LOW COST FAULT DETECTION SYSTEM

FOR RAILCARS AND TRACKS

A Thesis

by

SRIRAM T. VENGALATHUR

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2003

Major Subject: Mechanical Engineering

LOW COST FAULT DETECTION SYSTEM

FOR RAILCARS AND TRACKS

A Thesis

by

SRIRAM T. VENGALATHUR

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Reza Langari
(Chair of Committee)

Sooyong Lee
(Member)

Paul Roschke
(Member)

John Weese
(Head of Department)

August 2003

Major Subject: Mechanical Engineering

ABSTRACT

Low Cost Fault Detection System for Railcars and Tracks. (August 2003)
Sriram T. Vengalathur, B.E., B.M.S.C.E. (University of Bangalore), India
Chair of Advisory Committee: Dr. Reza Langari

A “low cost fault detection system” that identifies wheel flats and defective tracks is explored here. This is achieved with the conjunction of sensors, microcontrollers and Radio Frequency (RF) transceivers.

The objective of the proposed research is to identify faults plaguing railcars and to be able to clearly distinguish the faults of a railcar from the inherent faults in the track. The focus of the research though, is mainly to identify wheel flats and defective tracks.

The thesis has been written with the premise that the results from the simulation software GENSYS are close to the real time data that would have been obtained from an actual railcar. Based on the results of GENSYS, a suitable algorithm is written that helps segregate a fault in a railcar from a defect in a track.

The above code is implemented using hardware including microcontrollers, accelerometers, RF transceivers and a real time monitor. An enclosure houses the system completely, so that it is ready for application in a real environment.

This also involves selection of suitable hardware so that there is a uniform source of power supply that reduces the cost and assists in building a robust system.

To my parents and my sisters

ACKNOWLEDGEMENTS

I would like to thank Dr. Reza Langari for his invaluable guidance throughout the project and for bearing with many a delay. This thesis would not have been possible without his unwavering support and tremendous patience, thank you.

I would also like to thank Dr. Sooyong Lee and Dr. Roschke for their comments and insights.

There are many people who helped me at various stages of my project. I would like to thank the following: Mr. Ingemar Persson of GENSYS, for his valuable time and help during the simulations, Dr. Dongyoon Hyun, for valuable suggestions and support during the project review, Mohammad Jaradat, for sparing his time to help in hardware selection, Vijayaditya Kasarabada, for helping me out with the complex VC++ coding.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION………………………………………………… 1

A. Introduction……………………………………..1
B. Objective………………………………………..2
C. Justification for the proposed research………… 3
D. Literature review…………………………… 3
E. Summary of contributions………………………5
F. Outline of this thesis…………………………… 5

II CAUSES OF DERAILMENT…………………………………..... 6

A. Introduction…………………………………….. 6
B. Conditions other than a defective railcar that might cause derailment…………………………. 6
C. Faults in a railcar…………………………….. 8

III GENSYS……………………………………………………… 19

A. Introduction…………………………………….. 19
B. Railcar model used …………………………... 20
C. Simulations in GENSYS……………………...22
D. Conclusion……………………………………… 43

IV PRINCIPLES OF FAULT DETECTION……………………........ 44

A. Introduction…………………………………...44
B. On-board fault detection system……………….. 44
C. Identifying defective tracks…………………….. 48

V HARDWARE SYSTEM FOR FAULT DETECTION……………. 49

A. Introduction…………………………………...49
B. Overview of the system……………………........49
C. Organizing the fault detection system………….. 53
D. Functioning of the hardware………………….... 56
E. Hardware architecture…………………………... 56
F. Fault detection software…………………………57
G. Introduction of 68HC12………………………… 58
H. Overview of serial communication in 68HC12….. 62
I. Data registers used in serial communication……. 71
J. RF transmission………………………………… 71

CHAPTER Page

VI LABORATORY TESTING, CONCLUSION AND FUTURE WORK……………………………………………… 77

A. Introduction…………………………………….. 77
B. Prototype-1 with RS232 cables and Ming RF transceivers………………………….. 77
C. Development of an enclosure……………………78
D. Lab set-up……………………………………..80
E. Testing procedure……………………………...81
F. Conclusion……………………………………..82
G. Further work…………………………………...82

REFERENCES…………………………………………………………………….. 84

APPENDIX I ……………………………………………………………………... 87

APPENDIX II …………………………………………………………………... 89

APPENDIX III…………………………………………….............................94

APPENDIX IV…………………………………………………………………….. 109

APPENDIX V …………………………………………………………………... 158

VITA……………………………………………………………….………160

LIST OF FIGURES

FIGURE Page

1 Fatigue spalling………………………………………………………………. 10

2 Section view of spall…………………………………………………………… 11

3 Brinelling……………………………………………………………………….. 11

4 Fragment indentation…………………………………………………………. 12

5 Peeling…………………………………………………………………………12

6 Smearing………………………………………………………………………… 13

7 Etching…………………………………………………………………………13

8 Pitting……………………………………………………………………………14

9 Spun cone……………………………………………………………………...14

10 Wheel spalling………………………………………………………………….. 15

11 Shattered rim……………………………………………………………………. 16

12 Corrugated wheel……………………………………………………………….. 16

13 Wheel flats………………………………………………………………………. 17

14 Close up of one of the flats……………………………………………………… 18

15 Typical assembly of a 3-piece bogie………………………………………….20

16 3-piece bogie model used for simulations in GENSYS…………………….21

17 Longitudinal position of the center of gravity of the leading axle of the leading bogie……………………………………………….. 24

18 Longitudinal position of the center of gravity of the trailing axle of the leading bogie……………………………………………….. 25

19 Longitudinal position of the center of gravity of the leading axle of the trailing bogie……………………………………………. 26

20 Longitudinal position of the center of gravity of the trailing axle of the trailing bogie……………………………………………. 27

21 Position of the bolster beam…………………………………………………….. 28

22 Vertical acceleration in car-body over leading bogie ………………………29

23 Vertical acceleration in the center of the car-body…………………………30

24 Vertical acceleration in car-body over trailing bogie…………………….. 31

FIGURE Page

25 Car accelerations at different sections……………………………………. 32
26 Vertical force, tread, left wheel…………………………………………… 33
27 Vertical force, tread, right wheel……………………………………………34
28 Flange climb ratio left wheel, first axle……………………………………..35
29 Flange climb ratio right wheel, first axle……………………………………36
30 Flange climb ratio left wheel, second axle…………………………………37
31 Flange climb ratio right wheel, second axle……………………………….. 38
32 Flange climb ratio second bogie, left wheel, first axle…………………….. 39
33 Flange climb ratio second bogie, right wheel, first axle…………………… 40
34 Flange climb ratio second bogie, right wheel, second axle………………... 41
35 Response from an accelerometer in presence of a bearing fault……………46
36 Response from an accelerometer in presence of a wheel flat……………… 47
37 Overview of the system with the slave…………………………………….. 50
38 Overview of the system with the master……………………………………51
39 Expanded view of the process of RF transmission………………………… 52
40 Block diagram of the expanded wide mode of M68HC12A4……………. 62
41 Typical structure of each byte in NRZ format…………………………….. 63
42 Setup with Ming transceivers……………………………………………. 78
43 Penultimate stage of the enclosure…………………………………………79
44 Enclosed unit……………………………………………………………… 79
45 Final test set-up……………………………………………………………..80
46 Enlarged view of the encircled area in Fig. 45…………………………….. 81
47 Real time plot ……………………………………………………………. 82
48 Flow chart for the system………………………………………………….. 88
49 SCI control register 1……………………………………………………… 90
50 SCI control register 2………………………………………………………90
51 SCI status register 1……………………………………………………….. 91
52 SCI data register low……………………………………………………… 92

FIGURE Page

53 SCI baud control register high………………………………………………….. 93
54 SCI baud control register low…………………………………………………… 93

LIST OF TABLES

FIGURE Page

1 Simulation parameters………………………………………………………... 22

2 Test conditions…………………………………………………………….... 23

3 Memory map for 68HC12………………………………………………… 61

4 Sandwich structure of a data packet……………………………………….. 66

CHAPTER I
INTRODUCTION

A. Introduction

The railways in America suffer considerable losses each year due to derailments. The causes for these derailments are defects in the track and inherent faults in the railcar. Several attempts have been made to prevent these problems, but most of them have been restricted to the analysis of track and train dynamics; the derailment problem has not been analyzed extensively from the viewpoint of the defects in the track and the railcar. This gap motivates the topic for this thesis, i.e. to look at remedial measures from a different perspective.

Railways have come a long way in terms of development. There has been a tremendous progress in reduction of travel time with modern technologies contributing to speed of the engines. However, one thing that has not undergone a major change is the track; the tracks are the same in many countries. This is a major source of worry since the old tracks may not be able to handle current high-speed locomotives and may be a source of derailment [1]. Various kinds of special purpose railcars that are designed to handle different types of payloads also add to the existing problem, because the present day tracks are not designed to handle different types of locomotives and cars (as is the situation now in North American railways). Replacing or laying thousands of miles special purpose track is no menial task; it is also not cost effective [2]. Thus the current need is to develop a system that identifies potential faults in both railcars and tracks. The goal is to identify where the fault lies. It might so happen that faults existing in the railcar might damage the track.

Derailment of a train occurs when the wheels lift and slip out of the track. In more specific terms, a derailment occurs when the ratio of lateral displacement to vertical displacement, which is termed the L/V, ratio-exceeds a critical limit whose value is typically 1.2 [3].

The journal model is *IEEE/ASME Transactions on Mechatronics.*

There are several different ways or mechanisms by which a train/railcar can derail. The derailment might be due the defects in the track or due to a defect in the railcar itself.

A major cause for concern in the rail industry is the faults in railcar itself. There can be instances in which the track buckles due to a faulty bogie. Some faults in railcars like wheel-flats cause permanent damage to the tracks. Thus, it becomes crucial to rectify these faults before they cause further damage.

It is critical to identify a faulty bogie and also identify which part of a track is defective so that corrective measures can be taken.

Defects in a railcar can be very broadly classified as:

- Defects in the bogie and trucks
- Bearing faults
- Wheel defects

Of the above-mentioned faults wheel defects and bearing faults are the most damaging ones. Bearing faults have been researched extensively in the past as compared to wheel defects. Of all the physical damage that occurs in a wheel, a wheel flat is the most critical, because it is the one that occurs most frequently and causes severe damage to the track. A primary source for this is due to uneven braking of the wheels.

B. Objective

The main objective of the proposed research is to identify faults that plague a railcar and be able to clearly distinguish the faults of a railcar from inherent faults in the track. However, focus of this research is mainly to identify wheel flats. That is, the aim here is to develop a low cost, fault detection system that identifies whether the potential source for derailment is a faulty railcar or a defect in the track.

In order to achieve the desired objective, significant preliminary research needs to be done in terms of development of hardware and software. Research in terms of hardware involves development of the RF (radio frequency) links in conjunction with microcontrollers and also manufacturing the enclosure that houses the requisite hardware. Research in terms of hardware involves development of a robust algorithm that does not simply detect a fault but also does the same for the system as a whole that is, noise elimination algorithm for reliable RF transmission.

C. Justification for the proposed research

Most of the research carried out in the railroad industry thus far has been restricted to the analysis track and train dynamics. In the area of fault detection, detection of bearing faults has received maximum attention. Wheel flats have not been given priority although they cause significant damage. In current research efforts, detection of a wheel flat has been a relatively neglected field. The closest anyone has come to identifying a wheel flat by similar fault detection techniques is by using the bearing fault detection method of Dr. A. K. Chan [4], at Texas A&M University, where the signals meant to analyze the bearing fault are used to identify wheel flats. This method has not been very successful in detecting wheel flats as accurately as bearing faults.

An “on-board-real-time” fault detection system has not been explored extensively in this field, thus making the current research important.

D. Literature review

There has been significant work in the past with regard to identifying the cause for a fault, especially in identifying the bearing fault. The “Acoustic Bearing Detector “ [4] has been devised by Dr. A. K. Chan at Texas A&M University. This method is supposed to detect a faulty bearing to an accuracy of 85%. There are a couple of

drawbacks to this system though; this system cannot detect certain conditions like the grooved axle condition or defective roller condition. This is not an onboard system, the sensors are placed adjacent to the track, and these sensors pick up the sound waves from the bearings whizzing past them and analyze it on the spot. The maintenance of these systems can be pretty expensive. However, there is no such method for identifying a wheel flat condition. The acoustic bearing detector sometimes catches a signal, which is a characteristic response of a wheel flat, but no explicit algorithm has been created to identify wheel flats.

Preliminary study by R. M. Kaul [5], for a train protection warning system does use the concept of sending certain frequency signals to a box (with receivers to analyze signals) placed in each of the cars. This system aims at stopping a train for certain faults and does not use the “centralized system” for transmitting signals back and forth between the main unit and the controller. Also this system has not been developed to identify faults in a railcar.

Kumagai et al. [6], have done some extensive research on the occurrence of wheel flats and have devised certain measures to prevent it. But the issues tackled are more on the material science side; it does not indicate any method to identify the beginning of wheel flat, i.e. how the fault manifests when the train is in motion.

Research by D. H. Stone [7], indicates causes for the propagation of a shattered rim, but analysis is done only after damage has already occurred. Although certain specific causes are ascertained, it does not indicate how to take preventive measures when the defect is manifesting in a moving car.

The closest anybody has come to the proposed research is A. Filip [8], who has done substantial research in the area of a train integrity monitoring system. This system is useful to identify railcars that detach from the main cars. The drawback of this system though is that it uses relatively expensive GPS antennae and a sophisticated computer system. In addition it does not identify any other fault in a railcar.

TRACS [9], a system developed by Par Astrom of ABB, Sweden uses Motorola microprocessor for general monitoring of system but does not address the wheel flat issue. This system needs additional memory to store data thus adding to the costs.

E. Summary of contributions

The following items serve to summarize the contributions of this thesis

- Formulation of the problem statement by researching different kinds of faults.
- Modifying an existing simulation software GENSYS to suit the need of North American railroads.
- Creating the faults in the software and running the simulations for different test conditions to see the responses of a fault.
- Analyzing the behavior of the vehicle and deciding a suitable scheme to identify these faults.
- Conceptualization of a system to identify these faults.
- Implement the system using suitable electronics and hardware.
- Address the issues in radio frequency (RF) transmission.
- Write a suitable algorithm to implement a fault detection system.
- Create a Graphical User Interface (GUI) in VC++ to detect the faulty signals from a sensor.
- Package the setup, i.e. the microcontroller and the RF Transceivers in an enclosure such that it can be used in any kind of rugged environment.

F. Outline of this thesis

The organization of this thesis is as follows: Chapter II looks at the different causes for derailment and identifies the different problems that plague a railcar. Chapter III looks at the software used for simulation: GENSYS, the results are analyzed and

related to the type of fault. Chapter IV looks at a possible fault detection system to tackle the faults encountered in Chapters II and III. Chapter IV deals with the principle of fault detection. Chapter V deals extensively with the hardware used and the solutions to the problems encountered. Chapter VI concludes the thesis.

CHAPTER II
CAUSES OF DERAILMENT

A. Introduction

Before looking at potential causes for derailment of a railcar, let us analyze how a derailment might occur. Technically speaking the derailment of a train occurs when the wheels lift and slip out of the track. In more specific terms, derailment occurs when the ratio of lateral displacement to vertical displacement- termed the L/V ratio-exceeds a critical limit whose value is typically 1.2 [3].

There are several different ways or mechanisms by which a train/railcar can derail. The derailment might be due the defects in the track or due to a defect in the railcar itself. Section B explains some conditions other than the defects in the railcar that might cause the derailment and section C deals with the defects in the railcar that might cause the derailment.

B. Conditions other than a defective railcar that might cause derailment

Listed below are the conditions/mechanisms that might cause a railcar to derail. The conditions and mechanisms listed below are excluding defective railcars.

(a.) Derailment due to resonance [1]

The tracks are not defect free. Defects like waviness in lateral and vertical directions show up in the long run. This might excite railcars moving at high speeds to a resonant condition, which may lead to derailment.

(b.) Resonance due to packing material [1]

Most railcars are provided with packing materials whose primary function is to absorb the undesired vibrations and protect the load it is carrying. However, faulty packing occasionally drives the railcar to resonance.

(c.) Derailment due to the wheels being lifted of the track [1]

This situation might arise when the train is moving at a high speed on a curve and the wheel on the outer side might lift of the track above a critical speed. Also when a locomotive is negotiating a sharp curve it uses high horse power for better traction that gives rise to high contact stresses at the wheel-track interface that might cause the wheel to slip over the track.

C. Faults in a railcar

A major cause for concern is faults in the railcar itself. Derailments caused by the above mentioned factors are also initiated by faults in railcars. There can also be instances that the track buckles due to a faulty bogie [1].

Some faults in railcars like wheel-flats cause permanent damage to the tracks. It is very crucial to rectify these faults before they can cause further damage. It becomes critical here, to identify the faulty bogie and also identify which part of a track is defective so that corrective measures can be taken.

Defects in a railcar can be broadly classified as:

- Defects in the bogie and trucks
- Bearing faults
- Wheel defects

Of the above list, the last two are the major cause for concern. Extensive research has been conducted to identify the manifestation of these faults. The wheel flat has been relatively unexplored as far as fault detection is concerned. These are discussed at a later part in this thesis.

(a.) Bogie defects

The physical deformation of the parts of the railcar falls into this category. The defects can be a warped bogie/truck, which happens because of laterally or angularly misaligned wheel sets. This kind of a defect might cause the track to expand at certain points.

(b.) Bearing faults

Derailments of trains caused by wheel bearing faults are a significant problem in the rail industry. Trains often consist of 80-100 wagons and up to 1,600 bearings [10]. The failure of a bearing can cause significant damage to both the cars and the tracks.

One of the most extensively researched fields involves the bearing fault. Listed below are three of the most common conditions of occurrence of a bearing fault.

- Hot Bearing
- Spalling
- Spun Cone

As the name suggests a hot bearing occurs when the bearing is getting abnormally hot. If this condition is not identified in its initial stages it may lead to a catastrophic failure. This condition occurs mainly due to the loss/contamination of the lubricant that would increase the friction and hence high temperatures on the bearing surfaces result in a heat build up that leads to failure of the axle journal. Portions of oil wick could also get caught between the bearing and the journal resulting in heat [11].

The other common defect is fatigue spalling that is caused by metal stress fatigue. Spalling is basically sub-surface defects that propagate to the surface because of cyclic stresses. Spalling is the final stage of this propagation. When the material imperfection finally breaks away at the surface it is known as “Spall.” Fig. 1 shows a typical fatigue spall on a cone, Fig. 2 shows the section views through spalled components. Both of these are typical examples of metal stress fatigue explained above.

Brinelling is a common defect encountered in roller bearings. This occurs when indentations are made in the raceway made by rollers under extreme loading conditions. Brinelling can lead to spalling due to uneven load distribution [11]. A typical example of Brinelling is shown in Fig. 3.

Fragment indentation is a condition in roller bearings that occurs when the surface is damaged by debris passing through the roll tracks that result in surface dents [11]. These indentations can contribute to fatigue by acting as stress concentration point. This is illustrated in Fig. 4.

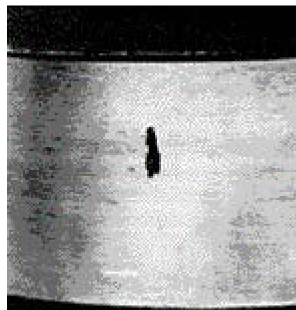


Fig. 1. Fatigue spalling [12]

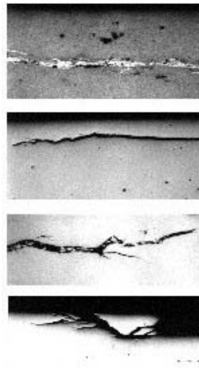


Fig. 2. Section view of spall [13]

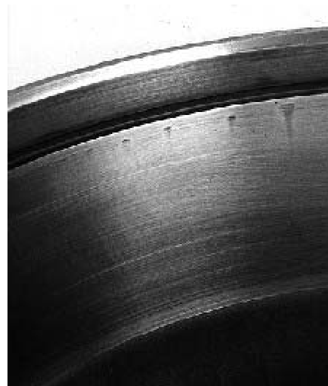


Fig. 3. Brinelling [13]

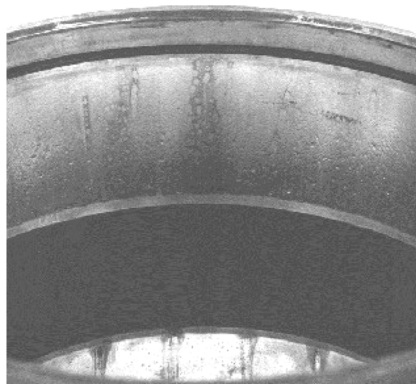


Fig. 4. Fragment indentation [13]

Peeling and smearing are conditions that arise when the rollers move in improper lubricating conditions. Peeling is minute particles coming of coming away from the surface. Smearing arises because of transfer of metal from one surface to another [11]. The phenomena are illustrated in Fig. 5 and Fig. 6.

Fig. 5. Peeling [13]

Fig. 6. Smearing [13]

Water etching and pitting are problems caused by the presence of moisture. During cooling of a hot bearing the vacuum inside the bearing attracts air and water vapor. This is shown in Figs. 7 and Fig. 8.

Fig. 7. Etching [13]

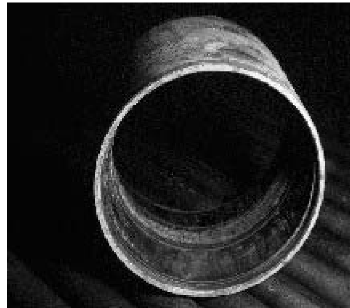


Fig. 8. Pitting [13]

Spun Cone is a condition in which the bearing wears out in a tapered fashion. This is because of non-conformity of the axle-journal diameters and also because of age factors. This is shown in Fig. 9.

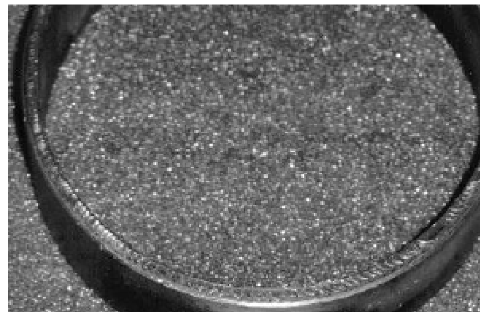


Fig. 9. Spun cone [13]

Bearing faults in huge proportions predominantly causes derailments. Therefore, it is very crucial to identify this fault in its initial stages and take requisite actions.

(c.) Wheel Defects

This is an important element, it also shelters many a faults. Listed below are a few of them

- Wheel Spalling
- Shattered Rim
- Corrugated Wheels
- Cracks
- Sub-Surface Defects
- Treading
- Wheel Flats

All these are the physical damage that occurs to the wheel.

The propagation of crack on the surface of the wheel contributes to wheel spalling (which is similar to occurrence in a bearing). Shown in Fig. 10 is a serious condition of a wheel spall.

Fig. 10. Wheel spalling [13]

Shattered rim is an extreme case of wheel spall as can be seen from Fig. 11.

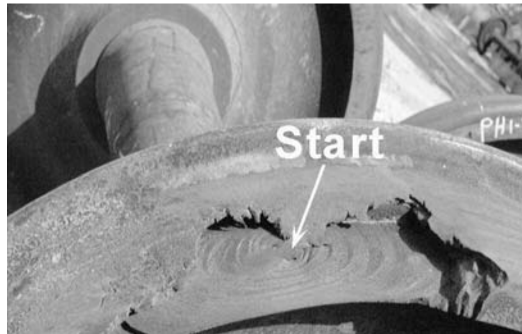


Fig. 11. Shattered rim [14]

Corrugation is a series of irregular waves on a structure. In railways corrugation is mainly seen on tracks (corrugated rails). But this defect can also be seen sometimes on the circumference of a wheel as shown in Fig. 12.

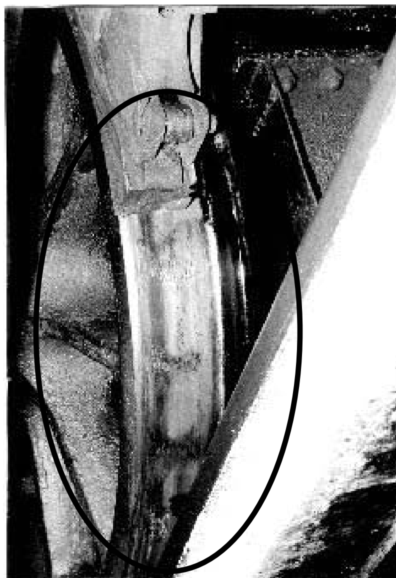


Fig. 12. Corrugated wheel [15]

One of the reasons for corrugation is great traction force on the wheel. This phenomenon is also called “polygonisation”.

