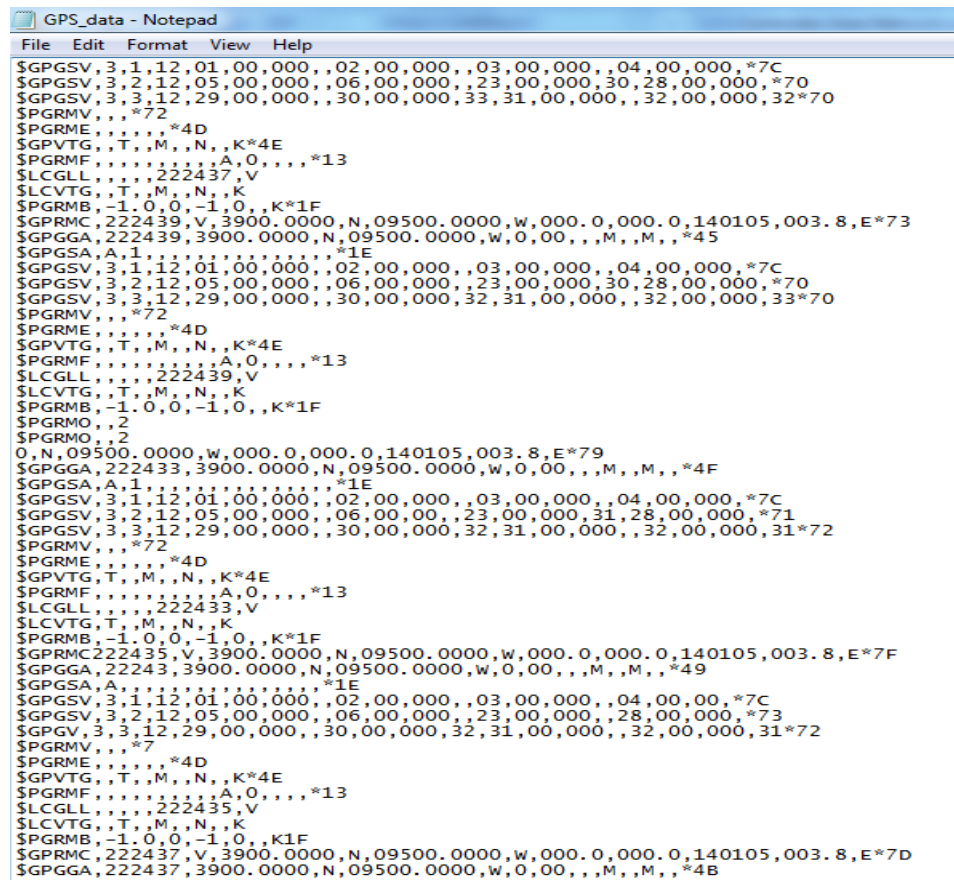


FIGURE 6.5: LIDAR data received over CAN bus.



```

GPS_data - Notepad
File Edit Format View Help
$GPGSV,3,1,12,01,00,000,,02,00,000,,03,00,000,,04,00,000,*7C
$GPGSV,3,2,12,05,00,000,,06,00,000,,23,00,000,,30,28,00,000,*70
$GPGSV,3,3,12,29,00,000,,30,00,000,,33,31,00,000,,32,00,000,32*70
$PGRMV,,,*72
$PGRME,,,,*4D
$GPVTG,,T,M,N,K*4E
$PGRMF,,,,,A,0,,,*,13
$LCGLL,,,222437,V
$LCVTG,,T,M,N,K
$PGRMB,-1.0,0,-1.0,,K*1F
$GPRMC,222439,V,3900.0000,N,09500.0000,W,0.0,0.000,0.140105,003.8,E*73
$GPGGA,222439,3900.0000,N,09500.0000,W,0.00,,M,M,,*45
$GPGSA,A,1,,,,,,*1E
$GPGSV,3,1,12,01,00,000,,02,00,000,,03,00,000,,04,00,000,*7C
$GPGSV,3,2,12,05,00,000,,06,00,000,,23,00,000,,30,28,00,000,*70
$GPGSV,3,3,12,29,00,000,,30,00,000,,32,31,00,000,,32,00,000,33*70
$PGRMV,,,*72
$PGRME,,,,*4D
$GPVTG,,T,M,N,K*4E
$PGRMF,,,,,A,0,,,*,13
$LCGLL,,,222439,V
$LCVTG,,T,M,N,K
$PGRMB,-1.0,0,-1.0,,K*1F
$PGRMO,,2
$PGRMO,,2
0,N,09500.0000,W,0.0,0.000,0.140105,003.8,E*79
$GPGGA,222433,3900.0000,N,09500.0000,W,0.00,,M,M,,*4F
$GPGSA,A,1,,,,,,*1E
$GPGSV,3,1,12,01,00,000,,02,00,000,,03,00,000,,04,00,000,*7C
$GPGSV,3,2,12,05,00,000,,06,00,000,,23,00,000,,31,28,00,000,*71
$GPGSV,3,3,12,29,00,000,,30,00,000,,32,31,00,000,,32,00,000,31*72
$PGRMV,,,*72
$PGRME,,,,*4D
$GPVTG,,T,M,N,K*4E
$PGRMF,,,,,A,0,,,*,13
$LCGLL,,,222433,V
$LCVTG,,T,M,N,K
$PGRMB,-1.0,0,-1.0,,K*1F
$GPRMC,222435,V,3900.0000,N,09500.0000,W,0.0,0.000,0.140105,003.8,E*7F
$GPGGA,22243,3900.0000,N,09500.0000,W,0.00,,M,M,,*49
$GPGSA,A,1,,,,,,*1E
$GPGSV,3,1,12,01,00,000,,02,00,000,,03,00,000,,04,00,000,*7C
$GPGSV,3,2,12,05,00,000,,06,00,000,,23,00,000,,28,00,000,*73
$GPGSV,3,3,12,29,00,000,,30,00,000,,32,31,00,000,,32,00,000,31*72
$PGRMV,,,*7
$PGRME,,,,*4D
$GPVTG,,T,M,N,K*4E
$PGRMF,,,,,A,0,,,*,13
$LCGLL,,,222435,V
$LCVTG,,T,M,N,K
$PGRMB,-1.0,0,-1.0,,K*1F
$GPRMC,222437,V,3900.0000,N,09500.0000,W,0.0,0.000,0.140105,003.8,E*7D
$GPGGA,222437,3900.0000,N,09500.0000,W,0.00,,M,M,,*4B