Wheel flat is one of the widely researched topics as far as the defects in a rail car go. This, as the name suggests, is flatness on the surface of the wheel. Uneven braking and uneven loads are the two main causal factors. This also has a damaging effect on the track as can be visualized by Figs. 13 and 14.

Fig. 13. Wheel flats [16]

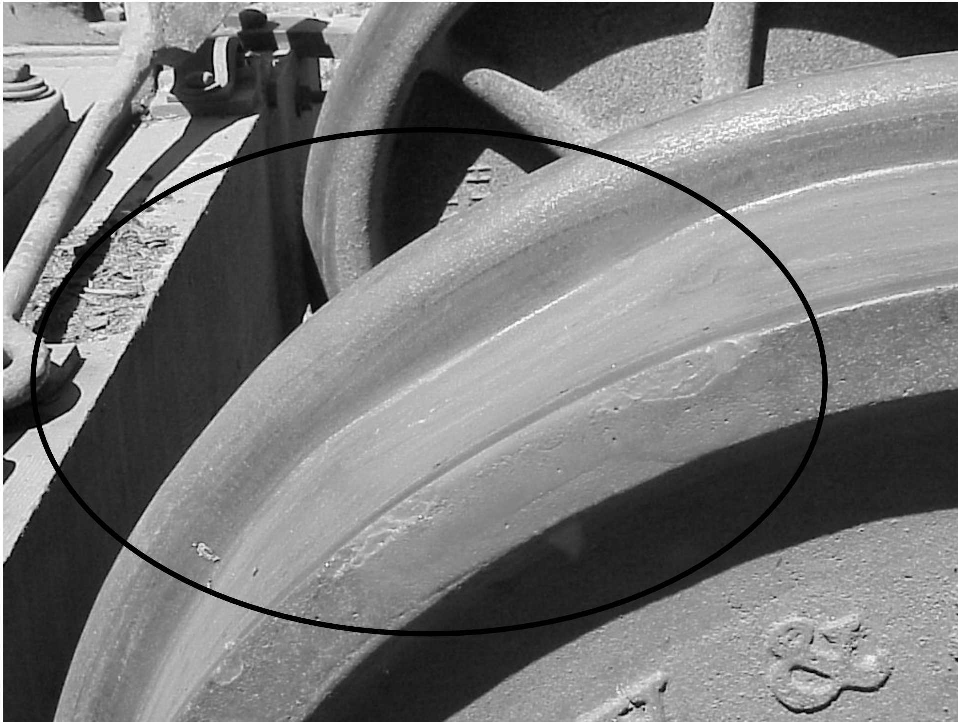


Fig. 14. Close up of one of the flats [16]

Of all the faults listed above, the wheel flat is the critical one, for this is the one that occurs most frequently and is not detected easily by any of the present diagnostic methods. Thus, this fault along with the bearing fault has been chosen for analysis by the proposed system as can be seen in later chapters.

CHAPTER III
GENSYS

A. Introduction

The response of an accelerometer due to defects on track, wheel, and bearings was initially an unknown quantity to begin with. No extensive research has been done in North American railways using an accelerometer as a sensor. This makes it difficult to get access to data that would show what a faulty data would look like. Restriction to access already existing data from the American association of railroads (AAR) test facility ruled out using a previous test data.

It was decided that simulation was the closest we could come to an actual test on a railcar. This apart, using software would give us the flexibility of simulating different test conditions that would have been difficult considering the effort and the cost involved.

There are very few rail vehicle dynamics simulation packages in use; this makes selection process relatively easy. Available software packages are VAMPIRE, GENSYS, SIMPACK, ADAMS and NUCARS. The selection process was done taking into account the results of "The Manchester Benchmarks for Rail Vehicle Simulation" [17].

The Manchester Benchmarks indicate that all the software codes perform similarly in most of the regions and concur with the general perspective. Keeping in mind the response from the companies, the cost and flexibility of the software, GENSYS was initially chosen to explore the different condition experienced by a railcar.

GENSYS is a multibody computer code and is widely used in railroad research in Europe and has been constantly validated by asea Brown Boveri (ABB) and Adtranz, Sweden, for different kinds of rail vehicle [18]. The source code for GENSYS is written mainly in FORTRAN-77 with the graphics part being taken care of by ANSI-C.

B. Railcar model used

Most of the bogies in North American railroads consist of 3 pieces, which differs from European railcars. In a 3-piece bogie model wheel sets support two side frames that support a bolster. The bolster is connected to the car body by a central pivot and side bearers with sliding surfaces. Seven sets of concentric springs provide the vertical and lateral suspension between the side frames and the bolster as shown in Fig. 15. Damping for the model is provided by spring loaded snubbing wedges that are located between the ends of the side frame and the bolster. The wedges are positioned such that a part of the body weight goes through the wedges causing the normal forces and thus the damping to vary with vehicle load [19]. A typical 3-piece bogie assembly is shown in Fig. 15.

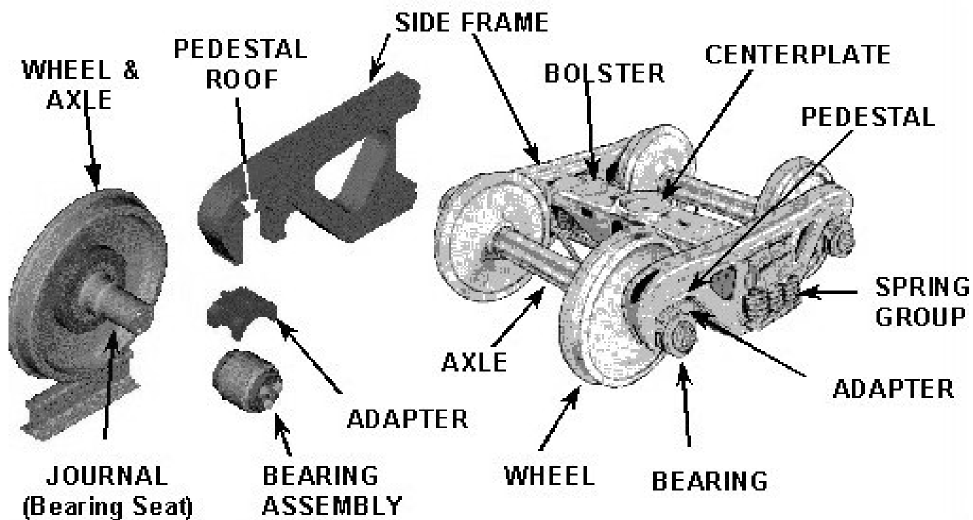


Fig. 15. Typical assembly of a 3-piece bogie [13]

The base model in GENSYS was created keeping in mind the European railcars. It had to be remodeled to match the 3-piece bogie model; the main code was modified to create a 3-piece bogie.

In GENSYS vehicle bodies, bogies and wheel sets are modeled as rigid bodies and have 6 degrees of freedom [20]. The track is modeled as a rigid body and with each connected wheel set there is a degree of freedom for each track body and hence the models contain 46 degrees of freedom. The 3-piece model of a truck created using GENSYS is shown in Fig. 16.

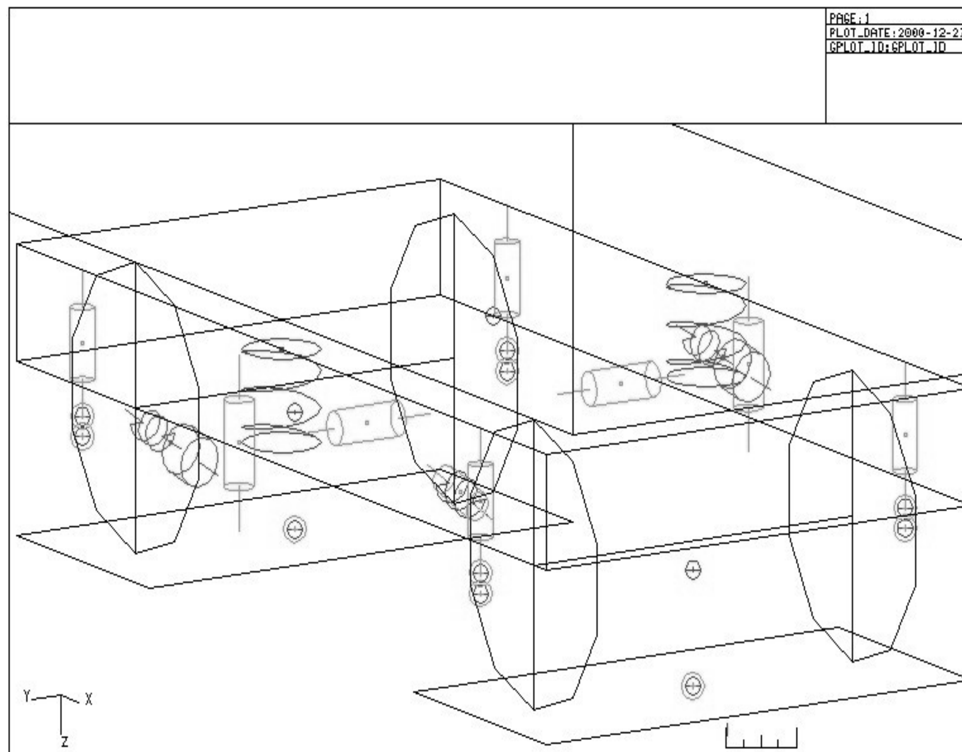


Fig. 16. 3-Piece bogie model used for simulations in GENSYS [21]

A 3-piece bogie conforming to the standards of American railroads was created with the option of changing the loads from a full load of 120 ton to any other value. For simulation purposes full and half loads, i.e. 120 and 60 tons were used at 3 different speeds.

C. Simulations in GENSYS

As mentioned in the earlier chapters, we are trying to identify wheel flats with the option of a bearing fault being included for future research. Thus when the faults were created using GENSYS bearing fault was also included.

(a.) Creation of faults

Changing the wheel geometry in the original code simulates a wheel flat. Thus at the circumference of the wheel a small disturbance is introduced which is very close to a miniscule wheel flat.

Bearing fault was introduced into the model by making a small deformation in the bearing, which would very roughly represent a bearing fault.

(b.) Simulation

Simulation was run for 2 different load sets at 3 different speeds.
The main simulation parameters are shown in table 1.

Table 1. Simulation parameters

Simulation distance	1.6 kms
Simulation loads	60, 120 tons
Simulation speeds	50, 70, 90 km/h

The different test conditions were:
Full load: 120 tons
Half load: 60 tons

Table 2. Test conditions

LOAD	**SPEED (km/h)**	**FAULT**
Full	50	None
Full	70	None
Full	90	None
Half	50	None
Half	70	None
Half	90	None
Full	50	Wheel flat
Full	70	Wheel flat
Full	90	Wheel flat
Half	50	Wheel flat
Half	70	Wheel flat
Half	90	Wheel flat
Full	50	Bearing fault
Full	70	Bearing fault
Full	90	Bearing fault
Half	50	Bearing fault
Half	70	Bearing fault
Half	90	Bearing fault
Full	50	Wheel flat & Bearing fault
Full	70	Wheel flat & Bearing fault
Full	90	Wheel flat & Bearing fault
Half	50	Wheel flat & Bearing fault
Half	70	Wheel flat & Bearing fault
Half	90	Wheel flat & Bearing fault

(c.) Discussion of results

Figs. 17-34 show results of the simulation of a combined bearing fault and wheel flat at 70 kmph and with a full load. The results are discussed after the figures.

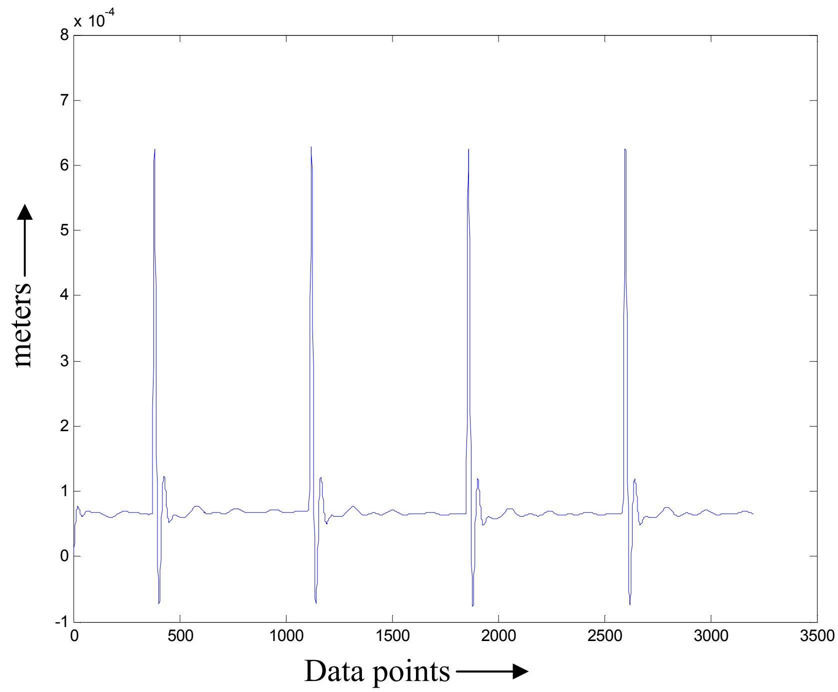


Fig. 17. Longitudinal position of the center of gravity of the leading axle of the leading bogie

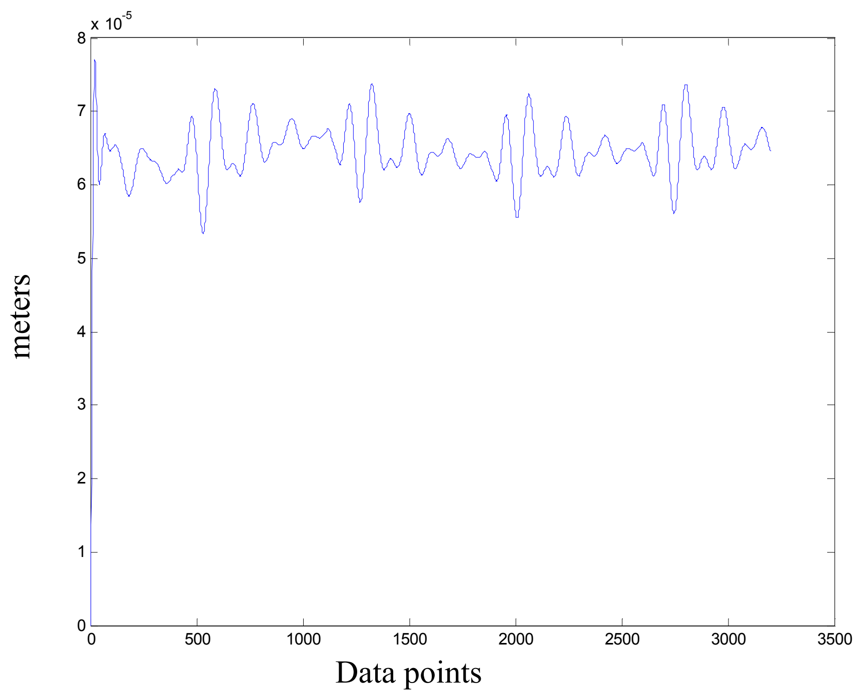


Fig. 18. Longitudinal position of the center of gravity of the trailing axle of the leading bogie

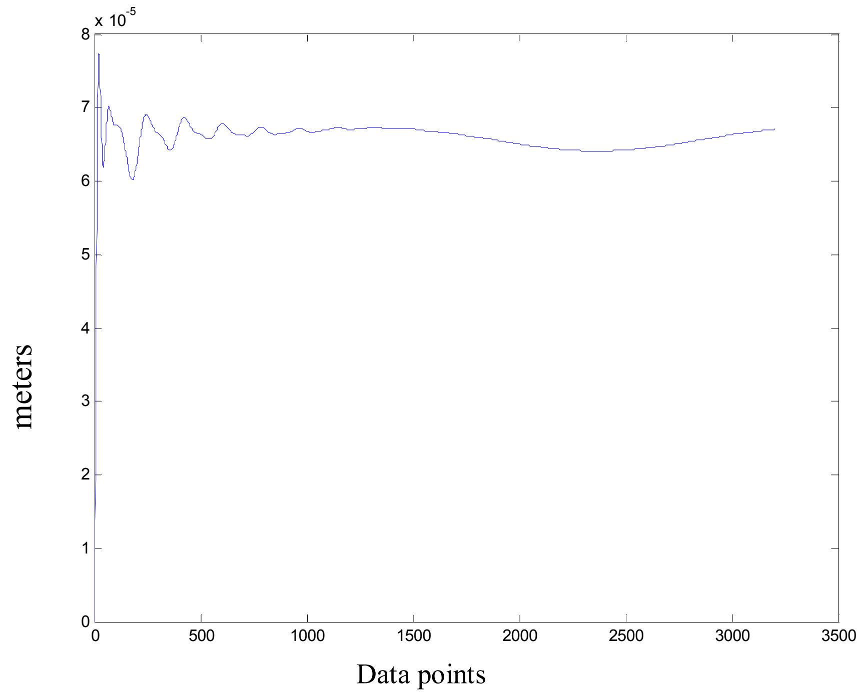


Fig. 19. Longitudinal position of the center of gravity of the leading axle of the trailing bogie

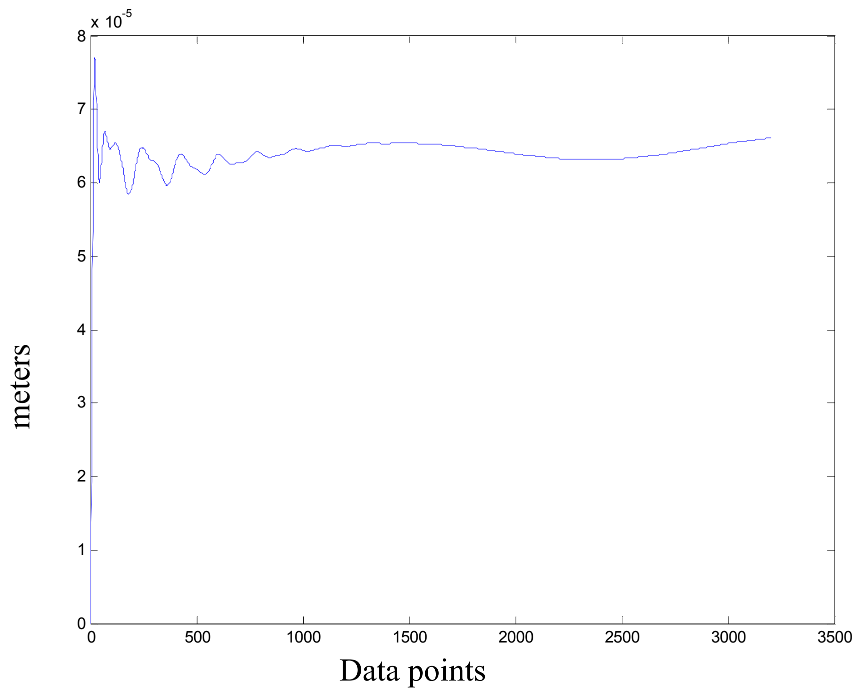


Fig. 20. Longitudinal position of the center of gravity of the trailing axle of the trailing bogie

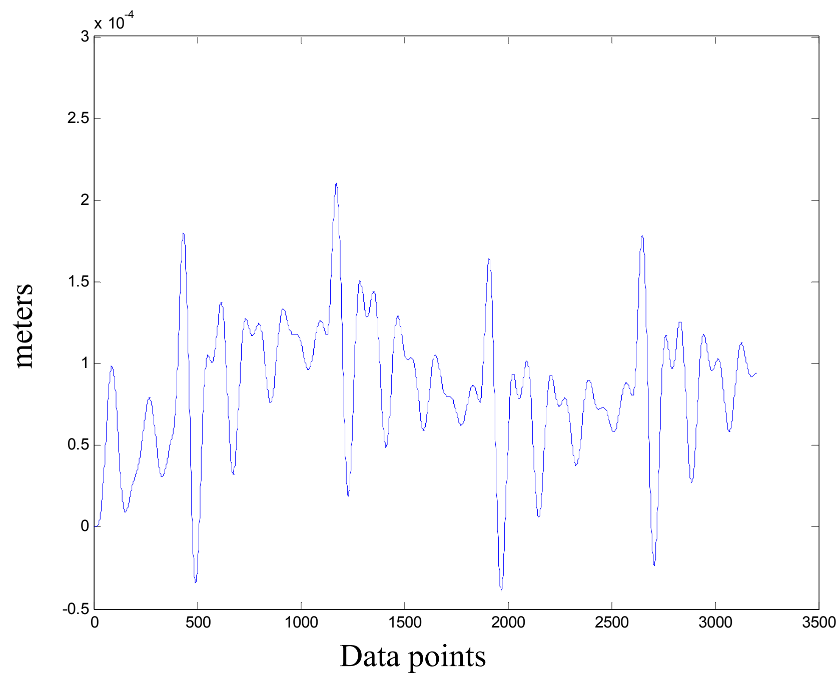


Fig. 21. Position of the bolster beam

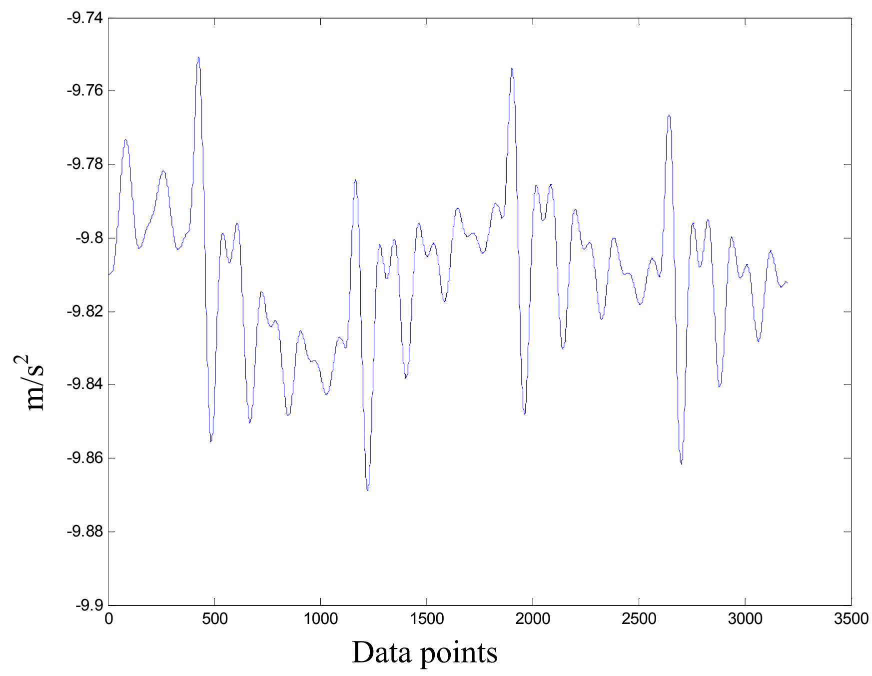


Fig. 22. Vertical acceleration in car-body over leading bogie

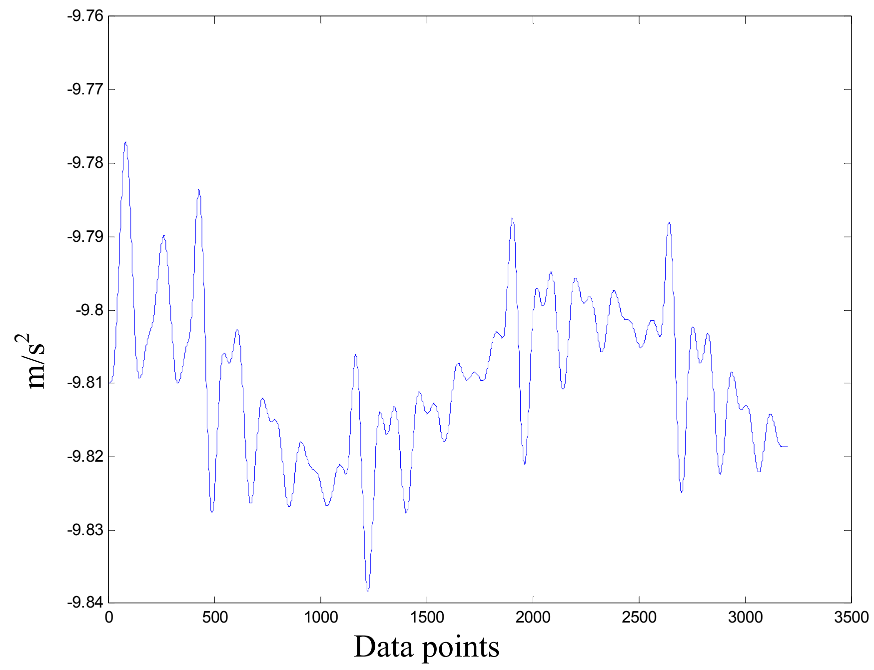


Fig. 23. Vertical acceleration in the center of the car-body

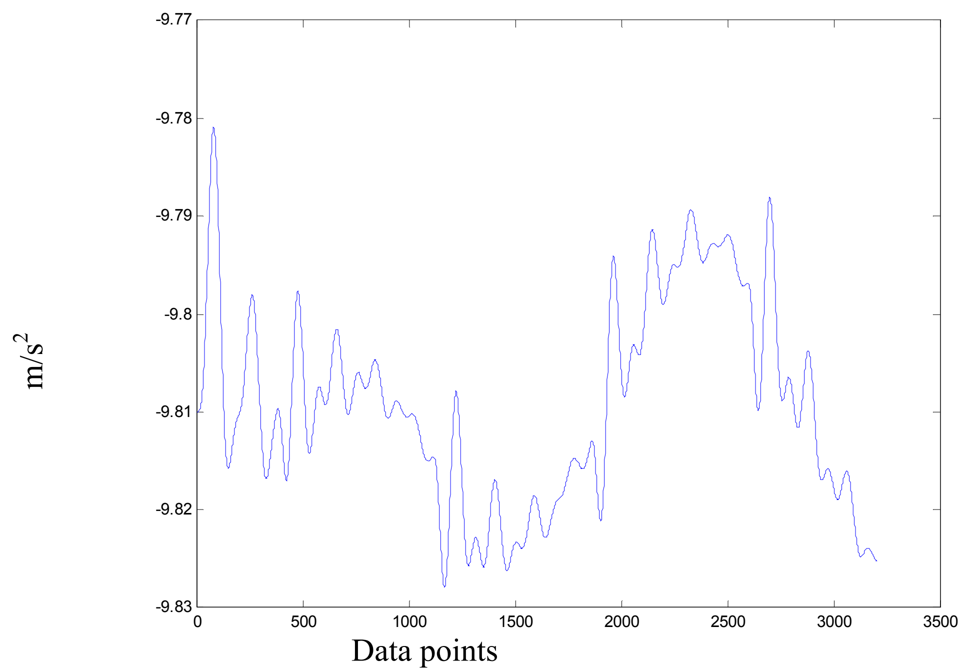


Fig. 24. Vertical acceleration in car-body over trailing bogie

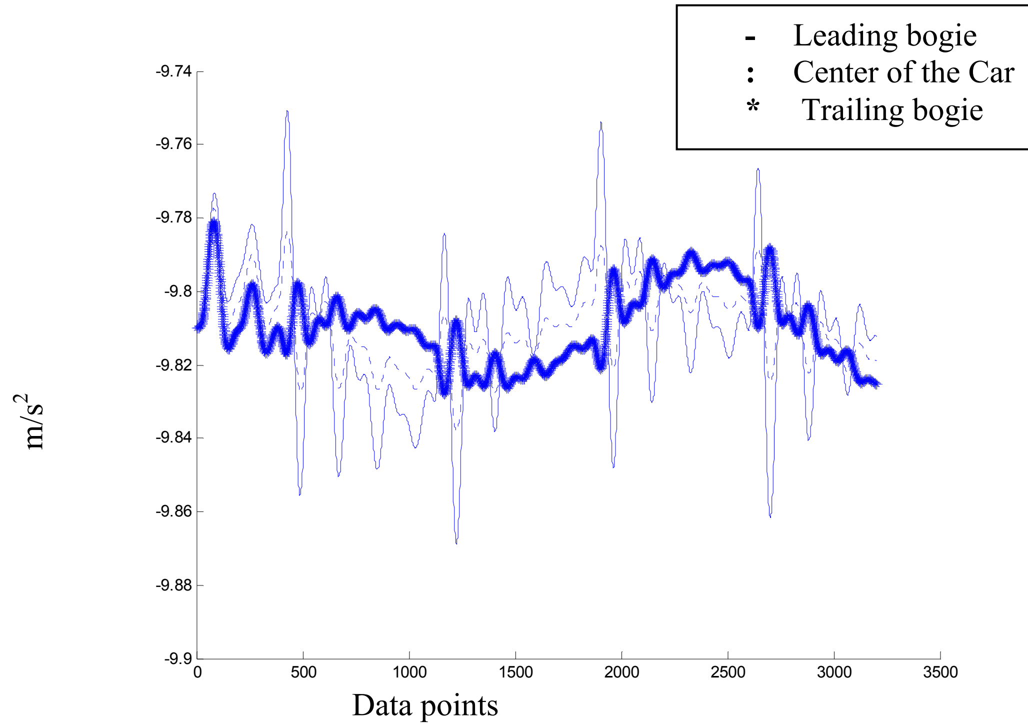


Fig. 25. Car accelerations at different sections

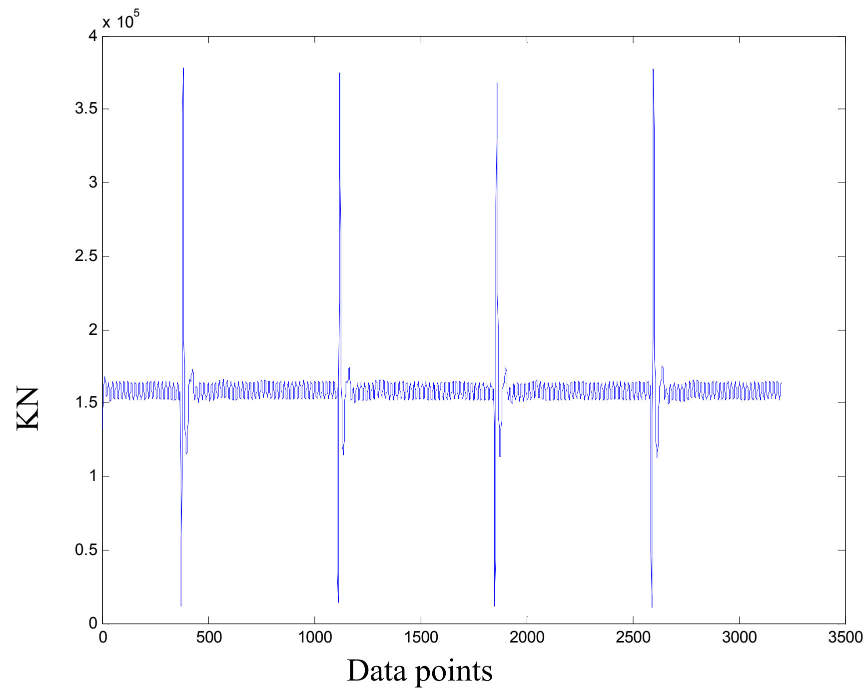


Fig. 26. Vertical force, tread, left wheel

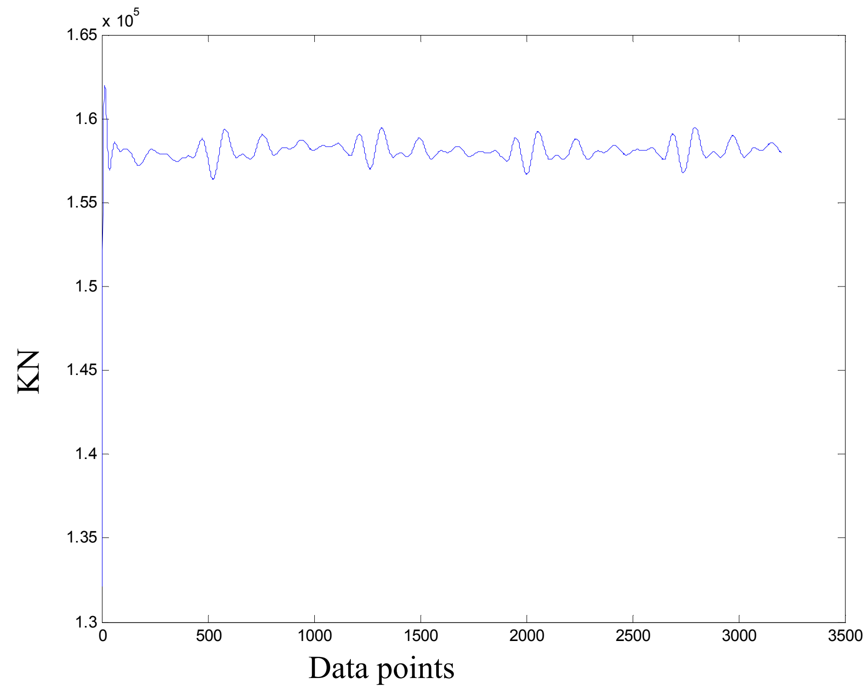


Fig. 27. Vertical force, tread, right wheel

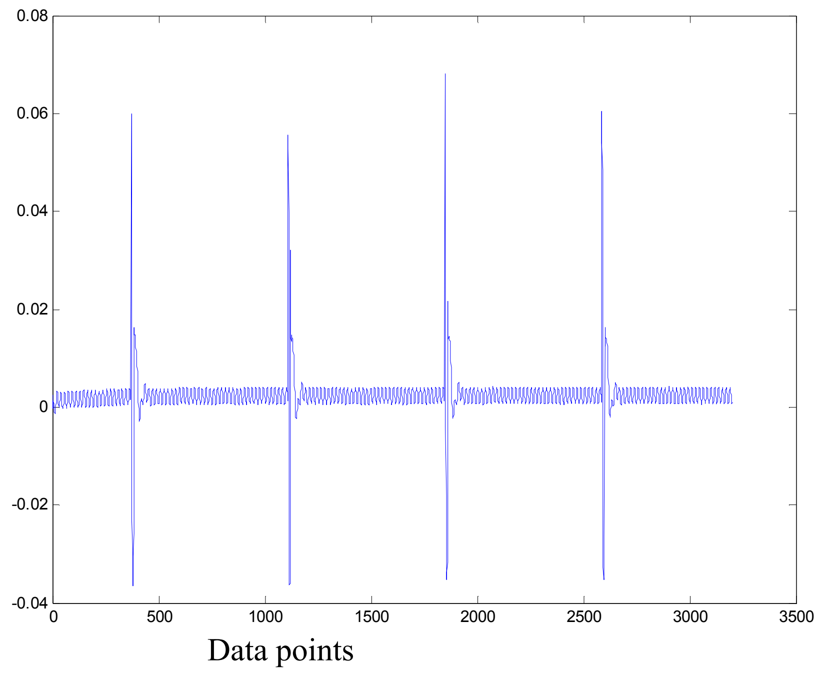


Fig. 28. Flange climb ratio left wheel, first axle

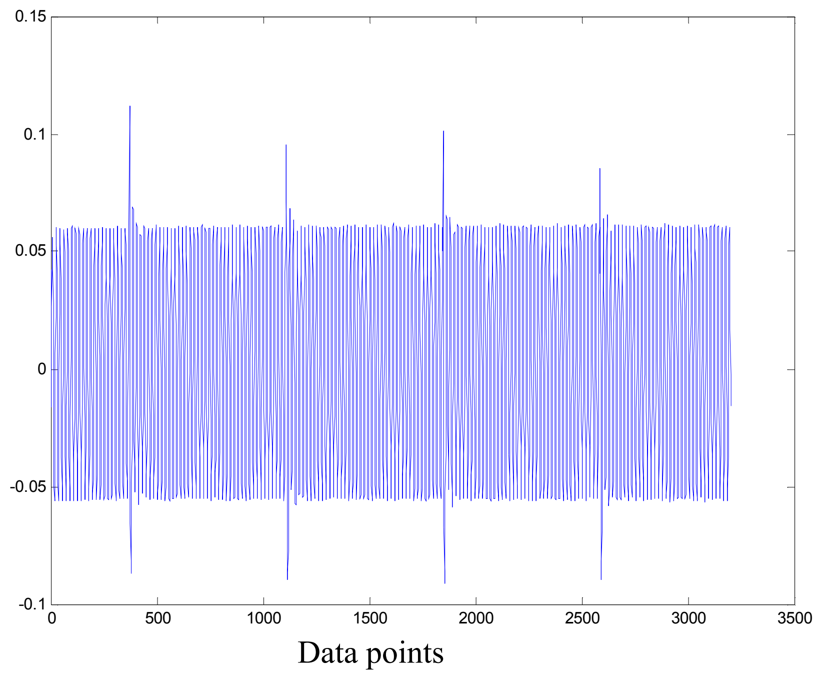


Fig. 29. Flange climb ratio right wheel, first axle

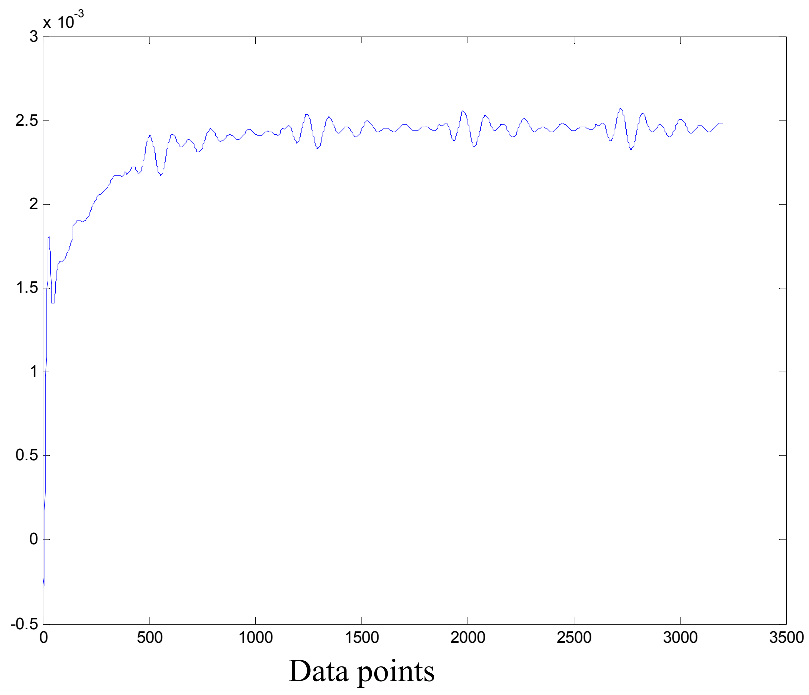


Fig. 30. Flange climb ratio left wheel, second axle

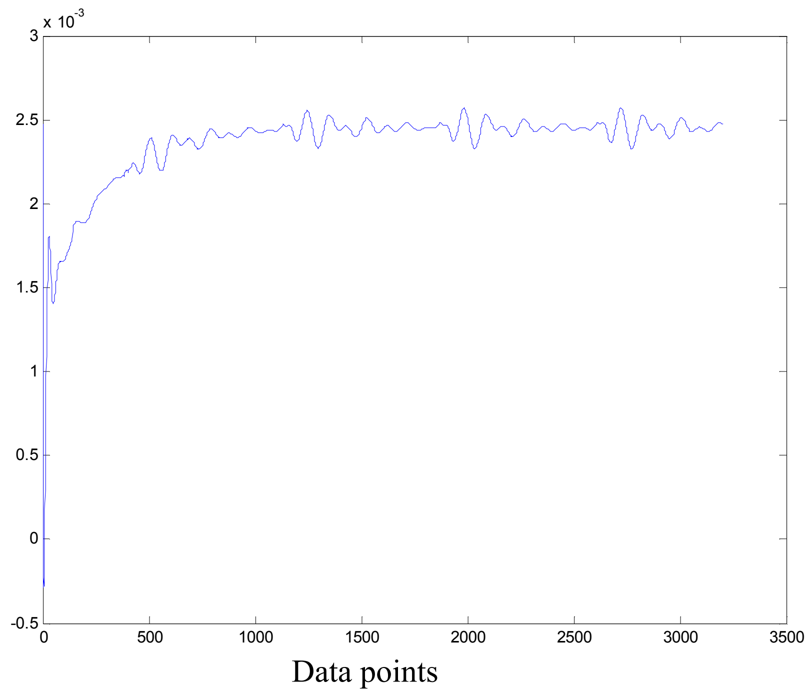


Fig. 31. Flange climb ratio right wheel, second axle

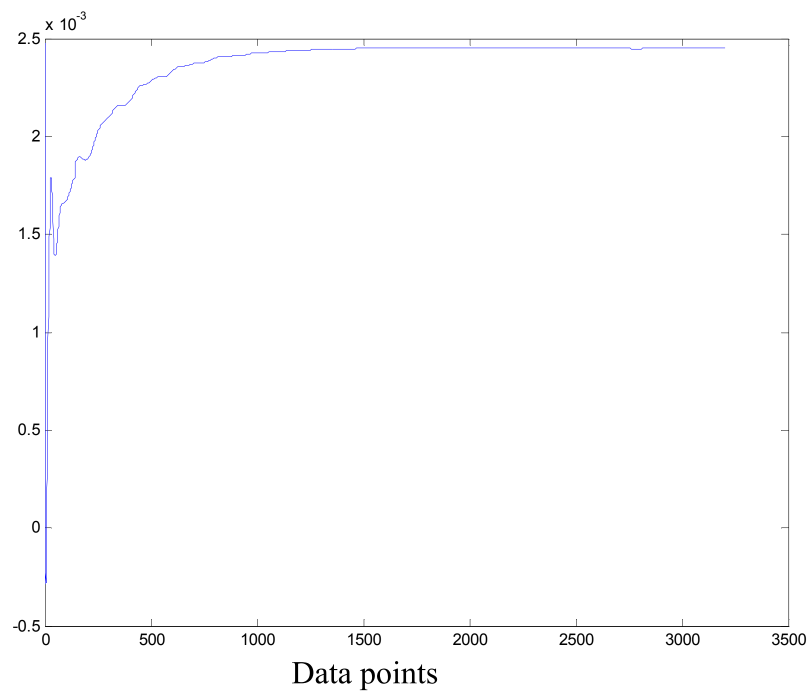


Fig. 32. Flange climb ratio second bogie, left wheel, and first axle

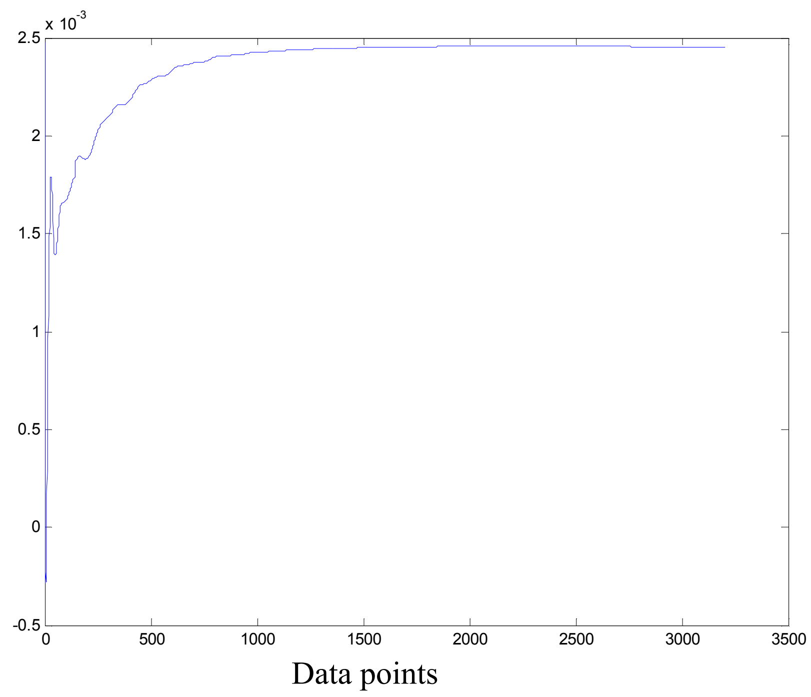


Fig. 33. Flange climb ratio second bogie, right wheel, and first axle

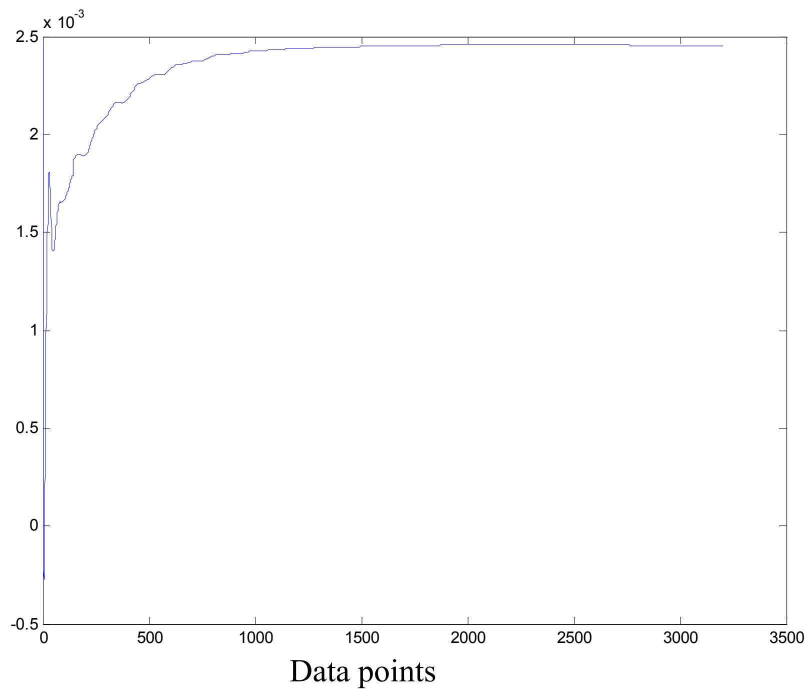


Fig. 34. Flange climb ratio second bogie, right wheel, and second axle

Attached above are a few of the important plots from the simulations run on GENSYS. These plots enable a better overview of a rail vehicle behavior, more specifically at the points we desire in presence of a fault. The software gives us flexibility to place different sensors at vantage points to measure variables like acceleration and contact forces.

Fig. 17 is a plot of the longitudinal position of the C.G of leading axle of the leading bogie. It can be seen that the bearing fault is not having any noticeable effect on the leading position of the bogie, but the wheel flat shows up on the plot in a big way. This is because of the impact that a wheel flat causes. There is a temporary loss of contact between the wheel and the rail and just before the moment of impact. A large force on the wheel is transmitted to the axle and thus to the bogie. This impact can be

detected by any inexpensive sensor and can be detected easily as opposed to the bearing fault whose effect is not obvious in the plot.

The next three figures show the effect of the wheel flat waning away. Figs. 18-20 shows the lateral position of the trailing axle of the leading bogie and the leading and trailing axles of the trailing bogie, that is the wheel flat has a major effect only of the axle where it is located. The next plot explained further verifies this.

Fig. 21 shows the position of the bolster beam. In the first look it seems that both the wheel flat and the bearing fault have an effect on the bolster beam which is surprising considering the fact that the bearing fault had almost no effect on the leading axle which, is nearer to the fault than the bolster. A closer look at the plot indicates that the magnitude is much smaller than for the other plots. This further verifies the fact that the effect of the wheel flat wanes as the distance from the fault increases.

The next plot shown in Fig. 22 shows the vertical acceleration in car-body over leading bogie. This can be easily monitored by an accelerometer. The plot when looked at without any reference might appear difficult to identify any faults, but, as we look the plots in the next two figures, i.e. Fig. 23 and Fig. 24 we can see that the magnitude of acceleration is distinguishably higher than that of the accelerations in the center of the car and at the trailing bogie. In fact the accelerations at the center of the car and at the trailing bogie do not show any pattern as compared to the leading bogie, where sharp peaks can be noticed at the points where wheel flat makes contact with the rail.

The next plot summarizes what was said in the previous paragraph. Fig. 25. shows accelerations at the leading bogie, the center of the car and the trailing bogie. It can be seen clearly from this plot that the faulty bogie gives out sharp peaks indicating a fault. The other parts of the plots almost match in all the three cases, although the leading bogie does show some higher peaks because of the bearing fault, such small differences will be difficult to distinguish in real situations.

Plots in Figs. 26 and 27 indicate the vertical force at the left and the right wheels in the leading bogie. The forces at the other axles are very small and can be neglected. This case gives an option of exploring a force sensor to identify faults. It can be clearly

seen that the magnitude of the force shoots up the moment the wheel flat touches the rail. The bearing fault too seems to increase the contact force, though not on a very large scale. The fault on the left wheel does not affect the contact forces at the right wheel, thus enabling us to use the forces at the wheel rail interface to identify the faults explicitly and more conclusively.