```

FIGURE 6.6: GPS data received over CAN bus.

The FIGURE 6.6 shows the GPS data received over CAN bus. As the GPS was tested in a closed area, it could not receive any satellite signals so most of the GPS data is zeros or default values.

IMU\_data - Notepad

File	Edit	Format	View	Help						
\$	,-4	,-198	,-194	,376	,381	,368	,337	,698	,408	,#
\$	,-5	,-197	,-195	,375	,381	,367	,337	,698	,408	,#
\$	,-5	,-201	,-195	,374	,381	,367	,337	,698	,408	,#
\$	,-3	,-198	,-195	,376	,381	,369	,337	,698	,408	,#
\$	,-5	,-199	,-194	,376	,382	,369	,337	,698	,408	,#
\$	,-5	,-198	,-193	,375	,381	,368	,337	,698	,408	,#
\$	,-5	,-198	,-195	,376	,381	,369	,337	,701	,409	,#
\$	,-3	,-200	,-193	,376	,381	,369	,337	,701	,409	,#
\$	,-4	,-201	,-194	,375	,381	,367	,337	,701	,409	,#
\$	,-3	,-201	,-194	,375	,381	,368	,337	,701	,409	,#
\$	,-2	,-200	,-194	,376	,380	,368	,337	,701	,409	,#
\$	,-4	,-199	,-193	,375	,380	,367	,337	,701	,409	,#
\$	,-5	,-199	,-193	,375	,381	,368	,339	,702	,402	,#
\$	,-4	,-201	,-195	,376	,381	,368	,339	,702	,402	,#
\$	,-3	,-199	,-194	,376	,382	,368	,339	,702	,402	,#
\$	,-5	,-199	,-193	,375	,381	,368	,339	,702	,402	,#
\$	,-5	,-201	,-194	,376	,381	,369	,339	,702	,402	,#
\$	,-5	,-199	,-193	,375	,382	,368	,341	,705	,408	,#
\$	,-4	,-200	,-192	,375	,381	,367	,341	,705	,408	,#
\$	,-4	,-200	,-195	,376	,382	,368	,341	,705	,408	,#
\$	,-4	,-199	,-195	,375	,381	,368	,341	,705	,408	,#
\$	,-6	,-199	,-193	,375	,381	,367	,341	,705	,408	,#
\$	,-6	,-200	,-195	,375	,381	,368	,341	,705	,408	,#
\$	,-4	,-199	,-195	,376	,381	,367	,342	,704	,411	,#
\$	,-4	,-199	,-195	,375	,381	,368	,342	,704	,411	,#
\$	,-4	,-199	,-193	,375	,381	,368	,342	,704	,411	,#
\$	,-4	,-200	,-193	,376	,381	,369	,342	,704	,411	,#
\$	,-5	,-199	,-194	,376	,381	,368	,342	,704	,411	,#
\$	,-5	,-198	,-194	,375	,381	,368	,342	,704	,411	,#
\$	,-4	,-201	,-193	,376	,381	,368	,341	,701	,398	,#
\$	,-3	,-200	,-194	,375	,381	,368	,341	,701	,398	,#
\$	,-5	,-198	,-193	,375	,381	,368	,341	,701	,398	,#
\$	,-4	,-199	,-195	,376	,381	,368	,341	,701	,398	,#
\$	,-4	,-200	,-196	,376	,381	,368	,341	,701	,398	,#
\$	,-4	,-198	,-195	,375	,381	,368	,341	,701	,398	,#
\$	,-4	,-197	,-194	,376	,381	,367	,341	,697	,411	,#
\$	,-4	,-201	,-195	,376	,381	,368	,341	,697	,411	,#
\$	,-5	,-201	,-194	,376	,382	,368	,341	,697	,411	,#
\$	,-5	,-197	,-193	,375	,381	,368	,341	,697	,411	,#
\$	,-4	,-202	,-195	,375	,382	,368	,341	,697	,411	,#
\$	,-3	,-198	,-193	,376	,380	,368	,341	,697	,411	,#
\$	,-4	,-201	,-195	,375	,381	,368	,339	,699	,409	,#
\$	,-5	,-198	,-194	,375	,381	,368	,339	,699	,409	,#
\$	,-4	,-201	,-195	,375	,382	,368	,339	,699	,409	,#
\$	,-4	,-199	,-194	,375	,381	,368	,339	,699	,409	,#
\$	,-4	,-200	,-195	,375	,381	,368	,339	,699	,409	,#
\$	,-3	,-199	,-192	,376	,380	,369	,343	,701	,410	,#
\$	,-5	,-200	,-194	,375	,381	,368	,343	,701	,410	,#
\$	,-5	,-199	,-193	,375	,381	,368	,343	,701	,410	,#
\$	,-4	,-200	,-194	,375	,381	,368	,343	,701	,410	,#
\$	,-3	,-198	,-193	,375	,381	,368	,343	,701	,410	,#
\$	,-4	,-200	,-195	,375	,381	,368	,343	,701	,410	,#
\$	,-4	,-200	,-194	,376	,380	,368	,343	,703	,403	,#