The remaining plots deal extensively with the most important factor in identifying (or preventing) the derailment: the flange climb ratio. As mentioned earlier L/V ratio is a crucial factor in derailment of rail vehicles. The critical value typically lies around 1.2. None of the cases in the simulation threaten to approach this value for the magnitude of the fault created is not that serious. The fault created is at the right wheel, thus we can see that the magnitude of the L/V ratio is high at the right wheel-track interface, a part of the repercussion can be seen at left wheel-track interface.

D. Conclusion

Numerical simulations reported in this chapter gave us a very clear picture on how the response of the vehicle will be in presence of a fault. This also gives us an opportunity to explore different sensors like strain gages and accelerometers to get similar kind of responses from a real vehicle. On the basis of this simulation the system was further developed.

CHAPTER IV

PRINCIPLES OF FAULT DETECTION

A. Introduction

With regards to this project, fault detection implies, identifying the faults that have a potential to become damaging ones. And "damaging ones" further implies those faults that would not just damage the tracks but might lead to derailment.

Since, not all the faults listed in chapter II occur frequently, we prioritize and treat those faults that occur most frequently, which are also the ones that cause the maximum damage. Wheel flats and bearing fault are two such commonly occurring faults. As already discussed earlier bearing faults that are left unattended can lead to catastrophic failures.

Wheel flat has not been given as much importance as its bearing counterpart as far as the detection part of it goes, but it ranks alongside bearing fault in terms of the damage caused to the railways, both in terms of damage to the tracks and monetarily.

Thus, these two faults, i.e. wheel flats and bearing faults, have been chosen for the first stage of fault detection. The other faults have been ignored for the moment since their damage is relatively less, but this system can be later extended to other kinds of faults.

B. On-board fault detection system

As mentioned in section C of chapter I, there are well-established systems to identify bearing faults; the same systems sometimes detect signals of a wheel flat,

though very infrequently. A major disadvantage of this system is that it is a wayside fault detection system. Although it can identify a bearing fault up to an accuracy of 85%, maintaining a large number of such systems throughout the railways network will prove to be an expensive affair.

The aim here is to create a low-cost, low-maintenance on-board fault detection system; this will not only eliminate the costly job of installing an expensive system and maintaining it but will also eliminate unnecessary processing of a multitude of data.

The principle of operation of the system will basically be to pick up signals from some kind of a sensor and analyze those signals to look for the fault, which would basically be identifying a signature response. The idea is based on the assumption that each kind of fault will give out a signature response (as has been seen in various systems).

Vibration at certain points is considered to look for a fault. This was decided after running some simulations on GENSYS as discussed in the previous chapter.

The vibration response was unique for both types of faults. The bearing fault was characterized by a high frequency signal, whereas sharp peaks at regular intervals characterize the wheel flat, as can be seen from Figs. 35 and 36. These responses were obtained by simulating in GENSYS.

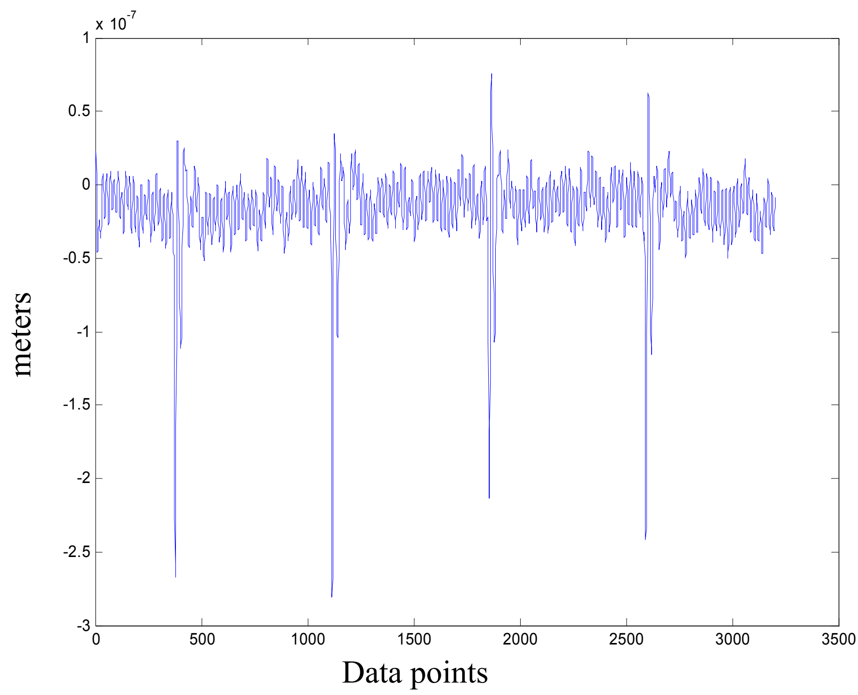


Fig. 35. Response from an accelerometer in presence of a bearing fault

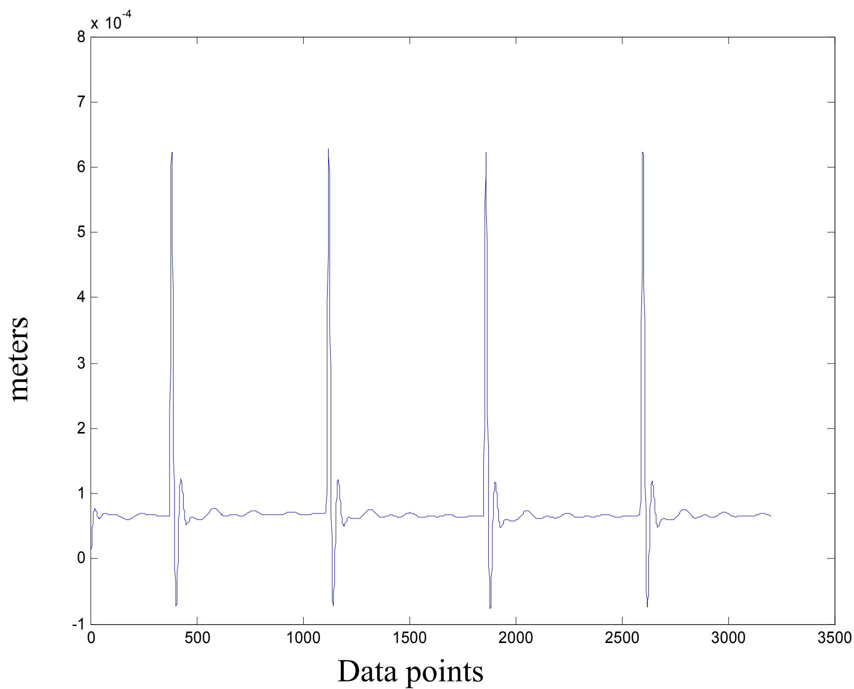


Fig. 36. Response from an accelerometer in presence of a wheel flat

These plots appear as expected. A wheel flat is approximately a smooth flat surface at the circumference of the rim. When a flat hits the rail the wheel loses contact with the rail for a fraction of a second; when it comes in contact again with the rail it creates a certain impact, which stands out as a sharp peak in the response. On the other hand the bearing fault on the other hand is made up of many small surface defects, and is located at the center of the wheel and thus gives out a high frequency response.

It is relatively easy to identify a wheel flat for it stands out very significantly, the effect of noise and other factors can be ignored while identifying a wheel flat.

For a bearing fault, different types of frequency analysis can be used to identify the fault, but in real world it becomes much more difficult, for there may be many other similar kinds of noises.

Considering that we do not have any access to existing data at AAR and certain memory issues from the sensor we chose to narrow our target to just wheel flat for the moment.

Thus using vibrations emanating from certain points in a railcar as a signature response, the fault detection system will be packaged and placed at a suitable point on the railcar.

C. Identifying defective tracks

A very interesting find that came out during this research was the ease with which faulty tracks could be identified. Electronics have been used explicitly to find out different types of faults, but never has been track defects identified, save for those track cars, which can never explore the length of the tracks all over North America. This follows a very simple algorithm, which waits for all the signals from different cars and then analyzes the characteristic of these signals. If all the cars have emitted the same signal in a given period of time then we know that there has to be a defect outside the cars, which points to a defective track. Thus we can record the distance where the fault lies in the track. A very basic algorithm is explained in appendix III.

CHAPTER V

HARDWARE SYSTEM FOR FAULT DETECTION

A. Introduction

The faults are identified using various hardware setups; all the requisite hardware should gel as a single unit while monitoring the fault.

Since the need of the hour is a low-cost system, one of the key issues to be kept in mind is the issue of power management of the hardware. As will be seen in the following sections the electronic part of the hardware involved requires some sort of voltage source for functioning and by proper selection of the electronic units, a single power source can be used to supply power to the hardware; this also reduces the periodic checks of hardware in the system.

B. Overview of the system

From the previous chapter we know that we need some kind of a sensor that will send the signals for identification of the fault. There has to be a unit, which will monitor all the vibration in a railcar and to supervise such small units we will need a main monitoring system. We will call the main monitoring system as “Master” and the other smaller units as “Slaves,” in a broad sense.

Communication between the master and slave monitoring systems will take place via radio frequency (RF) transmission. Thus the system overall will look similar to Fig. 37. The signals picked up from the sensor will be sent through the RF link to the master for data logging.

Since, the signals picked up from the sensor will have to be processed constantly for identifying faults, each slave will have a Motorola 68HC12B32 board, and the master will have a Motorola 68HC12A4 board. Thus, signals from the sensor are sent to the controller, which in turn, based on the seriousness of the fault communicates to the master through the RF link (Fig. 38).

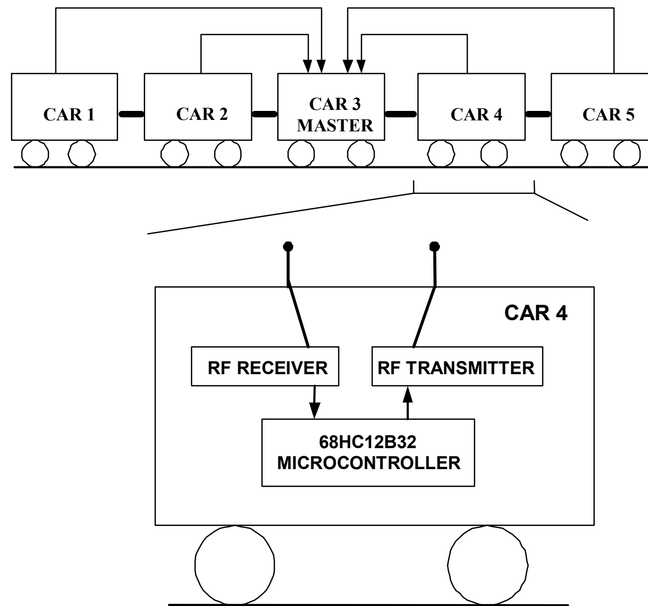


Fig. 37. Overview of the system with the slave

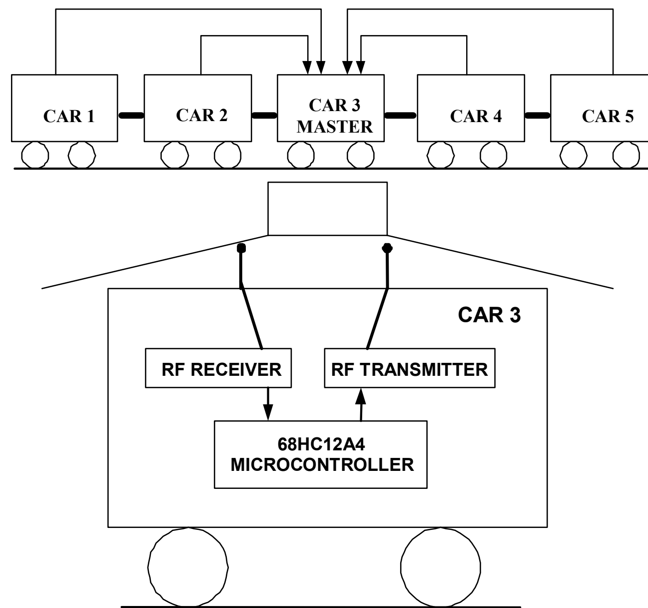


Fig. 38. Overview of the system with the master

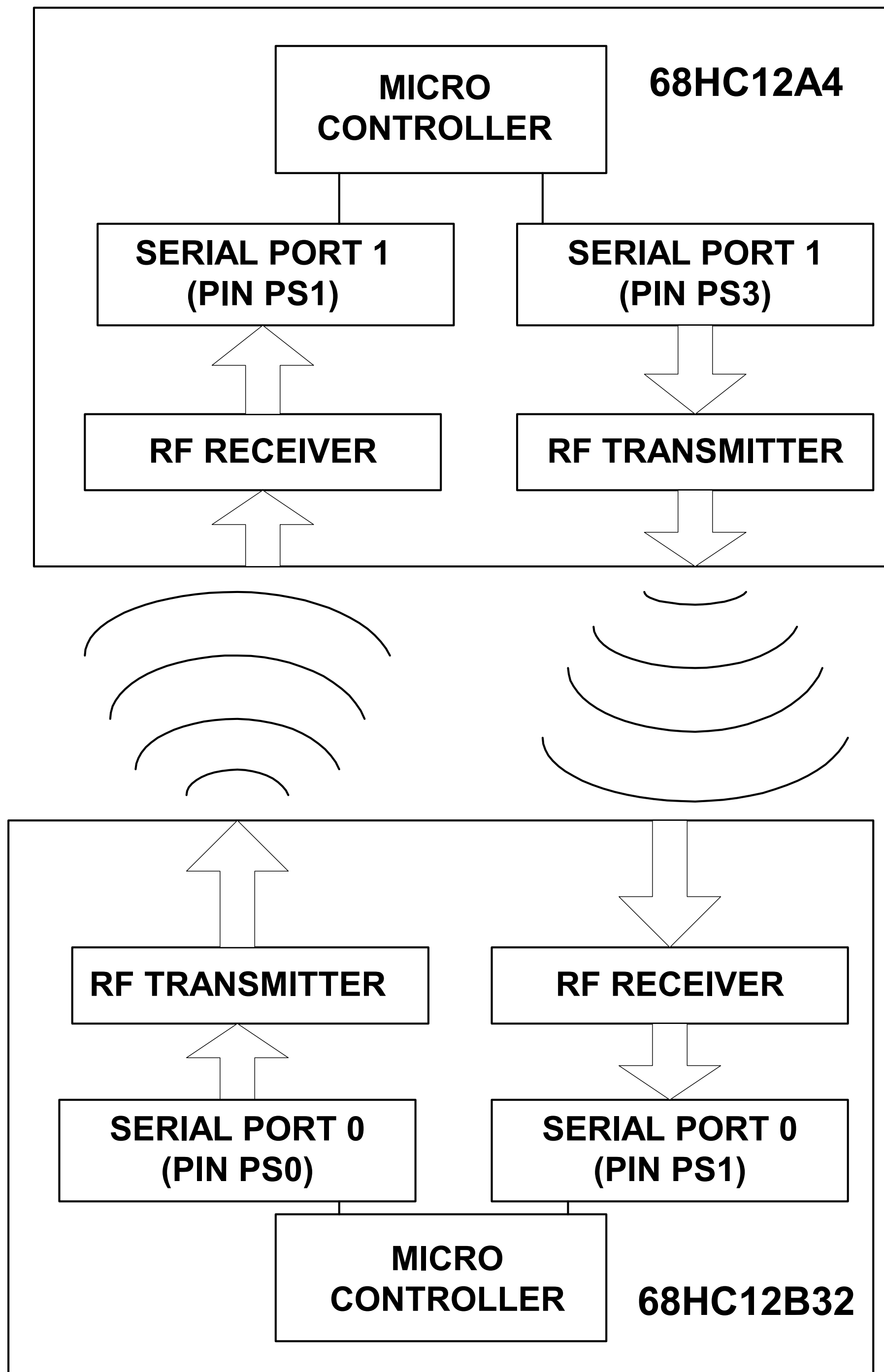


Fig. 39. Expanded view of the process of RF transmission

C. Organizing the fault detection system

Since we have an overall idea on how to implement the system, we can split the task into three broad categories.

- Selection of a proper sensor
- Selection of hardware for analysis purposes
- Assembling the whole unit to create a fault detection system

(a.) Selection of a proper sensor

Since we are aiming at identifying wheel flat and the bearing fault, it will be appropriate to place the sensor at a point where it can pick up identifiable signals from both the points. Since the bearing fault occurs mainly at the wheel axle junction and the wheel flat at the surface of the rim, the best place for the sensor would be just before the wheel-axle interface.

This area will be subjected to extreme conditions like variations in temperature, dust, mechanical strains etc. A very robust sensor will therefore be needed, which will not only pick up the vibrations from the interface but also survive the harsh environs.

Piezoelectric accelerometers are designed for low mechanical strain and are relatively unaffected by thermal transients, thus it was decided as an ideal choice for a sensor. To this end, we chose Analog Devices' ADXL202®, which is a 2-axis piezoelectric accelerometer.

(b.) Selection of hardware for analysis purpose

After sensor receives the signal, some kind of a processor is needed for preliminary analysis that would also prevent unnecessary transmission of data that might

clog the master processor. Since the sensor is placed in all the railcars, the centralized controller-master will monitor the sensors and processors in all the cars.

Thus, this can be broadly classified into two different stages.

- Sensor-Slave Stage
- Slave- Master Stage

As the names suggest, sensor-slave stage involves the interaction between the slave and the sensor and the slave-master stage involves the interactions between the master and the slave.

Both the slave and master will have a fair amount of processing to do. The slave has to continuously monitor the data and look for any ominous signs of a fault. In case a fault is manifest the slave alerts the master. The tasks a slave is expected to perform mainly are:

- Respond to the commands sent by the master.
- Pick up the signal from the sensor and process the signal to look for high frequency components (characteristic of a bearing fault).
- Pick up the signal from the sensor and look for sharp peaks (characteristic of a wheel flat).

The master processor on the other hand is expected to have certain functionalities, a few of which are:

- Capable of interpreting user-entered commands on a PC and sends signals to the slaves accordingly.
- Be able to prioritize signals arriving from various slaves.
- Be able to interpret the data and decide whether a fault lies in a particular section of a track or on some railcar.

Thus it is clear that some kind of a micro controller that has adequate processing capabilities is needed to perform the functions of both the master and slave.

The controllers are expected to perform various tasks like isolating the noise from the actual data, processing the data, and responding to commands issued by the user.

It is imperative that the controllers used have sufficient memory apart from having the desired functionalities. After conducted a preliminary research on the various kinds of commercially available controllers, Motorola 68HCXX series of microcontrollers were found to have requisite functionalities required for our purpose. Keeping in mind the various constraints like the cost-efficiency, the memory issue, the ease of use and available documentation for the controllers, the Motorola M68HC12A4 and M68HC12B32 boards were chosen. Both have the capability to act as a "Master" controller. For our purpose though the M68HC12B32 has been chosen as the "Master" controller.

These are 16-Bit microcontrollers with 1K of RAM and 768bytes of EEPROM with facility to incorporate fuzzy logic.

The next issue that crops up is how to effect the communication between the master and the slave. Since Motorola 68HCXX series have been chosen, the serial ports of the microcontroller can be used to effect serial communication (explained later) between the boards. The simplest way to communicate is just to connect the serial ports through an RS-232 cable. But the problem arises when you want to make these connections in a real situation; the total number of cars in a typical freight train is approximately 100-150, with the total length of the exceeding 1,000 feet. Having the master-slave communication through a wire in this case is virtually impossible for a host of reasons, which is explained in RF transmission section. Wireless communication is not only efficient but also robust and fast. For this reason a Radio Frequency (RF) link was chosen over communication by wires. Glolab corporations' TM1V and RM1V transmitters and receivers were chosen for this purpose. The transmitter module TM1V is a 418 MHz RF transmitter and it is capable of sending serial data at a rate of up to 4800 bps (bits per second). A huge plus for the transmitter considering that we are aiming at a low cost system is the low power consumption of 1.5 ma. The receiver module RM1V also has the same features. When TM1V and RM1V are used in conjunction, serial data can be transmitted easily over a distance of 300 feet. The technical details of these are attached in appendix V.

(c.) Assembling of the fault detection system

The idea, as stressed earlier, is to pick up the signals from a sensor and transfer the signal through a microcontroller, which in turn transmits data though a RF transceiver to a "Master" controller, which will analyze the data further. It is proposed to package the entire hardware- sensor, microcontroller and transceiver- in a single enclosure so that it can be attached to a railcar at an appropriate location. This makes the system simple to maintain and use.

D. Functioning of the hardware

After the components are assembled, the master (68HC12A4) and the slave (68HC12B32) subsystems appear as shown in Fig. 38. Thus when the master receives a command from the user it performs the required function. Let us say that the command is to fetch the accelerometer data, the master interprets the command accordingly and then sends a command through pin PS3 of the serial port1 to the RF transmitter. These commands are in the form of a packet that is received by the RF receiver on the slave side and sent to the microcontroller through the pin PS0 of serial port 0. The microcontroller now fetches the accelerometer data from the analog – digital converter and sends the data through pin PS1 of the serial port 0 to the RF transmitter. This data is received by the RF receiver on the master side and send to the microcontroller through the pin PS1 of the serial port 1. The master then prints this value on the screen. This is just a very basic explanation on how the data actually flows in this set-up.

E. Hardware architecture

Thus to summarize, the Hardware architecture consists of:

- A PC, which acts as a user-friendly terminal, and that interacts with a "Master" subsystem.

- A “Master” subsystem will execute the user prompted commands.
- A “Slave” subsystem responds to the “Master” commands.

The “Slave” subsystem is comprised of the following hardware:

- Motorola 68HC12B32 Board
- Accelerometer
- Glolab TM1V and RM1VTransmitter and Receiver

The “Master” subsystem comprises of the following hardware:
Motorola 68HC12A4 Board
Glolab TM1V and RM1V Transmitter and Receiver
All these will be assembled in an enclosure as will be seen in the last chapter.

F. Fault detection software

The fault detection software should be able to identify the faults manifested in the car. Below is a simple algorithm used for this purpose
Pseudo code for the master:

- Wait for interrupts to occur
- If interrupt is from a slave then transmit the data to the PC terminal
- If some information is required from the slave, communicate with the slave and re-transmit information to the user.
- If receive interrupts from different slaves, analyze all the data and determine the kind of fault, if any.

Pseudo code for the slave:

- Wait for the interrupt from the master, but keep processing the signals from the sensor.
- If there is a critical signal from the sensor, then immediately inform the master, otherwise keep analyzing data from the sensor.
- If and interrupt occurs because of a command from the master, respond to the request by sending appropriate information, the command is usually for a missing data.

Identification of a critical signal in a Slave

- Continuously check the Accelerometer readings
- If the amplitude of the signal goes over the preset (threshold) value, then indicate that there is a fault
- If the command is to check for a bearing fault, then store some values from the accelerometer and perform FFT on the signal to identify very high frequency components.

The flowchart for this system is shown in appendix I.

G. Introduction to 68HC12

Motorola has two variants of their 68HC12 Micro controller, the 68HC12A4 and the 68HC12B32. The 68HC12 series has 16- Bit Micro controllers. These micro controllers are compatible with the 68HC11, therefore there is not an issue of incompatibility with the old code; all HC11 commands are accepted by the 68HC12.

The 68HC12A4 board contains a low-power, high speed CPU. Two asynchronous serial communication interfaces, a serial peripheral interface, a flexible 8-channel timer, a 16-bit pulse accumulator module, an 8-channel, 8-bit analog-digital converter, 1 kilobyte of RAM, 4 kilobytes of EEPROM, and a single-wire Background Debug Mode (BDM) module. The 68HC12A4 also has some additional features like the

capability to expand over 5 MB of memory, non-multiplexed address and data buses, and phase-locked loop and 24-key wakeup lines with interrupt capabilities.

The 68HC12B32 has almost all the processing capabilities of the A5. Few things different from the A4 are that it has only 1 asynchronous serial communication interface and 768 Bytes of EEPROM. The B32 has some unique features like 32 KB flash EEPROM and a built in pulse-width modulator.

The A4 is normally configured to run mainly in an expanded mode with some of the resources existing outside the chip, whereas the B32 is configured to run in a single chip mode are all the resources are in the chip.

The A4 though can be operated in seven different modes of operation. They being

- Special single chip
- Special expanded narrow
- Special peripheral
- Special expanded wide
- Normal single chip
- Normal expanded narrow
- Reserved
- Normal expanded wide

The factory default mode is the normal expanded wide mode. In this mode the expanded bus is present with a 16-bit data bus. Port D is the low byte data bus and port C is the high byte data bus.