FIGURE 6.7: IMU data received over CAN bus.

The FIGURE 6.7 shows the IMU data received over CAN bus. The first three values of the IMU data represents the orientation of the device with respect to three axes, next three represents the values of acceleration with respect to three axes and the last three represent the value for the strength of magnetic field with respect to three axes.



## CHAPTER 7: CONCLUSION AND FUTURE SCOPE

### 7.1 Conclusion

This thesis documented a design of RS232-CAN module which forms a communication bridge between the RS232 and CAN protocols. The module provides a protocol conversion with a plug and play feature where modification of neither the hardware nor the software is needed. The module is mainly designed to implement CAN bus on the Honda Four Trax ATV.

An autonomous vehicle is built with the ATV and the CAN protocol is used as the vehicle network in the ATV. CAN is a reliable, robust and real time serial communication protocol which reduces wiring harness, weight and complexity. The protocol creates a master to master communication bus and every message transmitted on the bus is received by all the nodes connected to the bus. The message filtering technique decides whether the received data is relevant to the node or not. The error detection and fault confinement techniques provided by the protocol are the added features which keep the bus working without any errors and virtually detach the faulty nodes which transmit corrupt messages on the bus.

A prototype of CAN control system on the ATV has been implemented with four nodes where three nodes are connected to a LIDAR, GPS and IMU each and a base station as the fourth node which is connected to the PC. A Java based GUI application is

designed to communicate with the base station through a serial port and also to trigger the continuous transmission of data from the sensors. The rate at which the base station communicates with the PC should be equal to or lesser than the sum of rates at which the sensors communicate over RS232 to avoid the overflow of the data. The data which is received on the PC serial port is sorted out and written in three different text files where each file contains the data from the LIDAR, GPS and IMU respectively.

## 7.2 Future Scope

A camera module is in implementation process, as the CAN bus is not capable of communicating such high speed data, other higher data rate vehicle networks like FlexRay can be implemented. A Local Interconnect Network (LIN) bus can also be implemented as a sub network for the CAN bus which is designed mainly for simple switching applications. It can be used for triggering the continuous data transmission of the devices which releases some bandwidth on the CAN bus and also increases the number of devices that can be interfaced to the CAN bus.

The present RX62N evaluation board occupies a lot of space when mounted on the ATV, so a smaller size evaluation board would make the design compact. The design can be made more compact by incorporating a smaller size microcontroller, CAN controller and a RS232 module. The firmware designed for this thesis can be implemented in the compact design and a low cost RS232-CAN converter will be available for future researchers.

## BIBLIOGRAPHY

- [1] A. Lakhota, P. Edgington, S. Golconda, A. Maida, P. Meija and G. Seetharamam, "CajunBot-II: An Autonomous Vehicle for The DARPA Urban Challenge," , 2007.
- [2] B. H. Kim, D. K. Roh, Jang M. Lee, M. H. Lee, K. Son, M. C. Lee, J. W. Choi and S.H. Han, "Localization of a Mobile Robot using Images of a Moving Target," in *International Conference on Robotics and Automation*, Korea, 2001, pp. 253-258.
- [3] B. You, M. Hwangbo, S. Lee, Sang-Rok Oh, Y. Kwon and San Lim, "Development of a Home Service Robot 'ISAAC'," in *International Conference on Intelligent Robots and Systems*, Las Vegas Nevada, 2003, pp. 2630-2635.
- [4] *Hardware Manual, Renesas 32-Bit Microcomputer, RX Family/RX600 Series.*: Renesas Electronics America, Inc., 2010.
- [5] J.A. Gil, A. Pont, G. Benet, F.J. Blanes and M. Martinez, "A CAN Architecture For An Intelligent Mobile Robot," in *Proceedings of the Third IFAC Symposium in Intelligent Components and Instruments for Control Applications (SICICA '97)*, 1997, pp. 65-70.
- [6] J.W. Hofstee and D. Goense, "Simulation of a Controller Area Network-based Tractor," in *Journal of Agricultural Engineering Research*, 1999, pp. 383-394.
- [7] K. M. Zuberi and K. G. Shin, "Scheduling Messages on Controller Area Network for Real- Time CIM Applications," in *IEEE Transactions on Robotics and Automation*, 1997, pp. 310-316.
- [8] K. Tindell, A.Burns and A.J. Wellings, "Calculating Controller Area Network (CAN) Message Response Times," in *Control Engineering Practice, Volume 3, Issue 8*, UK, 1995, pp. 1163-1169.
- [9] M. Mock and E. Nett, "Real-Time Communication in Autonomous Robot Systems," in *Autonomous Decentralized Systems, 1999.*, 1999, pp. 34-41.
- [10] M. Wargui and Rachid, A., "Application of Controller Area Network to Mobile Robots," in *Electrotechnical Conference, 8th Mediterranean*, Italy, 1996, pp. 205-207.
- [11] R.A. Mckinney, "Components of an Autonomous ATV," University of North

Carolina, Charlotte, Masters Project 2009.

- [12] P. Pedreiras and L. Almeida, "EDF Message Scheduling on Controller Area Network," in *Computing and Control Engineering Journal*, 2002, pp. 163-170.
- [13] R.A.McKinney, M.J.Zapata, J.M.Conrad, T.W.Meiswinkel and S.Ahuja, "Components of an autonomous all-terrain vehicle," in *IEEE SoutheastCon 2010 (SoutheastCon)*, 2010, pp. 416-419.
- [14] S. Krywult, "Real Time Communication Systems for Small Autonomous Robots," in *Thesis report*, 2006.
- [15] S. SZABO and V. Oplustil, "Distributed CAN Based Control System for Robotic and Airborne Applications," in *Seventh International Conference on Control, Automation, Robotics And Vision(ICARCV '02)*, Singapore, 2002, pp. 1233-1238.
- [16] SICK AG. (2002, March) Drexel University. [Online].  
[http://www.pages.drexel.edu/~kws23/tutorials/sick/LMS\\_Quick\\_Manual\\_V1\\_1.pdf](http://www.pages.drexel.edu/~kws23/tutorials/sick/LMS_Quick_Manual_V1_1.pdf)
- [17] Sparkfun. (2010) Sparkfun Electronics. [Online].  
<http://www.sparkfun.com/products/9623>
- [18] (1997) The OSU Autonomous Vehicle. [Online]. <http://www2.ece.ohio-state.edu/citr/Demo97/osu-av.html>
- [19] V. K. Kongezos and C. R.Allen, "Wireless communication between A.G.V 's (Autonomous Guided Vehicle) and the industrial network CAN," in *International Conference on Robotics and Automation*, Washington,DC, 2002, pp. 434-437.
- [20] M.J. Zapata, "Obstacle Detection and Avoidance for an Autonomous All Terrain Vehicle," University of North Carolina, Charlotte, Masters Project 2009.

## APPENDIX A: CODE

```

/*****

/* FILE: YRDKRX62N_Glyph_Demo.c */

/* DATE :Wed, Jul 21, 2010 */

/* DESCRIPTION :Main Program */

/* CPU TYPE :RX610 */

*****/

/* Code written by Sunil Kumar Gurram for the thesis “Implementation of Controller Area
Network (CAN) bus in an autonomous all-terrain vehicle (ATV) */

#include "iodefine_RX62N.h"

#include "RX62N_LED_defs.h"

#include "UART_Queues_def.h"

#include "LMS.h"

*****/

Includes <system includes> "Glyph Includes"

*****/

#include <stdlib.h>

#include <stdio.h>

#include "src\Glyph\Glyph.h"

#include "CAN_defines.h"
```

```
/******
```

Declaration of global and external variables

```
*****/
```

```
int i,j,k = 0,p,temp = 0,data_transfer;
```

```
unsigned int dist_data[500],raw_data[500];
```

```
extern Q_T RS232_rx,CAN_rx;
```

```
extern int data_ready,rx_header,x,cont_datacmd,header,RXID,sid_acquired;
```

```
char buffer1[30], buffer2[], buffer3[],buffer4[],c;
```

```
unsigned char data=0,rxdata,f_data;
```

```
unsigned int data1 = 0,RDATA = 0, SDATA = 0,sid,rx_sid,tx_sid,sid_cleared =
```

```
0,data_size,baud_sel,baudrate;
```

```
/******
```

Private global variables and functions

```
*****/
```

```
T_glyphHandle G_lcd ;
```

```
static int G_nContrastValue = 105 ;
```

```
static int G_nContrastBoostValue = 5 ;
```

```
static void TextWrite( char * msg_string);
```

```
static void DrawFDI(T_glyphHandle aHandle);
```

```
void Init_port();
```

```
void BSP_Display_String(int8_t aLine, char * aText);
```

```
#define LCD_LINE1    0

#define LCD_LINE2    1

#define LCD_LINE3    2

#define LCD_LINE4    3
```

```
/******
```

```
* ID : 100.0
```

```
* Function Name: DrawFDI
```

```
* Description : draw the words FUTURE at (10, 10) DESIGNS at (20, 20)
```

```
* and INC at (30, 30)
```

```
* Argument : aHandle the Glyph Handle to this instance.
```

```
* Return Value : none
```

```
* Calling Functions : main
```

```
*****/
```

```
static void DrawFDI(T_glyphHandle aHandle)
```

```
{
```

```
    char buffer[30];
```

```
    BSP_Display_String(LCD_LINE1, "FUTURE");
```

```
    BSP_Display_String(LCD_LINE2, " DESIGNS");
```

```
    BSP_Display_String(LCD_LINE3, " INC");
```

```
    sprintf((char *)buffer, "(C0STR : %d)",  CAN0.C0STR.WORD);
```

```
    BSP_Display_String(LCD_LINE4, buffer);
```

```
    TextWrite(buffer);
```

```

    TextWrite("\r\n");
}