In the normal expanded narrow (x8) mode of operation, the expanded bus is

present with an 8-bit data bus. Port C functions as the data bus in this mode. No external bus is available in the Normal single chip mode of operation. All program and data fetches are from on-chip memory or peripheral registers. Ports A, B, C and D are available for general purpose I/O.

The special peripheral mode of operation is a test mode. The CPU is not active. On-chip peripherals may be accessed directly by an external bus master. It is not possible to change from this mode without going through reset.

In Special single chip mode, the background debug mode is immediately active out of reset. Execution begins from the background debug ROM. Commands are sent to the CPU through the background debug interface pin.

In Special expanded narrow mode port D may be used it view the upper 8 bits of the data bus.

In the special expanded wide, special expanded narrow and special single chip modes provide the same functionality as the respective normal modes. These modes are primarily for testing and provide access to several key features as described above. Not all the ports are available as I/O ports in all the modes; some are used as address or data buses.

The memory map for the A4 is shown in table 3. The memory map basically says which addresses are available for programming.

Table 3. Memory map for 68HC12 [22]

Address Range	Description	Location
$0000 - $01FF	CPU registers	On-chip (MCU)
$0800 - $09FF $0A00 - $0BFF	User code/data Reserved for D-Bug12	1 K on- chip RAM (MCU)
$1000 - $1FFF	User code/data	4 K on chip EEPROM (MCU)
$4000 - $7FFF	User code/data	16 K external RAM
$8000 - $9FFF $A000 - $FD7F $FD80 - $FDFF $FE00 - $FE7F $FE80 - $FEFF $FF00 - $FF7F $FF80 - $FFFF	Available for user programs D-Bug12 program D-Bug 12 startup code User-accessible functions D-Bug12 customization data Available for user programs Reserved for interrupt and reset vectors	32 K external EPROM

The basic structure and configuration of the ports of a 68HC12A4 in its expanded mode is shown in the Fig. 40 below.

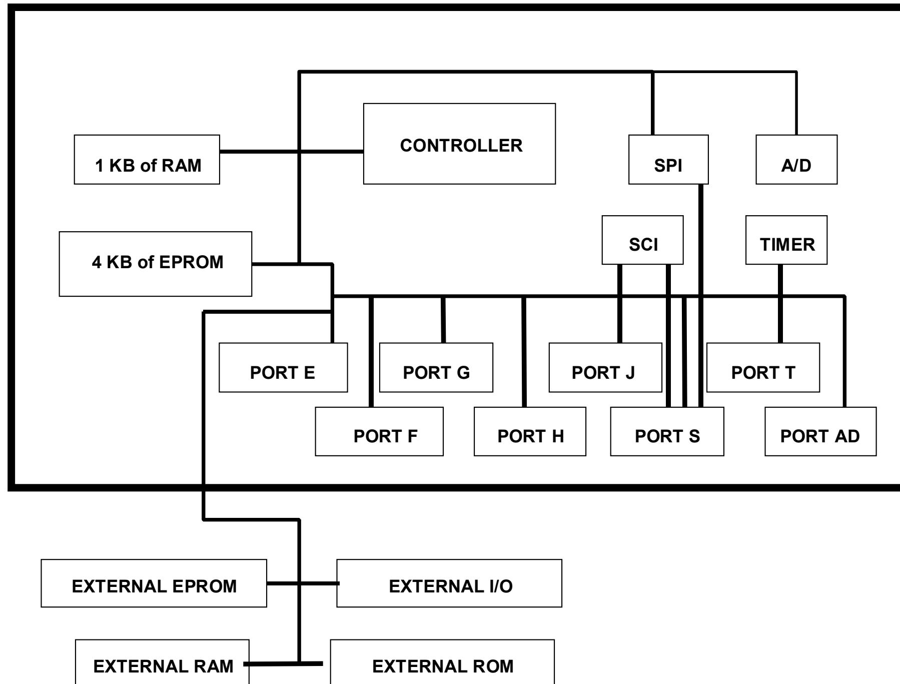


Fig. 40. Block diagram of the expanded wide mode of M68HV12A4 [22]

H. Overview of serial communication in 68HC12

The 68HC12 has three independent input / output systems: two serial communication interfaces (SCI) and a Serial Peripheral Interface (SPI). The pins of port-S also double up as serial communication pins, as shown in Fig. 39.

The SCI communication on the A4 and the B32 boards is in the NRZ format as shown in Fig. 41, i.e. data are sent in the following format- one start bit, eight or nine data bits and one stop bit.

STOP BIT	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	START BIT

Fig. 41. Typical structure of the each of byte in NRZ format

The form of communication used is generally asynchronous communication (discussed in the following section); SPI is used for synchronous communication. The boards will be in sleep mode most of the time except when a request is made by the user or a critical data are to be transmitted for further analysis, the former procedure is called as the "Wake-up" method, and this is explained later under the heading SCI system. This not only reduces unnecessary burden on the "Master" controller – we have to keep in mind that we are not just dealing with a single "Slave" but multiple "Slaves"- but also saves a lot of power, which is really crucial considering the low-cost of maintenance we are aiming at, not to forget the ease with which we can deal with systems in asynchronous communication.

(a.) Serial versus parallel communication

Sometimes it becomes imperative that two systems communicate with each other. For this type of communication there are a certain set of standards and methodologies. For communications between two 68HC12 boards, the communication can either be serial or parallel. In parallel communication an exclusive data line is reserved for each bit to be transferred and all bits are transferred almost simultaneously. By contrast, in serial communication there is just one line dedicated to data transfer and data are transferred bit by bit.

Obviously parallel communication is much faster than serial communication, but the inherent disadvantage is that you need many lines to transfer data, which at times might not be feasible, especially in this case where RF transmitter is to be used for

exchanging data. Therefore on a practical basis to send a byte, 8 RF transmitter/receivers are needed on each side.

(b.) Asynchronous versus synchronous communication

The principle of communication with respect to synchronous and asynchronous communication is same until the receiver wakes up. In both forms of communication the receiver will have an idle-line in a high state before the exchange of data. The similarity ends here.

In synchronous communication the "Master" and the "Slave" communicate between each other, setting their respective clocks. There are two main methods of synchronizing the transmitter and the receiver. The first method uses a unique word as a pulse. When the receiver receives this unique word, it synchronizes itself to the incoming data. The second method involves providing a shift clock signal, the clock signals pulses for every data put on the transmission line.

The "Master" can also be programmed to know that within a certain time it has to receive the data, and whatever comes to it outside this time range is not useful data.

Asynchronous communication as the name suggests does not involve synchronizing the clocks. The data will be sent in a "packet" format. Therefore the first set will be a start bit, the receiver is ready to receive the data and knows the data transmission is over when it receives two stops bits. The second method is to configure transmit (TxD) pin to logic 1, which is an idle signal, when the transmitter is idle. Thus when the receiver gets a falling edge, it samples the bit several times to ensure it is indeed a logic low, and if a valid start bit is received it starts receiving the incoming bits. Synchronous communication is generally faster than asynchronous communication.

But the mode of communication used in this case is asynchronous communication for we are not dealing with just two units communicating with each other, but with hundreds of units trying to synchronize the master clock. Whereas in asynchronous communication each packet contains information for the car it represents.

(c.) Communication protocol

A critical issue in any serial communication interfaces is the protocol involved. For the data transmission between the computer and the "Master" RS - 232 is ideal considering that the distance involved in data transmission is typically less than 50 feet and also that there is no difference in the potential. Also the ease of interfacing is a positive aspect.

Communication between the "Master" and the "Slave" is the key to this whole set-up. For initial testing RS-232 protocol was used for the above-mentioned reasons and also because wires were used to send the data back and forth between the "Master" and the "Slave" instead of using wireless communication as planned. This was to be replaced by wireless communication after successful testing with RS-232 cables.

The second phase of the project involves replacing the wires by RF-Transceivers for the communication between the "Master" and the "Slave." There are two main issues that need to be dealt with here:

a. How do we ensure that the receiver receives correct data and also knows it is the actual data?
b. How do we eliminate noise that might be in the data?

The area of concern is the latter, especially when we are dealing with sensors, transmitters and receivers; noises are bound to corrupt the data. The validity of the method thus rests on how well we can nullify the effect of noise in the data.

1. Packaging the data

The first priority though is to get the data across to the master. Since the possibility of losing data is the highest during the RF transmission, data are sent in form of a packet.

The packet is made of 5 layers, with the actual data sandwiched between the signature bytes. The master is programmed in such a way that it knows each packet

consists of two signature start bytes followed by a data byte, and then by two signature stop bytes as shown in table 4.

Table 4. Sandwich structure of a data packet

START BYTE
SIGNATURE SECOND BYTE
DATA BYTE (FROM A SENSOR)
SIGNATURE FOURTH BYTE
STOP BYTE

If the master finds any packet deviating from this standard format it will ask the slave to resend the packet. Each of the signature bytes will be in the NRZ format (Fig. 41.). A typical example of the signature bytes is shown in table 6.

For example, the receiver knows that it is supposed to receive $FF and $AA as the first two bytes, if it receives these two bytes then it knows that the transmission is smooth and the next byte coming is the data byte. But this process is not complete until the receiver confirms that the last two bytes $FA and $B0 are also present; this ensures that the data is the actual data and not some junk value.

2. Elimination of noise (ensuring the correctness of the data)

The second issue to be tackled is to ensure that the data in the package itself is free of noise. Here noise means the corruption of data by other RF signals. RF signals can easily be corrupted by many factors (Please refer to the section on RF transmission)

One of the most common methods for identifying noise is to take the average of the data [23]. The obvious disadvantage is the wastage of time and the issue of how many times the data has to be resent. Also, is there an optimal number of averaging that will ensure whether the data is free of noise?

Another alternative devised as a part of the project will allow a margin of noise in the data. The master after receiving the data, sends back a data byte for every 10 data byte received. The slave compares this with its data and ascertains the impact of the noise. If it feels the noise is negligible, the communication process goes uninterrupted; otherwise the slave sends a suitable correction factor to the master. Further data is adjusted according to the correction factor, till the noise reduction is seen.

Let us for example assume the first board is sending \$1, \$2, \$3, and so on, but the receiver always receives a value low i.e. \$0, \$1, \$2.When the second board resends the data and the first board knows that its value has been constantly offset, it will send a correction factor for board 1 to add. But this approach does not work if the data are corrupted because some particular object and does not change constantly.

3. Cyclic redundancy check

The method mentioned in the previous section is a very crude idea of the more advanced and established CRC (Cyclic Redundancy Check). CRC is a very well established technique to obtain data reliability. The CRC field is 2 bytes that hold a 16-bit binary value. The transmitting device that appends the CRC to the message calculates the CRC. The receiving device recalculates this value and compares the calculated value with the actual value in the CRC field. It results in an error if they are not equal.

There are several established algorithms to calculate CRC. Two of the methods are described below; the second one is used in our algorithm.

The CRC is calculated by first loading a 16-bit register to 1’s. Then successive bytes of the message are shifted to the present value of the register. The Start, Stop and parity bit do not apply to the CRC.

During the generation of CRC, each 8-bit character is XORed with the register contents; this result is shifted in the direction of the LSB with a zero in the MSB. The LSB is then extracted and examined. If the LSB was a 1, the register I XORed with a preset, fixed value, otherwise this operation does not take place.

This process id repeated until 8 shifts have been performed. After the 8 bits, the next 8-bit value is XORed with the registers value, this process continues for 8 more shifts. The final content will be the CRC value.

The above-mentioned method is for complex packets. Since our data has a much simpler structure it would be enough if all the bytes in the packet are XORed, the last byte in the packet is the CRC. The master upon receiving this packet recalculates the CRC and compares it with the last byte to check for correctness of the data.

(d.) Ports used in 68HC12A4 and 68HC12B32

The 68HC12A4 has eleven 8-bit ports and the 68HC12B32 has eight of them. Most of these ports can be configured as general purpose I/O in different operating modes. None of the ports though are needed for the system being implemented. The only port of interest is the Port S; which is the serial communication interface subsystem in both 68HC12A4 and 68HC12B32. This is the only port that is used. Although other ports like Port F have been used in the experimental stage to verify the data, they are not used in the actual system.

Pins of Port S are denoted as PSx, where ‘x’ stands for 0, 1, 2 or 3 corresponding to the pin numbers. For transmitting data pin PS1 or PS3 is used, for receiving data the RxD pin PS0 or PS2 is used. The 68HC12B32 has just one SCI port functional, i.e. SCI 0.

(e.) SCI subsystem

The SCI subsystem in 68HC12A4 uses the pin PS3 of port S for its transmitting line and pin PS1 for receive. Setting bits in the SCI control register SC1CR2 can enable them. Port S can be used for general-purpose input/output when not in use for serial communication.

The data bytes to be transmitted or are received are first stored in the SCI data register low (SC1DRL). Subsequently the data to be transmitted are read from SC1DRL and also the registers will read the data, which is received. The Initialization of the registers is explained in appendix II.

The SCI systems are capable of sending break signals, which is basically to wake up a receiver; this is an indication to the receiver that it is ready to transmit some data. The receiver can be put back to sleep by setting the RWU bit in SCI control register 2 (SC1CR2). The method of waking up the receiver as explained in the asynchronous communication subsection can be selected by changing the WAKE bit in SC1CR1 register.

(f.) Transmit operations

The transmit operation through the serial port takes place by shifting bytes to the 11-bit transmit shift register. The transmit shift register is already pre-configured with logic low START bit and logic HIGH stop bit. By polling the TDRE register we can determine whether the transmission has taken place, if this bit is set, it implies that the transmit data register is empty and is ready to receive another character. This register can be cleared by reading the SCI control register 1 (SC1CR1) first and then writing to the SCI data register (SC1DRL).

Thus the algorithm of the software to transmit a data byte is similar to the pseudo code below:

STEP 1: Configure the registers (refer to the code in appendix III)

a. Select the baud rate by writing to the baud rate register SC1BDH
b. Select the WAKE up mode and also the length of the data that is transferred
c. Enable transmit, receive and wake up interrupts by modifying SC1CR2
d. Clear all flags

STEP 2: Begin transmission

a. Poll the SC1CR2 register for interrupts and take appropriate action
b. Poll the SC1SR1 register, wait for the TDRE flag to be set
c. If the flag is set, write data to the data register
d. Transmit the next set of data

This can be run continuously, there is no need to clear the flag again, by writing to the data register the TDRE flag is automatically cleared.

(g.) Receive operation

The receive operation through the serial port is somewhat similar in principle to the transmit operation; this takes place by shifting the bytes from the 11-bit transmit shift register. By polling the RDRF register we can determine whether the reception is complete. If this bit is set, it implies that the receive data register is empty and is ready to receive another character. This register can be cleared, by reading the SCI control register SC1CR1 first and then reading the SCI data register (SC1DRL).

The algorithm of the software to receive a data byte is similar to the pseudocode below:

STEP 1: Configure the registers. (refer to the code in appendix III)

a. Select the baud rate by writing to the baud rate register SC1BDH
b. Select the WAKE up mode and also the length of the data that is transferred
c. Enable transmit, receive and wake up interrupts by modifying SC1CR2
d. Clear all flags

STEP 2: Begin transmission

a. Poll the SC1CR2 register for interrupts and take appropriate action
b. Poll the SC1SR1 register, wait for the RDRF flag to be set
c. If the flag is set, then read data from the data register
d. Wait for the next set of data

This can be run continuously, there is no need to clear the flag again, by writing to the data register the TDRE flag is automatically cleared.

I. Data registers used in serial communication

The registers used are shown in Figs. 46 - 50 in appendix II.

J. RF transmission

Different possibilities were considered before going ahead with the selection of RF transceivers for the purpose of data transmission between the master and the slave. The RF transceivers along with the microcontrollers and the sensor-in this case the accelerometer- are enclosed in a casing to protect them from dust and other external hazards. In this chapter we will concentrate on the RF transceivers. Attempt has been made to address all the possible issues regarding why we chose the RF transceivers. What are its advantages? What problems might be encountered?

(a.) Wireless solution

The first question that obviously needs to be addressed is why you need a wireless communication? Especially when the difficulties associated with wireless communications are known.

The system being designed here is supposed to be a low-cost, low-maintenance and easy to use system. Using wireless form of communication eliminates the possibility

of a human error, which might occur when someone with no technical knowledge attempts to check the system and in the process fouls the hardware by disturbing a small wire.

Whereas the entire wireless system will be in a enclosure, so if someone does want to check, all he has to do is remove the enclosure, replace it with another one (by just pushing it to a terminal).

Secondly, in a system with wires the possibilities of mechanical damage like snapping of the wire is very high in a harsh environment. The wires used for the microcontroller boards are very small in diameter, which will be unable to bear the rough environ.

Thirdly, let us assume that the wires are strong enough and do not break. We know that we are typically dealing with a hundred "Slave" controllers trying to send data to a single "Master" controller over distances varying from 800-1,000 ft. The first issue here is that we do not have enough serial ports. Secondly transmitting accurate data over 1000 ft through wires is not a very viable option.

Compared to the problems posed by the system with wires, the advantage of using a wireless system outweighs its disadvantages.

(b.) Selection of RF transceivers

Different types of waves like the ultrasonic waves; infrared waves and radio frequency waves can achieve data transmission by wireless means. But the first two waves (ultrasound and infrared) are absorbable by various objects like a human body etc. The RF waves on the other hand can travel through objects.

Thus when you are looking at sending data over several railcars, you have to send it through a means that will pass through obstacles.

The other advantage of a RF transmitter is that it requires very small amount of power for producing the waves. The RF transmitter we have chosen- Glolab TM1V/RM1V- requires just 5 volts, which is the same as the power needed to run not

just our microcontroller board but also the accelerometer. Thus the task of putting the whole thing into a single unit is made a lot easier.

A digital radio transmission allows a narrow band to carry a large amount of data and enables the receiver device to have minimum power usage. We can thus increase the shelf life of the system considerable. Since the receiver, transmitter and the sensor all have the same voltage levels, it is a lot easier to decode and analyze the data. This will not only speed up the process but also saves power.

The connections are also very simple and straightforward with RF transmitters. The serial ports of the microcontroller can be used to receive and send data from and to the transceivers.

(c.) Issues in RF transceivers

Using RF transceivers has its share of problems. This section deals with the various problems encountered while using an RF transceiver. The solution to these mentioned problems will be dealt with in the next section. A few of the important issues are listed below.

- RF waves are used in a wide range of devices like cell phones, palm pilots, pagers, GPS, car alarms, audio sets etc. Because of this certain phenomenon like “Intermodulation” and “Heterodyning” can occur. These are a great concern with respect to the receivers. Intermodulation occurs when the receiver receives two different frequencies, these can either add up or try to cancel out (resulting in a difference) and in the process produces new harmonics. Heterodyning is a process in which two similar frequencies interact with the receiver; this results in a whistling noise [24].
- Another critical problem we will definitely face is “Multipath Cancellation”. This occurs when the original RF signal reflects off a surface and combines with the original wave again, this may weaken the signal considerably apart from creating phase related problems [24].

- The distance between the "Slave" and the "Master" is a crucial factor. As the distance increases the signal fades considerably and may weaken, the receiver might even lose the signal completely. This again is due to a combination of factors like interference because of moving objects, high-current electric components etc [24].
- During experimentation, which involves sending a single byte, we observe that the first time the data is sent the receiver is still sleeping and before it wakes up data could be lost. This is because the first time the receiver was still not ready.
- Assuming that the data is being transmitted properly with no absolutely no interference and the data flow is not hindered by any means, what is the surety that the data is not corrupted? What is the guarantee that the data received by the receiver is the real data and is not contaminated by noise?

The above are a few of the issues we have tried to address in the following section.

(d.) Probable solutions

It is attempted in this section to obtain a solution for all of the above-mentioned problems.

1. Intermodulation and heterodyning

There is no real "Solution" to this problem as such. The only way to ensure that the signals do not get mixed up is to obtain a very broad and exclusive frequency bandwidth from the federal regulatory authorities.

The second step is to tune the receiver for the frequencies in a limited bandwidth.

2. Multipath cancellation and distance between the "Master" and "Slave"

These two problems are intertwined. This problem has to do with the placing of the transceiver-microcontroller-sensor enclosure. An optimal place on the railcar has to be chosen that will not only ensure that this set up is away from moving parts but also

from other electrical systems. This should also ensure that any objects do not hinder its path; this is to ensure that there will be no loss of data.

The higher the frequency range the longer will be the distance over which the transceivers can communicate with each other. But as the distance increases, the probability that the waves will reflect off some surface increases. Having more than one master on board can resolve this issue. This will not only reduce workload on a master but also increases the accuracy of the signals transmitted as the distance is considerably shortened.

The probable place would have been on the top of the railcar, but we will have problem in placing the sensors separately and then wiring them to the microcontroller to send the signals from the sensors. Apart from this placing the enclosure on top of the railcar will make it very vulnerable to the high-tension wires running parallel with the tracks. This is an issue that can be dealt with only during the testing stage by trying out various positions on the car body.

3. Awakening the receiver

As mentioned in the previous section we noticed that the first time a data is transmitted the receiver loses the first part of the data. This is because the receiver is idle until it receives the first data byte, but it is too late for it to receive the first stream of bytes.

There can be various approaches to this. The easiest probably will be to send the data twice, so that it is ready to receive the actual data the second time. Another alternative can be to send some junk bytes for a small fraction of a second, just to wake up the receiver. This is generally a high byte.

4. Checking correctness of the data

This issue has been dealt with in the previous chapter. Please refer sections 3.3.a and 3.3.b. But in addition to checking the data, the data will be passed through a signal-processing algorithm to minimize noise from the real data.

Although an attempt has been made to address all of these issues, the data transmission through RF transmission is so vulnerable that it often fails to achieve repeatability. The same set of data transmissions in almost the same settings might fail to execute because of some minor fault like moving objects around it,etc.

CHAPTER VI
LABORATORY TESTING, CONCLUSION AND FUTURE WORK

A. Introduction

This chapter explains the lab set up and the tests carried out to identify the wheel flat. Identifying bearing fault was not considered for this phase of the project because of many constraints. The following are the reasons behind:

- To identify a bearing fault, signal analysis must be done on a larger group of data than that is needed to identify a wheel flat. Selected processors do not have the capability to process a huge set of data.
- Simulating a bearing fault in a laboratory environment is very difficult (so there is no way to validate the algorithm unless access to a test facility is given).
- From the preliminary meeting we had with AAR/TTCI people, it was decided that to identify a bearing fault, an accelerometer would not be sufficient and we would need a special sensor, that should be integrated in the manufacturing process, it should be able to withstand high temperatures and also be robust.

Thus we restricted ourselves to just identifying a wheel flat.

B. Prototype-1 with RS232 cables and Ming RF transceivers

The initial testing was validated with RS - 232 cables that were then replaced by RF transmitters from Ming Corporation. A photo of the same is shown in Fig. 42. There were several problems associated with the Ming RF transmitters, a few of them being:

- Low data transfer rate of 1,200 bps.
- It is not suitable for serial communication (although this is mentioned explicitly, mini projects using the same have been attempted).
- Complex circuitry makes the task of enclosing even more difficult.

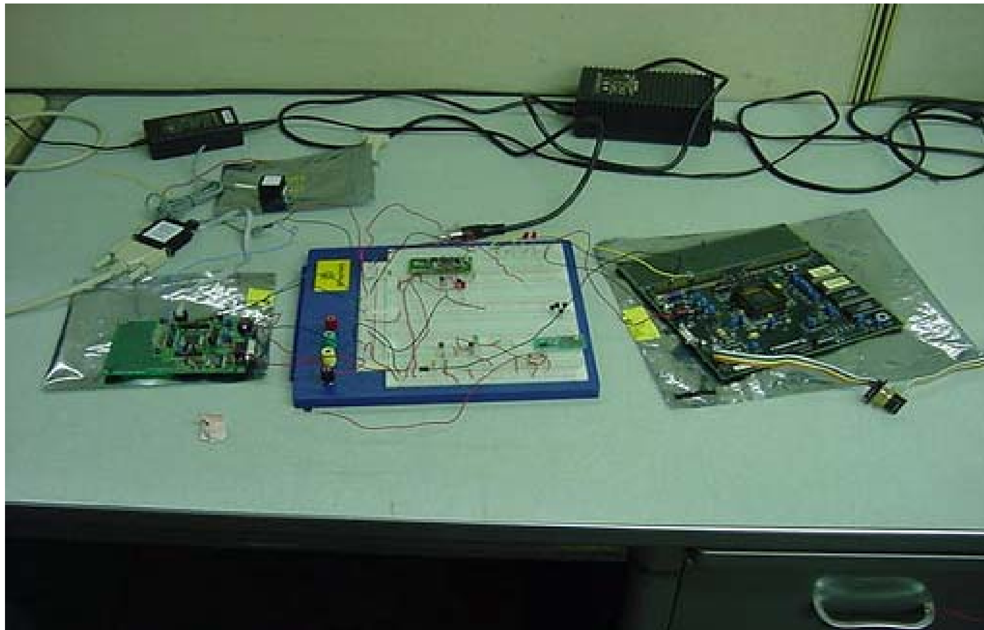


Fig. 42. Setup with Ming transceivers

- Needs amplification of the signals, thus allowing the possibility of amplifying the noise.