```

```

/*****

```

\* Function Name: main

\* Description : This main function is designed as a firmware for a CAN-RS232 node. This function waits for the Switch 1 press and after the Switch 1 is pressed the node acts as Base station or a Node station based upon the DIP switch value.

If the DIP switch value is 0xFF it acts as a Base station and for the other values

- The first 4 LSBs are used to select the receive ID for the CAN protocol and the transmit ID for the CAN protocol is 16 (as the First MSB of the DIP switch will always be 1) + the value set up with the first 4 LSB of the 8-bit DIP switch
- The bits from Bit4-Bit6 are used to select the baud rate for the RS232 protocol.

DIP switch value	Baud rate
001	2400
010	4800
011	9600
100	19200
101	38400
110	57600
111	115200

\* Argument : none



\* Return Value : none

\* Calling Functions : none

\*\*\*\*\*/

```
void main(void)
```

```
{
```

```
    ENABLE_LEDS;
```

```
    ALL_LEDS_OFF;
```

```
        Init_CAN();
```

```
        Init_port();
```

```
        Q_Init(&RS232_rx);
```

```
        Q_Init(&CAN_rx);
```

```
    if (GlyphOpen(&G_lcd, 0) == GLYPH_ERROR_NONE) {
```

```
        /* use Glyph full access direct functions */
```

```
        GlyphNormalScreen(G_lcd) ;
```

```
        GlyphSetFont(G_lcd, GLYPH_FONT_6_BY_13) ;
```

```
        GlyphClearScreen(G_lcd) ;
```

```
        //DrawFDI(G_lcd) ;
```

```
    while(1){
```

```
        sprintf((char *)buffer4, "(I/O val: %X)",PORTD.PORT.BYTE); // Displays the DIP switch value
```

```
        BSP_Display_String(LCD_LINE4, buffer4);
```

```
    if(!S1){
```

```

switch(PORTD.PORT.BYTE){

    case 0xFF: // Base station

        InitSCI(PRESCALER_115200,BRG_115200);

        for(i=RX_MBOX_STRT; i<RX_MBOX_END;i++){

            CAN_Rx(i,0,0x0000);

        }

        while(1)

    {

        if(!Q_Empty(&RS232_rx)){ //Checking for RS232 received data in
RS232 queue

            sid = Q_Dequeue(&RS232_rx);

            data_size = Q_ReturnSize(&RS232_rx);

            //Transfers all the data from the RS232 queue on to the CAN bus

            for(i=0;i< data_size;i++){

                CAN0.C0MCTL[TX_MBOX_ID].BIT.TX.SENTDATA = 0;

                CAN0.C0MCTL[TX_MBOX_ID].BIT.TX.TRMREQ = 0;

                while((((CAN0.C0MCTL[TX_MBOX_ID].BIT.TX.SENTDATA)&&(CAN0.C0MCTL[TX_MB
OX_ID].BIT.TX.TRMREQ)));

                CAN_Tx(TX_MBOX_ID, Q_Dequeue(&RS232_rx),sid);

                DisplayDelay(10);

            }

        }

    }

```

```
//Checks for the CAN received data in the CAN queue and transmits data on to RS232
```

```
    if(!Q_Empty(&CAN_rx)){  
        transmit_data(Q_Dequeue(&CAN_rx));  
    }
```

```
//Just in case if CAN receive queue gets full, it initializes the queue
```

```
    if(Q_Full(&CAN_rx)){  
        Q_Start(&CAN_rx);  
    }
```

```
//Displays the status of the node
```

```
    sprintf((char *)buffer1, "BASE NODE");  
    BSP_Display_String(LCD_LINE1, buffer1);
```

```
    sprintf((char *)buffer2, "Baud Rate : 115200");  
    BSP_Display_String(LCD_LINE2, buffer2);
```

```
    sprintf((char *)buffer4, "(CAN Status : %X)",CAN0.C0STR.WORD);  
    BSP_Display_String(LCD_LINE4, buffer4);
```

```
}
```

```
    break;
```

```
default: //Node station
```

```
    //Selecting the baud rate selector value from the Bit4-Bit6
```

```

        baud_sel = 0xE0 & PORTD.PORT.BYTE;

        baudrate = (baud_sel == 0x20) ? InitSCI(PRESCALER_2400,BRG_2400)
: (baud_sel == 0x40) ? InitSCI(PRESCALER_4800,BRG_4800) : (baud_sel == 0x60) ?
InitSCI(PRESCALER_9600,BRG_9600) : (baud_sel == 0x80) ?
InitSCI(PRESCALER_19200,BRG_19200) : (baud_sel == 0xA0) ?
InitSCI(PRESCALER_38400,BRG_38400) : (baud_sel == 0xC0) ?
InitSCI(PRESCALER_57600,BRG_57600) : (baud_sel == 0xE0) ?
InitSCI(PRESCALER_115200,BRG_115200) : 0;

        //Sets the transmit and receive ID for the CAN protocol

        tx_sid = (0x10 | (0x0F & PORTD.PORT.BYTE));

        rx_sid = 0x0F & PORTD.PORT.BYTE;

        //Initializes the receive mailbox with SID filter enabled

        for(i=RX_MBOX_STRT; i<RX_MBOX_END;i++){

                CAN_Rx(i,rx_sid,0x7FF);

        }

        while(1)

        {

                if(!Q_Empty(&RS232_rx)){ //Checking for RS232 received data in
RS232 queue

                        //Transfers all the data from the RS232 queue on to the CAN bus

                        for(i=0;i< Q_ReturnSize(&RS232_rx);i++){

                                CAN0.COMCTL[TX_MBOX_ID].BIT.TX.SENTDATA = 0;