For these main issues we decided to look for another RF transmitter for our next prototype.

C. Development of an enclosure

Enclosure for the unit was built out of aluminum alloy. Since space was a constraint, we decided to build a 2-layered enclosure as shown in Fig. 43. The base layer had the microcontroller and the top layer had the RF transmitter and receiver along with the antenna. Power for the unit is a set of batteries housed outside the enclosure with the flexibility of placing it inside too. The hardware selected is now put in an enclosure. The final two stages of building the enclosure are shown in Figs.43 and 44. The final unit is shown in Fig. 44.

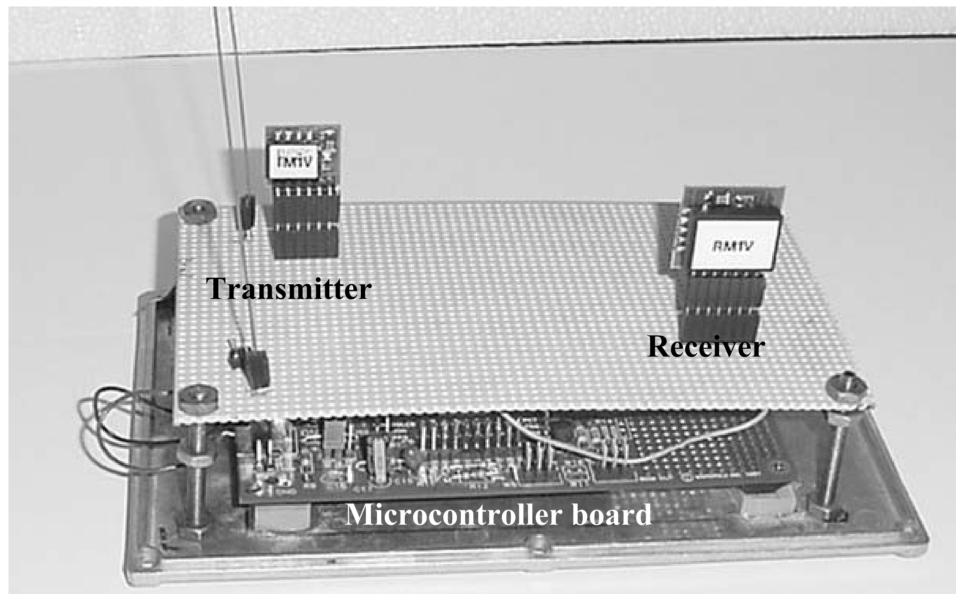


Fig. 43. Penultimate stage of the enclosure

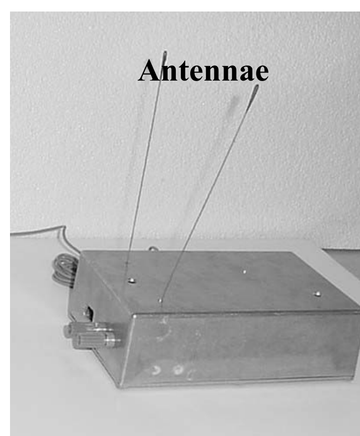


Fig. 44. Enclosed unit

D. Lab set-up

The final set-up for testing in a laboratory consists of

- Computer terminal on both sending and receiving side. On the receiving side we use it to monitor the data and also to initially give commands. The data from the accelerometer can be seen on the screen. Whereas for the sending side, the monitor is only to load the program initially (this can be eliminated in the final prototype).
- 68HC12A4 microcontroller
- 68HC12B32 microcontroller
- 2 sets of Glolab RF transmitter and receiver modules
- ADXL202 accelerometer
- Two sets of battery units to power the controllers and the accelerometer

The lab set up is as shown in Figs. 45 and 46.

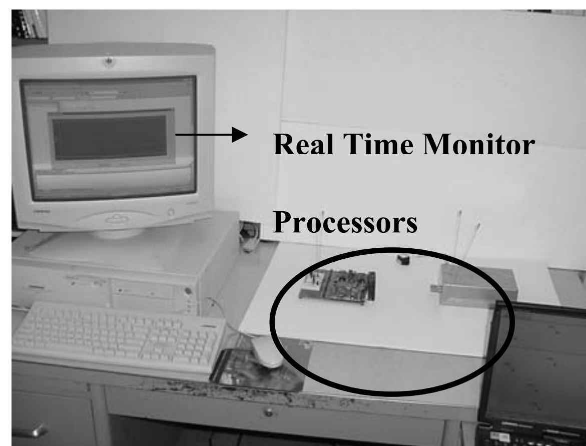


Fig. 45. Final test set-up

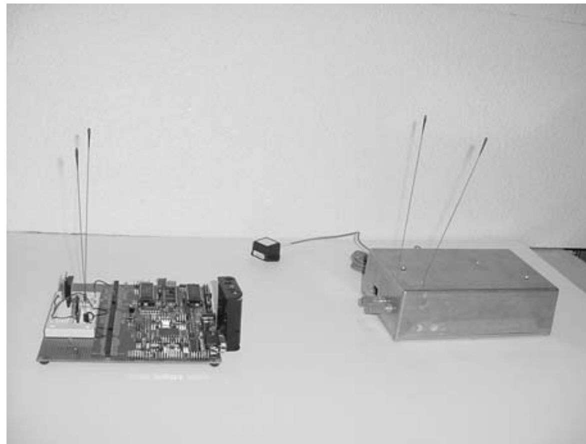


Fig. 46. Enlarged view of the encircled area in Fig. 45

E. Testing procedure

The testing procedure is very simple for a wheel flat; we did not need any test set-up. All we had to check was whether the slave microcontroller transmits the accelerometer data when it registers a value above a preset threshold (any threshold).

The second phase after this is to ensure that the packet is received properly on the receiving side. All this is taken care of by the software listed in appendix IV.

The test was successful; we could see that the slave microcontroller transmits data whenever it encounters a value greater than the threshold. The master controller checks for noise free data and displays it on the screen. Fig. 47 shows a data from an accelerometer. This code is written in VC++ and the other codes predominantly in assembly language and a bit of C. All the codes are attached in appendix IV.

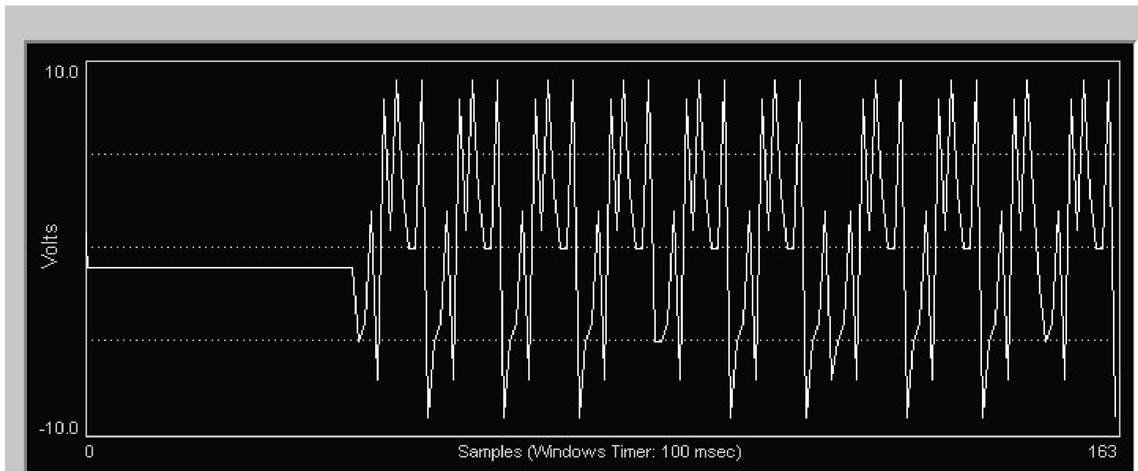


Fig. 47. Real time plot

F. Conclusion

It is clearly seen in the laboratory set-up that two way RF communications in conjunction with a fault-detecting unit is very much a feasible option. This unit is very easy to operate and maintained even by persons without technical know-how. The cost of installing and maintaining this system is very inexpensive in comparison with many of the existing wayside fault detection systems.

This unit also has an added capability of identifying defects in tracks/rails thus may save the railways considerable money and time.

G. Further work

To validate this system we will need access to a real railcar test facility. This apart from letting us know how the system would behave in a real environment would also help us locate a suitable place for this placing this unit on a railcar.

With the selection of a proper sensor the same principle can be used to identify a bearing fault.

The present processing unit (microcontroller) has a lot of unnecessary electronics. The final prototype can have a custom made microcontroller with only the necessary electronics, and this would not only reduce the cost of the system, and also reduce the size of the unit, giving us an option of including many other electronics-not necessarily from this project. Exploring MEMS technology for the same application would not only provide a very convenient and reliable technology but also a cheap option if manufactured in a very large scale.

This idea can be utilized in more ways than those touched upon in this thesis and this system can integrate many other electronics as that mentioned in [25].

REFERENCES

[1] V. K. Garg, Y. H. Tse, "Mathematical models for track/train dynamics," in *Conference on Advanced Techniques in Track Train Dynamics*, Chicago, pp. 223-240, 1977.

[2] G. P. Raymond, D. J. Turcke, O. J. Svec, "Nonlinear analysis of rail track structures," in *Conference on Advanced Techniques in Track Train Dynamics*, Chicago, pp. 299-313, 1977.

[3] C. A. O'Donell, A. K. Carter, "Factors influencing derailment risk," in *Railway Engineering, Systems, and Safety*, IMechE seminar publication, London, pp. 87-96, 1996.

[4] H. C. Choe , Y. Wan, A. K. Chan, "Neural pattern identification of railroad wheel-bearing faults from audible acoustic signals: comparison of FFT, CWT, and DWT features," http://ee.tamu.edu/~akchan/Demo/paper97.pdf, August 2002.

[5] R M Kaul, "Implementating a train protection and warning system," in *Proceedings of the International Conference on Fault Free Infrastructure*, Derby, UK, pp. 159-168, 1999.

[6] N. Kumagai, H. Ishikawa, K. Haga, T. Kigawa, K. Nagasem, "Factors of wheel flats occurrence and preventive measures," in *Proceedings of the Third International Conference on Contact Mechanics and Wear of Rail/Wheel Systems*, Cambridge, UK, pp. 277-287, 1990.

[7] D. H. Stone, "Shattered rim defects in wheels," in *ImechE Seminar Publication on Wheels and Axles*, Bury St Edmunds, pp. 75-84, 2000.

[8] A. Filip, "Train real-time position monitoring trials at Czech railways," in *Structural Integrity and Passenger Safety*, edited by C.A. Brebbia, Southampton, UK, WIT Press, pp. 151-166, 2000.

[9] P. Astrom, "Control Electronics on rail vehicles," in *Proceedings of the ASME/IEEE Joint Railroad Conference*, Atlanta, pp. 107-116, 1992

[10] J. Madejski, J. Grabczyk, “Track & rolling stock quality assurance related tools,” in *Structural Integrity & Passenger Safety*, Southampton, UK, WIT Press, 2000

[11] M. Fetty, Bearing Basics: Introduction to the cartridge tapered Roller Bearing manufactured by Brenco and used for Railroad Journal Bearing applications, http://www.brencoqbs.com/101/tsld001.htm, October 2000.

[12] The Timken Company, Bearings: Recognizing and preventing damage. http://www.timken.com/products/bearings/services/valueadd/prevent.asp, October 2002.

[13] Brenco, Inc.http://www.brencoqbs.com/introduc.htm, October 2002.

[14] T. Snyder, Union Pacific Railroad, Railroad Wheel Testing of Shattered Rim, http://www.ndt.net/article/0798/forum/snyder.htm, October 2002.

[15] G. Vohla, Ch. Linder, H. Bruchli : Corrugation of Railway Wheels.
http://www.ifm.ethz.ch/research/rail-4.html, Swiss Federal Institute of Technology, Zurich, October 2002.

[16] Society for the Preservation of Carter Railroad Resources.
http://www.spcrr.org/current_old.htm, Current Activities Archive, Ardenwood, California, October 2002.

[17] S. Iwnicki, “The Manchester benchmarks for rail vehicle simulation,” in S*upplement to Vehicle System Dynamics*, vol. 31, pp. 4-14, 1999

[18] ”*GENSYS User’s Manual*,” Release 9910, Desolver, Sweden, 1999.

[19] S. D. Iwnicki, S. Grassie, “Prediction of track damage using computer simulation tools,” in *2nd International Workshop on Freight Vehicle Design*, Manchester Metropolitan University, pp. 1-9, April 2001

[20] Swedish National Road and Transport Research Institute (VTI), “High speed railways : alignment optimization with vehicle reactions taken into consideration,” pp. 4-6, 1998.

[21] GENSYS home page. http://home.swipnet.se/gensys/, September 2002.

[22] *CPU 12 Reference Manual: M68HC12 Microcontrollers*, rev6, Motorola, Inc., 2002.

[23] B. Russel, D. Hampson, J. Chun, "Noise elimination and the radon transform," http://utam.geophys.utah.edu/ebooks/gg527/decon/mult.radon6.pdf, October 2002.

[24] P. S. Mundra, T. L. Singal, T. S. Kamal, "Mobile radio network design considering radio frequency interference," in *Proceedings of the ASME/IEEE joint railroad conference*, Atlanta, pp. 13-18, 1992.

[25] H. G. Moody, L. F. Sanders, T. Griffith, "Locomotive electronics system integration architecture," in *Proceedings of the ASME/IEEE joint railroad conference, Atlanta*, pp. 83-85, 1992.

APPENDIX I

A. FLOW CHART FOR THE SYSTEM

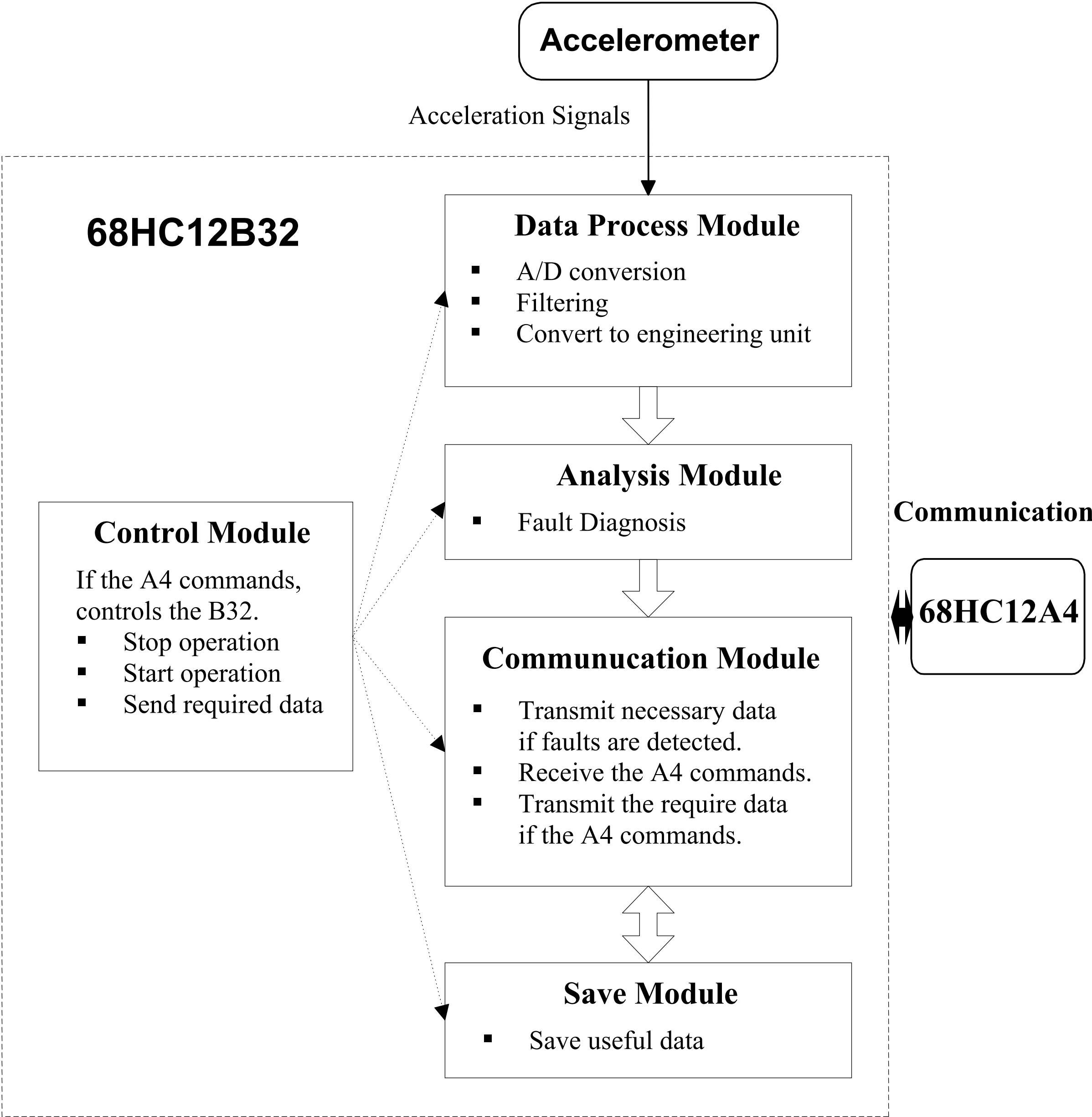


Fig. 48. Flow chart for the system

APPENDIX II

1. Data registers used in the 68HC12 boards

A. SCI control register 1 (SCXCR1)

LOOPS	WOMS	RSRC	M	WAKE	ILT	PE	PT

Fig. 49. SCI control register 1 (SCXCR1)

LOOPS: Setting this bit to '0' ensures that the SCI transmit and receive sections operate normally. The rest are don't care bits

When loops is set to '0', other's are don't care bits.

B. SCI control register 2 (SCXCR2)

TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK

Fig. 50. SCI control register 2 (SCXCR2)

TIE: Transmit Interrupt Enable Bit

Writing logic '1' enables this bit. The SCI interrupt is called whenever this bit is set.

TCIE: Transmit Complete Interrupt Enable Bit

Writing logic '1' enables this bit. Thus whenever a transmission is complete the SCI interrupt is called.

RIE: Receiver Interrupt Enable Bit

Writing logic '1' enables this bit. Thus the SCI interrupt is called whenever the RDRF flag is set.

ILIE: Idle Line Interrupt Enable Bit

Writing logic '1' enables this bit. Thus whenever the IDLE flag is set the SCI interrupt is enabled.

TE: Setting this bit to '1,' the SCI transmit logic is enabled. The TXD pin is dedicated to the transmitter.

RE: Setting this bit to '1,' the SCI receive logic is enabled.

RWU: Receiver wakeup control bit

0 = Normal SCI receiver

1 = Enable the wake up function and inhibits further receiver interrupts. Normally, hardware wakes the receiver by clearing this bit automatically.

SBK: Send break bit

0 = Break generator off

1 = Generate a break code, at least 10 or 11 continuous 0s.

As long as this bit remains set, the transmitter sends 0s. When SBK is changed to 0, the current frame of all 0s is finished before the TxD line goes to idle state.

C. SCI status register 1 (SCXSR1)

TDRE	**TC**	**RDRF**	**IDLE**	**OR**	**NF**	**FE**	**PF**

Fig. 51. SCI status register 1 (SCXSR1)

TDRE: Transmission Data Register Empty flag

If this bit is in logic '0' means that the data register is busy and the transmission is still incomplete. Whereas logic '1' indicates that the data register is now ready to receive new data for transmission.

TC: Transmit Complete flag

If this is at logic '0' it means that the transmitter is busy otherwise the transmitter is idle

RDRF: Receive Data Register Full flag

If this bit is in logic '0' it means the data register is empty and is ready to receive new data. If this is in logic '1' it means the data register is full and cannot receive any further data.

IDLE: Idle Line Detected Flag

This is a key register if we are to utilize the wake up facility in the micro controller. If this bit is at logic '0' the RxD line is active otherwise it is in and idle state.

OR: Overrun flag

New byte is ready to be transferred from the receive shift register to receive data register, but the data register is already full, the data transfer will be inhibited till this is cleared.

NF: Set during the same cycle as the RDRF bit but not set in case of an overrun.

0 = Unanimous decision

1 = Noise on a valid start bit, any of the data bits, or on the stop bit

D. SCI data register low (SC0DRL)

R7T7	R6T6	R5T5	R4T4	R3T3	R2T2	R1T1	R0T0

Fig. 52. SCI data register low (SC0DRL)

R7T7-R0T0: Receive/Transmit data bits 7-0

Reads access the eight bits of the read only SCI receive data register (RDR). Writes accesses to the eight bits of the write-only SCI transmit data register. SC0DRL and

SC0DRH form the 9-bit data word for the SCI. If the SCI is being used with a 7 or 8 bit data word, SC0DRL alone will suffice.

E. SCI baud control registers (SC1BDH/SC1BDL)

The SCI baud rate control registers are used to set the SCI transmission/reception rate.

BTST	**BSPL**	**BRLD**	**SBR12**	**SBR11**	**SBR10**	**SBR9**	**SBR8**

SC1BDH

Fig. 53. SCI baud control register high

SBR7	**SBR6**	**SBR5**	**SBR4**	**SBR3**	**SBR2**	**SBR1**	**SBR0**

SC1BDL

Fig. 54. SCI baud control register low

The baud rate is set using bits SBR [12:0]. The values are determined using the following relationship

SBR = MCLK/ (16*SCI Baud Rate)

where, MCLK is the master clock frequency in Hz.