```

```

    CAN0.C0MCTL[TX_MBOX_ID].BIT.TX.TRMREQ = 0;

    while((((CAN0.C0MCTL[TX_MBOX_ID].BIT.TX.SENTDATA)&&(CAN0.C0MCTL[
TX_MBOX_ID].BIT.TX.TRMREQ)))));

    CAN_Tx(TX_MBOX_ID, Q_Dequeue(&RS232_rx),tx_sid);

    DisplayDelay(10);
}

    }

//Checks for the CAN received data and transmits the data on to RS232
    if(!Q_Empty(&CAN_rx)){

        rxdata = Q_Dequeue(&CAN_rx);

        transmit_data(Q_Dequeue(&CAN_rx));

    }

//Just in case if RS232 receive queue gets full, it initializes the queue
    if(Q_Full(&RS232_rx)){

        Q_Start(&RS232_rx);

    }

//Displays the status of the node
    sprintf((char *)buffer1, "SENSOR NODE");

    BSP_Display_String(LCD_LINE1, buffer1);


    sprintf((char *)buffer2, "BaudRate = %d",baudrate);

    BSP_Display_String(LCD_LINE2, buffer2);

```

```

        sprintf((char *)buffer3, "RXID = %d",rx_sid);

        BSP_Display_String(LCD_LINE3, buffer3);


        sprintf((char *)buffer4, "(CANstatus: %X)",CAN0.C0STR.WORD);

        BSP_Display_String(LCD_LINE4, buffer4);

    }

    break;

}

}

}

else

{

    /* Output message on SCI2 */

    TextWrite("hello world! LCD Failed\r\n");

}

GlyphClose(&G_lcd) ;

TextWrite("Exiting Program, Good Bye!\r\n");

}

```

/\*\*\*\*\*\*

\* ID : 102.0

\* Function Name: InitSCI

\* Description : Initialize SCI2 to operate asynchronously with the selected baudrate  
from the prescaler and brg arguments.

\* Argument : prescaler --- Used to select the clock source for the UART

brg --- Used to select the baud rate for the UART using the baud rate  
generator value

\* Return Value : none

\* Calling Functions : none

\*\*\*\*\*/

static int InitSCI(int prescaler, int brg)

{

/\* Enable SCI2 \*/

MSTP(SCI2) = 0;

/\* RxD2-B and TxD2-B are used \*/

IOPORT.PFFSCI.BIT.SCI2S = 1;

/\* RxD2-B is input \*/

PORT5.DDR.BIT.B2 = 0;

/\* Enable Input Buffer on RxD2-B \*/

PORT5.ICR.BIT.B2 = 1;

/\* TxD2-B is output \*/

PORT5.DDR.BIT.B0 = 1;

```
/* Disable Tx/Rx */
```

```
SCI2.SCR.BYTE = 0;
```

```
/* Set mode register
```

```
-Asynchronous Mode
```

```
-8 bits
```

```
-no parity
```

```
-1 stop bit
```

```
-PCLK clock (n = 0) */
```

```
SCI2.SMR.BYTE = prescaler;
```

```
/* Enable RXI and TXI interrupts, even though we are not
```

```
using the interrupts, we will be checking the IR bits
```

```
as flags */
```

```
SCI2.SCR.BIT.RIE = 1;
```

```
SCI2.SCR.BIT.TIE = 1;
```

```
/* Clear IR bits for TIE and RIE */
```

```
IR(SCI2, RXI2) = 0;
```

```
IR(SCI2, TXI2) = 0;
```

```
/* Disable RXI and TXI interrupts in ICU because we are polling */
```

```
IEN(SCI2, RXI2) = 0;
```

```
IPR(SCI2, RXI2) = 2;
```



```

        IEN(SCI2, RXI2) = 1;

        IEN(SCI2, TXI2) = 0;

    /* Set baud rate to 115200

        N = (PCLK Frequency) / (64 * 2^(2*n - 1) * Bit Rate) - 1

        N = (48,000,000) / (64 * 2^(2*0 - 1) * 115200) - 1

        N = 12 */

        SCI2.BRR = brg;

    /* Enable Tx/Rx */

        SCI2.SCR.BYTE |= 0x30;

        return brg;
    }

    /**
     * ID : 103.0
     * Function Name: TextWrite
     * Description : Sends a text string to the terminal program.
     * Argument : msg_string - the string of characters to send.
     * Return Value : none
     * Calling Functions : none
     */

    static void TextWrite( char * msg_string)

```

```

{
    unsigned char i;

    /* This loop reads in the text string and puts it in the SCI2 transmit buffer */
    for (i=0; msg_string[i]; i++)
    {
        /* Wait for transmit buffer to be empty */

        while(IR(SCI2, TXI2) == 0);

        /* Clear TXI IR bit */

        IR(SCI2, TXI2) = 0;

        /* Write the character out */

        SCI2.TDR = msg_string[i];
    }
}

/*****

* ID : 103.0

* Function Name: BSP_Display_String

* Description : Sends a text string to the terminal program.

* Argument : aLine - index of line to draw string at

*           : aText - Pointer to zero-terminated ASCII text to draw at line

* Return Value : none

```

\* Calling Functions : main

\*\*\*\*\*/

void BSP\_Display\_String(int8\_t aLine, char \* aText)

{

int8\_t y = aLine \* 16;

/\* Draw text lines, 16 pixels high, 96 pixels wide \*/

/\* Clear the rectangle of this line \*/

GlyphEraseBlock(G\_lcd, 0, y, 95, y+15);

GlyphSetXY(G\_lcd, 0, y);

GlyphString(G\_lcd, (uint8\_t \*)aText, strlen(aText));

}

\*\*\*\*\*/

\* Function Name: Init\_CAN

\* Description : Initialize CAN to operate asynchronously at 500kbps.

\* Argument : none

\* Return Value : none

\* Calling Functions : none

\* Created by Sunil Kumar Gurram

\*\*\*\*\*/

void Init\_CAN()

{

IOPORT.PFJCAN.BIT.CAN0E = 1;

```
/* Port settings P3.2 - Tx(output) P3.3 - Rx (input) */
```

```
CTx_DDR = 1;
```

```
CRx_DDR = 0;
```

```
CRx_ICR = 1;
```

```
/*Clock settings Pclk(Fcan) set to 50MHz and Iclk set higher than Pclk*/
```

```
SYSTEM.SCKCR.BIT.PCK = 1;
```

```
SYSTEM.SCKCR.BIT.ICK = 0;
```

```
SYSTEM.SCKCR.BIT.BCK = 2;
```

```
SYSTEM.SCKCR.BIT.PSTOP1 = 0;
```

```
SYSTEM.SCKCR.BIT.PSTOP0 = 0;
```

```
MSTP(CAN0) = 0;
```

```
CAN0.C0CTLR.BIT.SLPM = 0; // Other than sleep mode
```

```
while(CAN0.C0STR.BIT.SLPST);
```

```
CAN0.C0CTLR.BIT.CANM = 1; // CAN reset mode
```

```
while(!(CAN0.C0STR.BIT.RSTST &&(!CAN0.C0STR.BIT.SLPST)));
```

```
/*Normal mailbox mode, standard id mode, overwrite mode, ID priority transmit mode*/
```

```
CAN0.C0CTLR.BIT.BOM = 0;
```

```
CAN0.C0CTLR.BIT.TSPS = 3;
```

```
CAN0.C0CTLR.BIT.TPM = 0;
```

```
CAN0.C0CTLR.BIT.MLM = 0;
```

```
CAN0.C0CTLR.BIT.IDFM = 0;
```

```
CAN0.C0CTLR.BIT.MBM = 0;
```

```
CAN0.C0CTLR.BIT.SLPM = 0;
```

```
CAN0.C0CTLR.BIT.TSRC = 0;
```

```
/*Fcan = 48 MHz, TSEG2 = 5, SJW = 4, TSEG1 = 16, Total = 25Tq, P + 1 = 4
```

```
Communication speed = 500kbps*/
```

```
CAN0.C0BCR.BIT.TSEG2 = 7;
```

```
CAN0.C0BCR.BIT.SJW = 1;
```

```
CAN0.C0BCR.BIT.BRP = 3;
```

```
CAN0.C0BCR.BIT.TSEG1 = 0x0E;
```

```
CAN0.C0CTLR.BIT.CANM = 2; // CAN halt mode
```

```
/*CAN test mode enabled, Self test (internal loop back)(should change only in CAN halt  
mode)*/
```

```
while(!CAN0.C0STR.BIT.HLTST);
```

```
CAN0.C0TCR.BIT.TSTE = 0;
```

```
CAN0.C0TCR.BIT.TSTM = 0;
```

```
/*TSRC should be set only in Operation mode*/
```

```
CAN0.C0CTLR.BIT.CANM = 0;
```

```
CAN0.C0CTLR.BIT.TSRC = 1;
```

```

//Enabling Interrupts for CAN

IEN(CAN0,RXM0) = 0;

IPR(CAN0,RXM0) = 4;

IEN(CAN0,RXM0) = 1;

for(i=0;i<32;i++){

    CAN0.COMCTL[i].BYTE = 0;

}

CAN0.COMIER = 0xFFFF;

}

/*****

* Function Name: CAN_Tx

* Description : Transmits data on to the CAN bus with the ID received from the arguments

* Argument : mbox_id --- selects the mbox to be used as a transmit mailbox

               data1 --- is the actual data to be transmitted

               sid ----- is the message ID for the data

* Return Value : none

* Calling Functions : none

* Created by Sunil Kumar Gurram

*****/

void CAN_Tx(int mbox_id, int data1,int sid){

    CAN0.COCTLR.BIT.CANM = 2; // CAN halt mode