APPENDIX III

A. ROUTINE FOR TRANSMITTING DATA

```
#include <iob32.h>
#include <stdio.h>

#define TDRE 0x80    /* transmit ready bit */
#define RDRF 0x20    /* receive ready bit*/
#define SIG     0xAA  /*The signature byte to create a CRC*/
#define TRUE    1
#define D_1MS (1000/4)
void delay(unsigned int ms);
int getch();
int C1,C2,C3,R,A,M,S,i,j,RCD,P,Q,T,I,d,k;

getch(){
        R = (ADR1H);
        return(R);
        }

void main(void)
        {

        SC0CR1 = 0X00;
        SC0CR2 = 0X0C;
        SC0BDH = 0x1A1;
        #asm
                LDAA 0X00C4
                STD  0X00C6
        #endasm
```

```
                ATDCTL2 = 0X80;
                ATDCTL3 = 0X00;
                ATDCTL4 = 0X01;
                ATDCTL5 = 0X25;
        A=0xFF;
        I=0x0F;
        P=0x31;
        Q=0x32;
        S=0x33;
        T=0x34;
        M=0x06;
        C1 = P^Q^S^T;

for(j=0;j<1000;j++){
                tran1();
                 }

for(;;){
while (!(SC0SR1 & TDRE))
                ;
        SC0DRL=I;

while (!(SC0SR1 & TDRE))
                ;
        SC0DRL=A;

while (!(SC0SR1 & TDRE))
                ;
```

```
        SC0DRL=P ;

while (!(SC0SR1 & TDRE))
                ;
        SC0DRL= Q;

while (!(SC0SR1 & TDRE))
                ;
        SC0DRL= S;

while (!(SC0SR1 & TDRE))
                ;
        SC0DRL= T;

for(j=0;j<4;j++){
                getch();

        while (!(SC0SR1 & TDRE))
                ;
        SC0DRL= R;
        C2=C2^R;
        }
        C3=C1^C2;

while (!(SC0SR1 & TDRE))
                ;
        SC0DRL= C3;
        recv();
        }
```

```
}

//SUB ROUTINES
// TRANSFERS OF FF's TO AWAKEN THE RECEIVERS AND GIVE SOME TIME
//TO SET UP
tran1(){
        while (!(SC0SR1 & TDRE))
                ;
        SC0DRL=A;
        }

//ACTUAL SIGNAL TRANSFER ROUTINE
recv(){
while (!(SC0SR1 & RDRF))
                ;
        RCD=SC0DRL;
     comp();
        }

comp(){
                if(RCD=C3){
                  while (!(SC0SR1 & TDRE))
                                ;
                      SC0DRL= RCD;
                  }else{
                        recv();
                   }
                }
```

B. SAME CODE IN ASSEMBLY LEVEL LANGUAGE FOR 68HC12

```
;RSTVEC  EQU  $F7FE
;COPCTL  EQU  $0016

REGBASE        EQU $0000
SC0BDH         EQU REGBASE+$C0
SC0CR1         EQU REGBASE+$C2
SC0CR2         EQU REGBASE+$C3
SC0SR1         EQU REGBASE+$C4
SC0DRL         EQU REGBASE+$C7
SC0DRH         EQU REGBASE+$C6
SC0BDL         EQU REGBASE+$C1

RDRF           EQU $20
TDRE           EQU $80
BA12           EQU $1A1


 ORG $0800

;    CLR  COPCTL
;   LDS  #$09FF

                 BSR  INTACL
                 BSR  INTREC
                 BSR  INTSEN
                 BSR  VAL
                 BSR  CRC
```

```
LOOP       BSR  LOAD
           BRA  LOOP

INTACL     LDAA    #$80   ;THIS IS THE INITIALISATION OF THE RESULTS
           STAA    $0062 ;THE A/D CONVERTER IS POWERED UP
           LDAA    #$00
           STAA    $0063
           LDAA    #$01
           STAA    $0064
           LDAA    #$25
           STAA    $0065   ;SET TO READ FROM CONTINUOUSLY
                           ;CHANNEL 6 & REGISTER 6
           RTS

INTREC     MOVB  #$00,SC0CR1
           MOVW  #BA12,SC0BDH
           LDAA  #$0C
           STAA  SC0CR2
           LDAA  SC0SR1
           LDAA  SC0DRL
           RTS

INTSEN     MOVB #$00,SC0CR1
           MOVW #BA12,SC0BDH
           LDAA #$0C
           STAA SC0CR2
           LDAA SC0SR1
           STD  SC0DRH
           RTS
```

```
VAL       LDAB  #$04
          LDX   #$0B01
          LDAA  #$0030
VALS      INCA
          STAA  1,X+
          DECB
          BNE   VALS
          RTS


CRC       LDAA  $0B01
          EORA  $0B02
          EORA  $0B03
          EORA  $0B04
          STAA  $0B00
          RTS

LOAD      LDAA    $0072        ;LOAD THE VALUES FROM RESULT
                               ;REGISTER OF THE A/D CONVERTER
          CMPA    #$7F         ;COMPARE WITH A THRESHOLD VALUE
          BHI     RECORD       ;TRANSMIT IF HIGHER
          BRA     LOAD

RECORD    LDAB    #$0A
          LDX     #$0A00
STORE     LDAA    $0072
          STAA    1,X+
          DECB
```

```
                BNE    STORE
                BSR    CRCR
                BRA    TRANS

CRCR            LDAA   $0A00        ;CALCULATION OF CRC
                EORA   $0A01
                EORA   $0A02
                EORA   $0A03
                EORA   $0A04
                EORA   $0A05
                EORA   $0A06
                EORA   $0A07
                EORA   $0A08
                EORA   $0A09
                EORA   $0A0A

                EORA   $0B00
                STAA   $0A0C
                RTS



TRANS           LDY #$50
SENDH           LDAA #$FF
SENH            TST  SC0SR1
                BPL  SENH
                STAA SC0DRL
                DEY
                BNE  SENDH
```

```
        LDAA #$0F
SENH1   TST  SC0SR1
        BPL  SENH1
        STAA SC0DRL


        LDAA #$FF
SENH2   TST  SC0SR1
        BPL  SENH2
        STAA SC0DRL

        LDAB #$04
        LDX  #$0B01
SEND2   LDAA 1,X+
SEND    TST  SC0SR1
        BPL  SEND
        STAA SC0DRL
        DECB
        BNE  SEND2

        LDAB #$0A
        LDX  #$0A00
SEND3   LDAA 1,X+
SEND4   TST  SC0SR1
        BPL  SEND4
        STAA SC0DRL
        DECB
        BNE  SEND3
```

```
                LDAA $0A0C
SEND5           TST  SC0SR1
                BPL  SEND5
                STAA SC0DRL
                RTS
 ;   ORG   RSTVEC
  ; FDB    $8000
```

C. CODE FOR RECEIVING AND VERIFYING THE CORRECTNESS OF THE DATA

```
REGBASE      EQU $0000
SC0BDH       EQU REGBASE+$C0
SC0CR1       EQU REGBASE+$C2
SC0CR2       EQU REGBASE+$C3
SC0SR1       EQU REGBASE+$C4
SC0DRL       EQU REGBASE+$C7
SC0DRH       EQU REGBASE+$C6
SC0BDL       EQU REGBASE+$C1
SC1BDH       EQU REGBASE+$C8
SC1CR1       EQU REGBASE+$CA
SC1CR2       EQU REGBASE+$CB
SC1SR1       EQU REGBASE+$CC
SC1DRL       EQU REGBASE+$CF
SC1DRH       EQU REGBASE+$CE
SC1BDL       EQU REGBASE+$C9
```

```
RDRF        EQU $20
TDRE        EQU $80
BA12        EQU $1A1


            ORG $0800
            BSR INTREC
            BSR INTSEN
LOOP        BSR REC
            BRA LOOP

INTREC      MOVB  #$00,SC1CR1
            MOVW  #BA12,SC1BDH
            LDAA  #$0C
            STAA  SC1CR2
            LDAA  SC1SR1
            LDAA  SC1DRL
            RTS

INTSEN      MOVB #$00,SC0CR1
            MOVW #BA12,SC0BDH
            LDAA #$0C
            STAA SC0CR2
            LDAA SC0SR1
            STD  SC0DRH
            RTS


REC         LDAA SC1SR1
```

```
            ANDA #RDRF
            BEQ  REC
            LDAA SC1DRL
            CMPA #$0F           ;CHECKING FOR THE FIRST ;SIGNATURE
                                ;BYTE
            BEQ  W2
            BRA  REC

W2          LDAA SC1SR1
            ANDA #RDRF
            BEQ  W2
            LDAA SC1DRL
            CMPA #$FF           ;CHECKING FOR THE SECOND SIGNATURE
                                ; BYTE
            BEQ  RECR
            BRA  REC

RECR        LDAB #$15           ;STORING THE INCOMING DATA BYTES
            LDX  #$0A00
REC1        LDAA SC1SR1
            ANDA #RDRF
            BEQ  REC1
            LDAA SC1DRL
            STAA 1,X+
            DECB
            BNE  REC1

            LDAA    $0A00
            EORA    $0A01         ;RECALCULATING THE CRC
```

```
            EORA    $0A02
            EORA    $0A03
            EORA    $0A04
            EORA    $0A05
            EORA    $0A06
            EORA    $0A07
            EORA    $0A08
            EORA    $0A09
            EORA    $0A0A
            EORA    $0A0B
            EORA    $0A0C
            EORA    $0A0D
            CMPA    $0A0E
            BEQ  SENDR
            BRA  MIST

MIST        LDAA #$EF
            TST  SC0SR1
            BPL  MIST
            STAA SC0DRL
            RTS

SENDR       LDAB #$0A          ;DISPLAY VALUES IF CRC MATCHES
            LDX  #$0A07
SEND2       LDAA 1,X-
SEND        TST  SC0SR1
            BPL  SEND
            STAA SC0DRL
            DECB
```

```
         BNE  SEND2
         LDAA $0A0E
TRC1     TST  SC0SR1
         BPL  TRC1
         STAA SC0DRL

         LDAA $EE
TRC2     TST  SC1SR1
         BPL  TRC2
         STAA SC1DRL

         LDAA $0A0E
TRC3     TST  SC1SR1
         BPL  TRC3
         STAA SC1DRL
         RTS
```

D. Pseudo code for identifying a defective track

1. Wait for signals from all the cars.
2. If all the signals exhibit the same characteristic in a given frame of time.
3. Then calculate the distance where this event occurred and calculate the distance using timer functions
4. It would be around this distance that there would be a defect in the track.

APPENDIX IV

A. OScopeCtrl.cpp : implementation file

This code has been modified from the original versions available on www.codeguru.com as on October' 2002.

```
#include "stdafx.h"
#include "math.h"
#include "OScopeCtrl.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__ ;
#endif
/////////////////////////////////////////////////////////////////////////////
// COScopeCtrl
COScopeCtrl::COScopeCtrl()
{
  m_dPreviousPosition =   0.0 ;
  m_nYDecimals = 3 ;
  m_dLowerLimit = -10.0 ;
  m_dUpperLimit =  10.0 ;
  m_dRange      =  m_dUpperLimit - m_dLowerLimit ;
  m_nShiftPixels     = 4 ;
  m_nHalfShiftPixels = m_nShiftPixels/2 ;                     // protected
  m_nPlotShiftPixels = m_nShiftPixels + m_nHalfShiftPixels ;  // protected

  m_crBackColor  = RGB(  0,   0,   0) ;  // see also SetBackgroundColor
  m_crGridColor  = RGB(  0, 255, 255) ;  // see also SetGridColor
```

```
  m_crPlotColor  = RGB(255, 255, 255) ;  // see also SetPlotColor
  m_penPlot.CreatePen(PS_SOLID, 0, m_crPlotColor) ;
  m_brushBack.CreateSolidBrush(m_crBackColor) ;
  m_strXUnitsString.Format("Samples") ;  // can also be set with SetXUnits
  m_strYUnitsString.Format("Y units") ;  // can also be set with SetYUnits
  m_pbitmapOldGrid = NULL ;
  m_pbitmapOldPlot = NULL ;
}  // COScopeCtrl
/////////////////////////////////////////////////////////////////////////////
COScopeCtrl::~COScopeCtrl()
{
  if (m_pbitmapOldGrid != NULL)
    m_dcGrid.SelectObject(m_pbitmapOldGrid) ;
  if (m_pbitmapOldPlot != NULL)
    m_dcPlot.SelectObject(m_pbitmapOldPlot) ;
} // ~COScopeCtrl

BEGIN_MESSAGE_MAP(COScopeCtrl, CWnd)
  //{{AFX_MSG_MAP(COScopeCtrl)
  ON_WM_PAINT()
  ON_WM_SIZE()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// COScopeCtrl message handlers
/////////////////////////////////////////////////////////////////////////////
BOOL COScopeCtrl::Create(DWORD dwStyle, const RECT& rect,
                         CWnd* pParentWnd, UINT nID)
{
```

```
  BOOL result ;
static CString className = AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW) ;

  result = CWnd::CreateEx(WS_EX_CLIENTEDGE | WS_EX_STATICEDGE,
                          className, NULL, dwStyle,
                          rect.left, rect.top, rect.right-rect.left, rect.bottom-rect.top,
                          pParentWnd->GetSafeHwnd(), (HMENU)nID) ;
  if (result != 0)
    InvalidateCtrl() ;
  return result ;
} // Create
/////////////////////////////////////////////////////////////////////////////
void COScopeCtrl::SetRange(double dLower, double dUpper, int nDecimalPlaces)
{
  ASSERT(dUpper > dLower) ;
  m_dLowerLimit     = dLower ;
  m_dUpperLimit     = dUpper ;
  m_nYDecimals      = nDecimalPlaces ;
  m_dRange          = m_dUpperLimit - m_dLowerLimit ;
  m_dVerticalFactor = (double)m_nPlotHeight / m_dRange ;
  InvalidateCtrl() ;
}  // SetRange
/////////////////////////////////////////////////////////////////////////////
void COScopeCtrl::SetXUnits(CString string)
{
  m_strXUnitsString = string ;
  InvalidateCtrl() ;
}  // SetXUnits
/////////////////////////////////////////////////////////////////////////////
```

```
void COScopeCtrl::SetYUnits(CString string)
{
  m_strYUnitsString = string ;

  InvalidateCtrl() ;
}  // SetYUnits
/////////////////////////////////////////////////////////////////////////////
void COScopeCtrl::SetGridColor(COLORREF color)
{
  m_crGridColor = color ;
  InvalidateCtrl() ;
}  // SetGridColor
/////////////////////////////////////////////////////////////////////////////
void COScopeCtrl::SetPlotColor(COLORREF color)
{
  m_crPlotColor = color ;
  m_penPlot.DeleteObject() ;
  m_penPlot.CreatePen(PS_SOLID, 0, m_crPlotColor) ;
  InvalidateCtrl() ;
}  // SetPlotColor
/////////////////////////////////////////////////////////////////////////////
void COScopeCtrl::SetBackgroundColor(COLORREF color)
{
  m_crBackColor = color ;
  m_brushBack.DeleteObject() ;
  m_brushBack.CreateSolidBrush(m_crBackColor) ;
  InvalidateCtrl() ;
}  // SetBackgroundColor
/////////////////////////////////////////////////////////////////////////////
```

```
void COScopeCtrl::InvalidateCtrl()
{
 int i ;
 int nCharacters ;
 int nTopGridPix, nMidGridPix, nBottomGridPix ;

 CPen *oldPen ;
 CPen solidPen(PS_SOLID, 0, m_crGridColor) ;
 CFont axisFont, yUnitFont, *oldFont ;
 CString strTemp ;
 CClientDC dc(this) ;

 if (m_dcGrid.GetSafeHdc() == NULL)
 {
  m_dcGrid.CreateCompatibleDC(&dc) ;
  m_bitmapGrid.CreateCompatibleBitmap(&dc, m_nClientWidth, m_nClientHeight) ;
  m_pbitmapOldGrid = m_dcGrid.SelectObject(&m_bitmapGrid) ;
 }
 m_dcGrid.SetBkColor (m_crBackColor) ;
 m_dcGrid.FillRect(m_rectClient, &m_brushBack) ;

 nCharacters = abs((int)log10(fabs(m_dUpperLimit))) ;
 nCharacters = max(nCharacters, abs((int)log10(fabs(m_dLowerLimit)))) ;
 nCharacters = nCharacters + 4 + m_nYDecimals ;
 m_rectPlot.left = m_rectClient.left + 6*(nCharacters) ;
 m_nPlotWidth    = m_rectPlot.Width() ;
 oldPen = m_dcGrid.SelectObject (&solidPen) ;
 m_dcGrid.MoveTo (m_rectPlot.left, m_rectPlot.top) ;
 m_dcGrid.LineTo (m_rectPlot.right+1, m_rectPlot.top) ;
```

```
m_dcGrid.LineTo (m_rectPlot.right+1, m_rectPlot.bottom+1) ;
m_dcGrid.LineTo (m_rectPlot.left, m_rectPlot.bottom+1) ;
m_dcGrid.LineTo (m_rectPlot.left, m_rectPlot.top) ;
m_dcGrid.SelectObject (oldPen) ;
nMidGridPix    = (m_rectPlot.top + m_rectPlot.bottom)/2 ;
nTopGridPix    = nMidGridPix - m_nPlotHeight/4 ;
nBottomGridPix = nMidGridPix + m_nPlotHeight/4 ;

for (i=m_rectPlot.left; i<m_rectPlot.right; i+=4)
{
  m_dcGrid.SetPixel (i, nTopGridPix,    m_crGridColor) ;
  m_dcGrid.SetPixel (i, nMidGridPix,    m_crGridColor) ;
  m_dcGrid.SetPixel (i, nBottomGridPix, m_crGridColor) ;
}
axisFont.CreateFont (14, 0, 0, 0, 300,
                FALSE, FALSE, 0, ANSI_CHARSET,
                OUT_DEFAULT_PRECIS,
                CLIP_DEFAULT_PRECIS,
                DEFAULT_QUALITY,
                DEFAULT_PITCH|FF_SWISS, "Arial") ;
yUnitFont.CreateFont (14, 0, 900, 0, 300,
                FALSE, FALSE, 0, ANSI_CHARSET,
                OUT_DEFAULT_PRECIS,
                CLIP_DEFAULT_PRECIS,
                DEFAULT_QUALITY,
                DEFAULT_PITCH|FF_SWISS, "Arial") ;
oldFont = m_dcGrid.SelectObject(&axisFont) ;
m_dcGrid.SetTextColor (m_crGridColor) ;
m_dcGrid.SetTextAlign (TA_RIGHT|TA_TOP) ;
```

```
strTemp.Format ("%.*lf", m_nYDecimals, m_dUpperLimit) ;
m_dcGrid.TextOut (m_rectPlot.left-4, m_rectPlot.top, strTemp) ;

m_dcGrid.SetTextAlign (TA_RIGHT|TA_BASELINE) ;
strTemp.Format ("%.*lf", m_nYDecimals, m_dLowerLimit) ;
m_dcGrid.TextOut (m_rectPlot.left-4, m_rectPlot.bottom, strTemp) ;
m_dcGrid.SetTextAlign (TA_LEFT|TA_TOP) ;
m_dcGrid.TextOut (m_rectPlot.left, m_rectPlot.bottom+4, "0") ;
m_dcGrid.SetTextAlign (TA_RIGHT|TA_TOP) ;
strTemp.Format ("%d", m_nPlotWidth/m_nShiftPixels) ;
m_dcGrid.TextOut (m_rectPlot.right, m_rectPlot.bottom+4, strTemp) ;

m_dcGrid.SetTextAlign (TA_CENTER|TA_TOP) ;
m_dcGrid.TextOut ((m_rectPlot.left+m_rectPlot.right)/2,
            m_rectPlot.bottom+4, m_strXUnitsString) ;
m_dcGrid.SelectObject(oldFont) ;
oldFont = m_dcGrid.SelectObject(&yUnitFont) ;
m_dcGrid.SetTextAlign (TA_CENTER|TA_BASELINE) ;
m_dcGrid.TextOut ((m_rectClient.left+m_rectPlot.left)/2,
            (m_rectPlot.bottom+m_rectPlot.top)/2, m_strYUnitsString) ;
m_dcGrid.SelectObject(oldFont) ;

if (m_dcPlot.GetSafeHdc() == NULL)
{
  m_dcPlot.CreateCompatibleDC(&dc) ;
  m_bitmapPlot.CreateCompatibleBitmap(&dc, m_nClientWidth, m_nClientHeight) ;
  m_pbitmapOldPlot = m_dcPlot.SelectObject(&m_bitmapPlot) ;
}
m_dcPlot.SetBkColor (m_crBackColor) ;
```

```
  m_dcPlot.FillRect(m_rectClient, &m_brushBack) ;
  InvalidateRect(m_rectClient) ;

} // InvalidateCtrl
/////////////////////////////////////////////////////////////////////////////
double COScopeCtrl::AppendPoint(double dNewPoint)
{
  double dPrevious ;
  dPrevious = m_dCurrentPosition ;
  m_dCurrentPosition = dNewPoint ;
  DrawPoint() ;

  Invalidate() ;
  return dPrevious ;
} // AppendPoint
 /////////////////////////////////////////////////////////////////////////////
void COScopeCtrl::OnPaint()
{
  CPaintDC dc(this) ;  // device context for painting
  CDC memDC ;
  CBitmap memBitmap ;
  CBitmap* oldBitmap ; // bitmap originally found in CMemDC
  // no real plotting work is performed here,
  // just putting the existing bitmaps on the client
  // to avoid flicker, establish a memory dc, draw to it

  memDC.CreateCompatibleDC(&dc) ;
  memBitmap.CreateCompatibleBitmap(&dc, m_nClientWidth, m_nClientHeight) ;
  oldBitmap = (CBitmap *)memDC.SelectObject(&memBitmap) ;
```

```
  if (memDC.GetSafeHdc() != NULL)
  {
    // first drop the grid on the memory dc
    memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                 &m_dcGrid, 0, 0, SRCCOPY) ;
    // now add the plot on top as a "pattern" via SRCPAINT.
    // works well with dark background and a light plot
    memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                 &m_dcPlot, 0, 0, SRCPAINT) ;  //SRCPAINT
    // finally send the result to the display
    dc.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
              &memDC, 0, 0, SRCCOPY) ;
  }

  memDC.SelectObject(oldBitmap) ;

} // OnPaint

/////////////////////////////////////////////////////////////////////////////
void COScopeCtrl::DrawPoint()
{
  // this does the work of "scrolling" the plot to the left
  // and appending a new data point all of the plotting is
  // directed to the memory based bitmap associated with m_dcPlot
  // the will subsequently be BitBlt'd to the client in OnPaint

  int currX, prevX, currY, prevY ;
```

```
CPen *oldPen ;
CRect rectCleanUp ;

if (m_dcPlot.GetSafeHdc() != NULL)
{
  // shift the plot by BitBlt'ing it to itself
  // note: the m_dcPlot covers the entire client
  //       but we only shift bitmap that is the size
  //       of the plot rectangle
  // grab the right side of the plot (exluding m_nShiftPixels on the left)
  // move this grabbed bitmap to the left by m_nShiftPixels

  m_dcPlot.BitBlt(m_rectPlot.left, m_rectPlot.top+1,
                  m_nPlotWidth, m_nPlotHeight, &m_dcPlot,
                  m_rectPlot.left+m_nShiftPixels, m_rectPlot.top+1,
                  SRCCOPY) ;

  // establish a rectangle over the right side of plot
  // which now needs to be cleaned up proir to adding the new point
  rectCleanUp = m_rectPlot ;
  rectCleanUp.left  = rectCleanUp.right - m_nShiftPixels ;

  // fill the cleanup area with the background
  m_dcPlot.FillRect(rectCleanUp, &m_brushBack) ;

  // draw the next line segement

  // grab the plotting pen
  oldPen = m_dcPlot.SelectObject(&m_penPlot) ;
```

```
// move to the previous point
prevX = m_rectPlot.right-m_nPlotShiftPixels ;
prevY = m_rectPlot.bottom -
        (long)((m_dPreviousPosition - m_dLowerLimit) * m_dVerticalFactor) ;
m_dcPlot.MoveTo (prevX, prevY) ;

// draw to the current point
currX = m_rectPlot.right-m_nHalfShiftPixels ;
currY = m_rectPlot.bottom -
        (long)((m_dCurrentPosition - m_dLowerLimit) * m_dVerticalFactor) ;
m_dcPlot.LineTo (currX, currY) ;

// restore the pen
m_dcPlot.SelectObject(oldPen) ;

// if the data leaks over the upper or lower plot boundaries
// fill the upper and lower leakage with the background
// this will facilitate clipping on an as needed basis
// as opposed to always calling IntersectClipRect
if ((prevY <= m_rectPlot.top) || (currY <= m_rectPlot.top))
  m_dcPlot.FillRect(CRect(prevX, m_rectClient.top, currX+1, m_rectPlot.top+1),
&m_brushBack) ;
if ((prevY >= m_rectPlot.bottom) || (currY >= m_rectPlot.bottom))
  m_dcPlot.FillRect(CRect(prevX, m_rectPlot.bottom+1, currX+1,
m_rectClient.bottom+1), &m_brushBack) ;

// store the current point for connection to the next point
m_dPreviousPosition = m_dCurrentPosition ;
```

```
  }

} // end DrawPoint

/////////////////////////////////////////////////////////////////////////////
void COScopeCtrl::OnSize(UINT nType, int cx, int cy)
{
  CWnd::OnSize(nType, cx, cy) ;

  // NOTE: OnSize automatically gets called during the setup of the control

  GetClientRect(m_rectClient) ;

  // set some member variables to avoid multiple function calls
  m_nClientHeight = m_rectClient.Height() ;
  m_nClientWidth  = m_rectClient.Width() ;

  // the "left" coordinate and "width" will be modified in
  // InvalidateCtrl to be based on the width of the y axis scaling
  m_rectPlot.left   = 20 ;
  m_rectPlot.top    = 10 ;
  m_rectPlot.right  = m_rectClient.right-10 ;
  m_rectPlot.bottom = m_rectClient.bottom-25 ;

  // set some member variables to avoid multiple function calls
  m_nPlotHeight = m_rectPlot.Height() ;
  m_nPlotWidth  = m_rectPlot.Width() ;
```

```
  // set the scaling factor for now, this can be adjusted
  // in the SetRange functions
  m_dVerticalFactor = (double)m_nPlotHeight / m_dRange ;

} // OnSize


/////////////////////////////////////////////////////////////////////////////
void COScopeCtrl::Reset()
{
  // to clear the existing data (in the form of a bitmap)
  // simply invalidate the entire control
  InvalidateCtrl() ;
}
```