```

```

/*Writing to COMB[] should be done when COMCTL is zero*/

CAN0.COMCTL[mbox_id].BYTE = 0;


CAN0.COMB[mbox_id].ID.BIT.IDE = 0;

CAN0.COMB[mbox_id].ID.BIT.RTR = 0;

CAN0.COMB[mbox_id].ID.BIT.SID = sid;

CAN0.COMB[mbox_id].DLC.BIT.DLC = 1;

CAN0.COMB[mbox_id].DATA[0] = data1;


//Sets the mailbox for transmission

CAN0.COMCTL[mbox_id].BIT.TX.TRMREQ = 0;

CAN0.COMCTL[mbox_id].BIT.TX.ONESHOT = 0;

CAN0.COMCTL[mbox_id].BIT.RX.TRMREQ = 1;

}

/*****

* Function Name: CAN_Rx

* Description : Receives data from the bus

* Argument : mbox_id --- selects the mailbox for receiving

            sid    --- allows the messages only with this ID

            sid_filter_enable --- enables the mailbox filter if 0x7FF and if the filter is enabled, allows

the    messages only with the selected SID

```

\* Return Value : none

\* Calling Functions : none

\* Created by Sunil Kumar Gurram

\*\*\*\*\*/

```
void CAN_Rx(int mbox_id,int sid, int sid_filter_enable){
```

```
    /*Writing to C0MB[] should be done when C0MCTL is zero*/
```

```
    CAN0.C0MCTL[mbox_id].BYTE = 0;
```

```
    CAN0.C0MB[mbox_id].ID.BIT.IDE = 0;
```

```
    CAN0.C0MB[mbox_id].ID.BIT.RTR = 0;
```

```
    CAN0.C0MB[mbox_id].ID.BIT.SID = sid;
```

```
    CAN0.C0MCTL[mbox_id].BIT.RX.RECREQ = 0;
```

```
    while(CAN0.C0MCTL[mbox_id].BIT.RX.RECREQ);
```

```
    CAN0.C0MCTL[mbox_id].BIT.RX.ONESHOT = 0;
```

```
    CAN0.C0MCTL[mbox_id].BIT.RX.RECREQ = 1;
```

```
    /*data should be written to C0MKR in halt mode*/
```

```
    CAN0.C0CTLR.BIT.CANM = 2; // CAN halt mode
```

```
    while(!CAN0.C0STR.BIT.HLTST);
```

```
    CAN0.C0MKR[1].BIT.SID = sid_filter_enable;
```

```
    CAN0.C0MKIVLR = 0;
```



```

        CAN0.C0CTLR.BIT.CANM = 0; // CAN operation mode
    }

/*****

* Function Name: transmit_data

* Description : Transmits data on to RS232

* Argument : data1 --- The data to be transmitted on to RS232

* Return Value : none

* Calling Functions : none

* Created by Sunil Kumar Gurram

*****/

int transmit_data(unsigned char data1){

    while(IR(SCI2, TXI2) == 0);

    IR(SCI2, TXI2) = 0;

    SCI2.TDR = data1;

    return 0;

}

/*****

* Function Name: receive_data

* Description : Receives data from RS232

* Argument : none

* Return Value : Returns the received value

* Calling Functions : none

* Created by Sunil Kumar Gurram

```

\*\*\*\*\*/

```
unsigned char receive_data(){  
    unsigned char rdata;  
    while(IR(SCI2,RXI2) == 0);  
        rdata = SCI2.RDR;  
        IR(SCI2, RXI2) = 0;  
    return rdata;  
}
```

\*\*\*\*\*/

\* Function Name: d3\_send\_string  
\* Description : Transmits string on to RS232  
\* Argument : s --- Pointer to the first character of the string  
\* Return Value : none  
\* Calling Functions : none

\*\*\*\*\*/

```
void d3_send_string(far int * s) {  
    while (*s != 'e') {  
        if (Q_Enqueue(&tx_q, *s))  
            s++;  
        else { }  
        while(IR(SCI2, TXI2) == 0);  
        IR(SCI2, TXI2) = 0;
```

```
        SCI2.TDR = Q_Dequeue(&tx_q);

    }

}
```

```
void DisplayDelay(unsigned long int units){

    unsigned long int counter = units * 0x100;

    while(counter--){ }

}
```

```
/******
```

```
* Function Name: Init_port
```

```
* Description : Initializes the port for the DIP switch
```

```
* Argument : none
```

```
* Return Value : none
```

```
* Calling Functions : none
```

```
* Created by Sunil Kumar Gurram
```

```
*****/
```

```
void Init_port(){

    PORTD.DDR.BYTE = 0;

    PORTD.ICR.BYTE = 1;

    PORTD.PCR.BYTE = 1;

}
```