B. TestOScope.cpp : Defines the class behaviors for the application.

```
#include "stdafx.h"
#include "TestOScope.h"
#include "TestOScopeDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/* ------------------------------------------------------------------- */
#ifdef __BORLANDC__
#pragma hdrstop            // borland specific
```

```
#include <condefs.h>
#pragma argsused
USEUNIT("Tserial_event.cpp");
#endif
//---------------------------------------------------------------------------
#include "conio.h"
#include "Tserial_event.h"
/////////////////////////////////////////////////////////////////////////////
// CTestOScopeApp

BEGIN_MESSAGE_MAP(CTestOScopeApp, CWinApp)
  //{{AFX_MSG_MAP(CTestOScopeApp)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //    DO NOT EDIT what you see in these blocks of generated code!
  //}}AFX_MSG
  ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CTestOScopeApp construction

CTestOScopeApp::CTestOScopeApp()
{
  // TODO: add construction code here,
  // Place all significant initialization in InitInstance
}
/////////////////////////////////////////////////////////////////////////////
// The one and only CTestOScopeApp object
//friend class CTestOScopeDlg;
CTestOScopeApp theApp;
```

```
CTestOScopeDlg* dlg = NULL;
DWORD* lpThreadId = NULL;


// CTestOScopeApp initialization
/* ======================================================== */
/* ===============  OnCharArrival    ===================== */
/* ======================================================== */
void CTestOScopeApp::OnDataArrival(int size, char *buffer)
{
   if ((size>0) && (buffer!=0))
   {
      buffer[size] = 0;
      printf("OnDataArrival: %s ",buffer);
                  dlg->m_OScopeCtrl.AppendPoint((double)atoi(buffer));
   }
}

/* ======================================================== */
/* ===============  OnCharArrival    ===================== */
/* ======================================================== */
void SerialEventManager(uint32 object, uint32 event)
{
   char *buffer;
   int   size;
   Tserial_event *com;

   com = (Tserial_event *) object;
   if (com!=0)
```

```
{
   switch(event)
   {
      case  SERIAL_CONNECTED  :
                        //printf("Connected ! \n");

      ::AfxMessageBox("Connected!");
                        break;
      case  SERIAL_DISCONNECTED  :
                        printf("Disonnected ! \n");
                                                                   break;
      case  SERIAL_DATA_SENT  :
                        //printf("Data sent ! \n");

      ::AfxMessageBox("Data Sent!");
                        break;
      case  SERIAL_RING       :
                        printf("DRING ! \n");
                        break;
      case  SERIAL_CD_ON      :
                        printf("Carrier Detected ! \n");
                        break;
      case  SERIAL_CD_OFF     :
                        printf("No more carrier ! \n");
                        break;
      case  SERIAL_DATA_ARRIVAL  :
                        size   = com->getDataInSize();
                        buffer = com->getDataInBuffer();
                        theApp.OnDataArrival(size, buffer);
```

```
                                com->dataHasBeenRead();
                                break;
                }
            }
}


BOOL CTestOScopeApp::InitInstance()
{
  DWORD WINAPI Graph_Plotter(LPVOID);
  AfxEnableControlContainer();
#ifdef _AFXDLL
  Enable3dControls();      // Call this when using MFC in a shared DLL
#else
  Enable3dControlsStatic();  // Call this when linking to MFC statically
#endif
  dlg = new CTestOScopeDlg;
  m_pMainWnd = dlg;

  HANDLE h_graph_plotter = ::CreateThread(
                        NULL,  // pointer to security attributes
                        0,     // initial thread stack size
                        Graph_Plotter,// pointer to thread function
                        dlg,         // argument for new thread
                        0,           // creation flags
                        lpThreadId    // pointer to receive thread ID
            );

  Send_recv_COM_data();
```

```
 WaitForSingleObject(h_graph_plotter, INFINITE);
 return TRUE;

}

void CTestOScopeApp::Send_recv_COM_data()
{
	//int		c;
	int		erreur;
	//char		txt[32];
	Tserial_event *com;

	com = new Tserial_event();
	if (com!=0)
	{
		com->setManager(SerialEventManager);
		erreur = com->connect("COM1", 19200, SERIAL_PARITY_NONE, 8,
true);
		if (!erreur)
		{
			::AfxMessageBox("Connected!");
			/*com->sendData("Hello World",11);
			com->setRxSize(5);

			// ------------------
			do
			{
				c = getch();
				printf("_%c",c);
```

```
                    txt[0] = c;
                    com->sendData(txt, 1);
                    com->setRxSize(1);
                }
                while (c!=32);*/

            }
            else
                AfxMessageBox("ERROR : com->connect");
            // ------------------
            // com->disconnect();

            // ------------------
            // delete com;
            // com = 0;
        }
}


DWORD WINAPI Graph_Plotter(LPVOID lpParameter)   // thread data
{

  CTestOScopeDlg* dlg = (CTestOScopeDlg*) lpParameter;

  int nResponse = dlg->DoModal();

  if (nResponse == IDOK)
  {
    // TODO: Place code here to handle when the dialog is
```

```
    //  dismissed with OK
  }
  else if (nResponse == IDCANCEL)


  return 0;
}
```

C. TestOScopeDlg.cpp : implementation file

```
#include "stdafx.h"
#include <stdlib.h>
#include "TestOScope.h"
#include "TestOScopeDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About
class CAboutDlg : public CDialog
{
public:
  CAboutDlg();
// Dialog Data
  //{{AFX_DATA(CAboutDlg)
  enum { IDD = IDD_ABOUTBOX };
```

```
  //}}AFX_DATA
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CAboutDlg)
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
  //}}AFX_VIRTUAL
// Implementation
protected:
  //{{AFX_MSG(CAboutDlg)
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
  //{{AFX_DATA_INIT(CAboutDlg)
  //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CAboutDlg)
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
  //{{AFX_MSG_MAP(CAboutDlg)
  //}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CTestOScopeDlg dialog

CTestOScopeDlg::CTestOScopeDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CTestOScopeDlg::IDD, pParent)
{
  //{{AFX_DATA_INIT(CTestOScopeDlg)
    // NOTE: the ClassWizard will add member initialization here
  //}}AFX_DATA_INIT
  // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
  m_bStartStop = FALSE ;
  srand( (unsigned)time( NULL ) );

}

void CTestOScopeDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CTestOScopeDlg)
    // NOTE: the ClassWizard will add DDX and DDV calls here
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CTestOScopeDlg, CDialog)
  //{{AFX_MSG_MAP(CTestOScopeDlg)
  ON_WM_SYSCOMMAND()
  ON_WM_PAINT()
```

```
 ON_WM_QUERYDRAGICON()
 //ON_WM_TIMER()
 //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CTestOScopeDlg message handlers

BOOL CTestOScopeDlg::OnInitDialog()
{
 CDialog::OnInitDialog();

 // Add "About..." menu item to system menu.

 // IDM_ABOUTBOX must be in the system command range.
 ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
 ASSERT(IDM_ABOUTBOX < 0xF000);

 CMenu* pSysMenu = GetSystemMenu(FALSE);
 if (pSysMenu != NULL)
 {
  CString strAboutMenu;
  strAboutMenu.LoadString(IDS_ABOUTBOX);
  if (!strAboutMenu.IsEmpty())
  {
   pSysMenu->AppendMenu(MF_SEPARATOR);
   pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
  }
 }
```

```
  // Set the icon for this dialog.  The framework does this automatically
  //  when the application's main window is not a dialog
  SetIcon(m_hIcon, TRUE);      // Set big icon
  SetIcon(m_hIcon, FALSE);    // Set small icon
    // TODO: Add extra initialization here
  // determine the rectangle for the control
  CRect rect;
  GetDlgItem(IDC_OSCOPE)->GetWindowRect(rect) ;
  ScreenToClient(rect) ;

  // create the control
  m_OScopeCtrl.Create(WS_VISIBLE | WS_CHILD, rect, this) ;

  // customize the control
  m_OScopeCtrl.SetRange(-10.0, 10.0, 1) ;
  m_OScopeCtrl.SetYUnits("Volts") ;
  m_OScopeCtrl.SetXUnits("Samples (Windows Timer: 100 msec)") ;
  m_OScopeCtrl.SetBackgroundColor(RGB(0, 0, 64)) ;
  m_OScopeCtrl.SetGridColor(RGB(192, 192, 255)) ;
  m_OScopeCtrl.SetPlotColor(RGB(255, 255, 255)) ;

  return TRUE;  // return TRUE  unless you set the focus to a control
}

void CTestOScopeDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
  if ((nID & 0xFFF0) == IDM_ABOUTBOX)
  {
    CAboutDlg dlgAbout;
```

```
    dlgAbout.DoModal();
  }
  else
  {
    CDialog::OnSysCommand(nID, lParam);
  }
}

void CTestOScopeDlg::OnPaint()
{
  if (IsIconic())
  {
    CPaintDC dc(this); // device context for painting

    SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

    // Center icon in client rectangle
    int cxIcon = GetSystemMetrics(SM_CXICON);
    int cyIcon = GetSystemMetrics(SM_CYICON);
    CRect rect;
    GetClientRect(&rect);
    int x = (rect.Width() - cxIcon + 1) / 2;
    int y = (rect.Height() - cyIcon + 1) / 2;

    // Draw the icon
    dc.DrawIcon(x, y, m_hIcon);
  }
  else
  {
```

```
    CDialog::OnPaint();
  }
}

HCURSOR CTestOScopeDlg::OnQueryDragIcon()
{
  return (HCURSOR) m_hIcon;
}

void CTestOScopeDlg::OnRunstop()
{
  // TODO: Add your control notification handler code here
  m_bStartStop ^= TRUE;

  if (m_bStartStop)
    SetTimer(1,100,NULL);
  else
    KillTimer(1);

}

/*void CTestOScopeDlg::OnTimer(UINT nIDEvent)
{
  double nRandom=0;

 // nRandom = -5.0 + 10.0*rand()/(double)RAND_MAX;

  // append the new value to the plot
  m_OScopeCtrl.AppendPoint(nRandom);
```

```
  CDialog::OnTimer(nIDEvent);
}*/

void CTestOScopeDlg::OnOK()
{
  m_bStartStop ^= TRUE;

  if (m_bStartStop)
    SetTimer(1,100,NULL);
  else
    KillTimer(1);

}

void CTestOScopeDlg::OnCancel()
{
  if (!m_bStartStop)
    KillTimer(1) ;

  CDialog::OnCancel();
}
```

D. Tserial_event.cpp

```
/* -------------------------------------------------------------------- */

#define STRICT
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <process.h>
#include <conio.h>
#include <windows.h>


#include "Tserial_event.h"

#define SIG_POWER_DOWN     0
#define SIG_READER         1
#define SIG_READ_DONE      2    // data received has been read
#define SIG_WRITER         3
#define SIG_DATA_TO_TX     4    // data waiting to be sent
#define SIG_MODEM_EVENTS   5
#define SIG_MODEM_CHECKED  6

void Tserial_event_thread_start(void *arg);

typedef unsigned (WINAPI *PBEGINTHREADEX_THREADFUNC) (LPVOID
lpThreadParameter);
typedef unsigned *PBEGINTHREADEX_THREADID;

/* -------------------------------------------------------------------- */
/* ---------------------  Tserial_event_thread_start  ------------------- */
/* -------------------------------------------------------------------- */
/**
   This function is not part of the Tserial_event object. It is simply used
   to start the thread from an external point of the object.
```

```
*/
void Tserial_event_thread_start(void *arg)
{
    class Tserial_event *serial_unit;

    serial_unit = (Tserial_event *) arg;

    if (serial_unit!=0)
        serial_unit->run();
}

/* -------------------------    Tserial_event ------------------------- */
Tserial_event::Tserial_event()
{
    int i;

    ready           = false;
    parityMode      = SERIAL_PARITY_NONE;
    port[0]         = 0;
    rate            = 0;
    threadid        = 0;
    serial_handle   = INVALID_HANDLE_VALUE;
    thread_handle   = 0;
    owner           = 0;
    tx_in_progress  = 0;
    rx_in_progress  = 0;
    max_rx_size     = 1;
    tx_size         = 0;
    received_size   = 0;
```

```
    check_modem      = false;

    manager          = 0;

    /* ---------------------------------------------------------- */
    // creating Events for the different sources
    for (i=0; i<SERIAL_SIGNAL_NBR; i++)
    {
        if ((i==SIG_READER) || (i==SIG_WRITER) || (i==SIG_MODEM_EVENTS))
            serial_events[i] = CreateEvent(NULL, TRUE, FALSE, NULL);  // Manual Reset
        else
            serial_events[i] = CreateEvent(NULL, FALSE, FALSE, NULL); // Auto reset
    }
}

/* -------------------------------------------------------------- */
/* -------------------------    ~Tserial_event ---------------------- */
/* -------------------------------------------------------------- */
Tserial_event::~Tserial_event()
{
    int i;

    if (thread_handle!=0)
        WaitForSingleObject(thread_handle, 2000);
    thread_handle = 0;

    /* ---------------------------------------------------- */
    for (i=0; i<SERIAL_SIGNAL_NBR; i++)         // deleting the events
    {
```

```
        if (serial_events[i]!=INVALID_HANDLE_VALUE)
            CloseHandle(serial_events[i]);
        serial_events[i] = INVALID_HANDLE_VALUE;
    }

    if (serial_handle!=INVALID_HANDLE_VALUE)
        CloseHandle(serial_handle);
    serial_handle = INVALID_HANDLE_VALUE;
}
/* -------------------------------------------------------------------- */
/* --------------------------    disconnect   ------------------------- */
/* -------------------------------------------------------------------- */
void Tserial_event::disconnect(void)
{
    ready = false;
    SetEvent(serial_events[SIG_POWER_DOWN]);

    if (thread_handle!=0)
        WaitForSingleObject(thread_handle, 2000);
    thread_handle = 0;
}
/* -------------------------------------------------------------------- */
/* --------------------------    connect     ------------------------- */
/* -------------------------------------------------------------------- */
/**
     Serial port, file and overlapped structures initialization
*/
int  Tserial_event::connect (char *port_arg, int  rate_arg,  int parity_arg,
                        char ByteSize , bool modem_events)
```

```
{
   int  erreur;
   DCB  dcb;
   int  i;
   COMMTIMEOUTS cto = { 0, 0, 0, 0, 0 };

   /* --------------------------------------------- */
   if (serial_handle!=INVALID_HANDLE_VALUE)
      CloseHandle(serial_handle);
   serial_handle = INVALID_HANDLE_VALUE;

   if (port_arg!=0)
   {
      strncpy(port, port_arg, 10);
      rate        = rate_arg;
      parityMode  = parity_arg;
      check_modem = modem_events;

      erreur      = 0;
      ZeroMemory(&ovReader   ,sizeof(ovReader)   );  // clearing the overlapped
      ZeroMemory(&ovWriter   ,sizeof(ovWriter)   );
      ZeroMemory(&ovWaitEvent,sizeof(ovWaitEvent));
      memset(&dcb,0,sizeof(dcb));

      /* ------------------------------------------------------------------- */
      // set DCB to configure the serial port
      dcb.DCBlength       = sizeof(dcb);

      /* ---------- Serial Port Config ------- */
```

```
dcb.BaudRate       = rate;

switch(parityMode)
{
   case SERIAL_PARITY_NONE:
            dcb.Parity      = NOPARITY;
            dcb.fParity     = 0;
            break;
   case SERIAL_PARITY_EVEN:
            dcb.Parity      = EVENPARITY;
            dcb.fParity     = 1;
            break;
   case SERIAL_PARITY_ODD:
            dcb.Parity      = ODDPARITY;
            dcb.fParity     = 1;
            break;
}


dcb.StopBits       = ONESTOPBIT;
dcb.ByteSize       = (BYTE) ByteSize;

dcb.fOutxCtsFlow    = 0;
dcb.fOutxDsrFlow    = 0;
dcb.fDtrControl     = DTR_CONTROL_DISABLE;
dcb.fDsrSensitivity = 0;
dcb.fRtsControl     = RTS_CONTROL_DISABLE;
dcb.fOutX          = 0;
dcb.fInX           = 0;
```

```
/* ----------------- misc parameters ----- */
dcb.fErrorChar     = 0;
dcb.fBinary        = 1;
dcb.fNull          = 0;
dcb.fAbortOnError  = 0;
dcb.wReserved      = 0;
dcb.XonLim         = 2;
dcb.XoffLim        = 4;
dcb.XonChar        = 0x13;
dcb.XoffChar       = 0x19;
dcb.EvtChar        = 0;

/* ------------------------------------------------------------------ */
serial_handle    = CreateFile(port, GENERIC_READ | GENERIC_WRITE,
          0, NULL, OPEN_EXISTING,FILE_FLAG_OVERLAPPED,NULL);
          // opening serial port

ovReader.hEvent    = serial_events[SIG_READER];
ovWriter.hEvent    = serial_events[SIG_WRITER];
ovWaitEvent.hEvent = serial_events[SIG_MODEM_EVENTS];

if (serial_handle    != INVALID_HANDLE_VALUE)
{
   if (check_modem)
   {
      if(!SetCommMask(serial_handle, EV_RING | EV_RLSD))
         erreur = 1;
   }
```

```
        else
        {
            if(!SetCommMask(serial_handle, 0))
                erreur = 1;
        }


        // set timeouts
        if(!SetCommTimeouts(serial_handle,&cto))
            erreur = 2;

        // set DCB
        if(!SetCommState(serial_handle,&dcb))
            erreur = 4;
    }
    else
        erreur = 8;
}
else
    erreur = 16;


/* --------------------------------------------- */
for (i=0; i<SERIAL_SIGNAL_NBR; i++)
{
    if (serial_events[i]==INVALID_HANDLE_VALUE)
        erreur = 32;
}
```

```
    /* -------------------------------------------- */
    if (erreur!=0)
    {
        CloseHandle(serial_handle);
        serial_handle = INVALID_HANDLE_VALUE;
    }
    else
    {
        // start thread
        thread_handle = (HANDLE) _beginthreadex(NULL,0,
                  (PBEGINTHREADEX_THREADFUNC) Tserial_event_thread_start,
                   this, 0, &threadid);
        /*if (thread_handle==-1)
            thread_handle=0;   */
    }

    /* -------------------------------------------- */
    return(erreur);
}
/* ---------------------------------------------------------------- */
/* ---------------------          setManager     --------------------- */
/* ---------------------------------------------------------------- */
void        Tserial_event::setManager(type_myCallBack manager_arg)
{
        manager = manager_arg;
}
/* ---------------------------------------------------------------- */
/* ---------------------          setRxSize     --------------------- */
/* ---------------------------------------------------------------- */
```

```
void        Tserial_event::setRxSize(int size)
{
       max_rx_size = size;
       if (max_rx_size>SERIAL_MAX_RX)
          max_rx_size = SERIAL_MAX_RX;
}
/* ---------------------------------------------------------------- */
/* --------------------        setManager     -------------------- */
/* ---------------------------------------------------------------- */
char *      Tserial_event::getDataInBuffer(void)
{
   return(rxBuffer);
}
/* ---------------------------------------------------------------- */
/* --------------------        setManager     -------------------- */
/* ---------------------------------------------------------------- */
int      Tserial_event::getDataInSize(void)
{
   return(received_size);
}
/* ---------------------------------------------------------------- */
/* --------------------        setManager     -------------------- */
/* ---------------------------------------------------------------- */
void   Tserial_event::dataHasBeenRead(void)
{
   SetEvent(serial_events[SIG_READ_DONE]);
}
/* ---------------------------------------------------------------- */
/* ---------------------  getNbrOfBytes  ------------------------- */
```

```
/* ------------------------------------------------------------------- */
int Tserial_event::getNbrOfBytes    (void)
{
   struct _COMSTAT status;
   int             n;
   unsigned long   etat;

   n = 0;

   if (serial_handle!=INVALID_HANDLE_VALUE)
   {
      ClearCommError(serial_handle, &etat, &status);
      n = status.cbInQue;
   }
   return(n);
}
/* ------------------------------------------------------------------- */
/* --------------------------    sendData     ------------------------- */
/* ------------------------------------------------------------------- */
void Tserial_event::sendData (char *buffer, int size)
{
   if ((!tx_in_progress) && (size<SERIAL_MAX_TX) && (buffer!=0))
   {
      tx_in_progress = 1;
      memcpy(txBuffer, buffer, size);
      tx_size = size;
      SetEvent(serial_events[SIG_DATA_TO_TX]);
      // indicating data to be sent
   }
```

```
}
/* --------------------------   OnEvent   ------------------------- */
void Tserial_event::OnEvent (unsigned long events)
{
   unsigned long ModemStat;

   GetCommModemStatus(serial_handle, &ModemStat);

   if ((events & EV_RING)!=0)
   {
      if ((ModemStat &  MS_RING_ON)!= 0)
      {
         if (manager!=0)
            manager((uint32) this, SERIAL_RING);
      }
   }

   if ((events & EV_RLSD)!=0)
   {
      if ((ModemStat &  MS_RLSD_ON)!= 0)
      {
         if (manager!=0)
            manager((uint32) this, SERIAL_CD_ON);
      }
      else
      {
         if (manager!=0)
            manager((uint32) this, SERIAL_CD_OFF);
      }
```

```
    }
}
/* ------------------------------------------------------------------ */
/* --------------------------       run      ------------------------ */
/* ------------------------------------------------------------------ */

#define DEBUG_EVENTS
/* */

void Tserial_event::run(void)
{
    bool         done;
    long         status;
    unsigned long read_nbr, result_nbr;
    char         success;

    ready                = true;
    done                 = false;
    tx_in_progress       = 0;
    rx_in_progress       = 0;
    WaitCommEventInProgress = 0;

    if (manager!=0)
        manager((uint32) this, SERIAL_CONNECTED);

    GetLastError();              // just to clear any pending error
    SetEvent(serial_events[SIG_READ_DONE]);
    if (check_modem)
        SetEvent(serial_events[SIG_MODEM_CHECKED]);
```

```
    while(!done)
  {
    /* ------------------------------------------------------------------ */
    /*                                                  */
    /*                                                  */
    /*                                                  */
    /*                  Waiting  for signals                    */
    /*                                                  */
    /*                                                  */
    /*                                                  */
    /* ------------------------------------------------------------------ */
    status = WaitForMultipleObjects(SERIAL_SIGNAL_NBR, serial_events,
                            FALSE, INFINITE);

    // processing answer to filter other failures
    status = status - WAIT_OBJECT_0;
    if ((status<0) || (status>=SERIAL_SIGNAL_NBR))
      done=true;   // error
    else
    {
      /* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*/
      /* ++++++++++++++++++++ EVENT DISPATCHER ++++++++++++++++++
*/
      /* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*/
      switch(status)
      {
```

```
case SIG_POWER_DOWN:
            done = true;
  break;
/* #           RX              # */

case SIG_READ_DONE:
  // previous reading is finished
  // I start a new one here
  if (!rx_in_progress)
  {
    // locking reading
    rx_in_progress = 1;
    // starting a new read
    success = (char) ReadFile(serial_handle,&rxBuffer,
              max_rx_size,&read_nbr,&ovReader);
    if (!success)
    {
      // failure
      if(GetLastError() != ERROR_IO_PENDING )
      {
        // real failure => quiting
        done = true;
        #ifdef DEBUG_EVENTS
        printf("Readfile error (not pending)\n");
        #endif DEBUG_EVENTS
      }
      #ifdef DEBUG_EVENTS
      else
        printf("ReadFile pending\n");
```

```
        #endif DEBUG_EVENTS
      }
      #ifdef DEBUG_EVENTS
      else
      {
        printf("ReadFile immediate success\n");
      }
      #endif
    }
    break;
  /* ############################################################ */
  case SIG_READER:
    // reading the result of the terminated read
    //BOOL GetOverlappedResult(
    //    HANDLE hFile,        // handle of file, pipe, or communications device
    //    LPOVERLAPPED lpOverlapped,    // address of overlapped structure
    //    LPDWORD lpNumberOfBytesTransferred,        // address of actual
bytes count
    //    BOOL bWait          // wait flag
    //   );
    //
    if (GetOverlappedResult(serial_handle, &ovReader,
      &result_nbr, FALSE))
    {
      #ifdef DEBUG_EVENTS
        printf("ReadFile => GetOverlappedResult done\n");
      #endif DEBUG_EVENTS
      // no error => OK
      // Read operation completed successfully
```

```
        ResetEvent(serial_events[SIG_READER]);
        // Write operation completed successfully
        received_size  = result_nbr;
        rx_in_progress = 0; // read has ended
        // if incoming data, I process them
        if ((result_nbr!=0) &&(manager!=0))
            manager((uint32) this, SERIAL_DATA_ARRIVAL);
        // I automatically restart a new read once the
        // previous is completed.
        //SetEvent(serial_events[SIG_READ_DONE]);
        // BUG CORRECTION 02.06.22
    }
    else
    {
        // GetOverlapped didn't succeed !
        // What's the reason ?
        if(GetLastError()!= ERROR_IO_PENDING )
            done = 1;  // failure
    }
    break;
/* #                    TX                         # */
case SIG_DATA_TO_TX:
    success = (char) WriteFile(serial_handle, txBuffer, tx_size,
                    &result_nbr, &ovWriter);
        if (!success)
        {
            // ouups, failure
            if(GetLastError() != ERROR_IO_PENDING )
            {
```

```
                // real failure => quiting
                done = true;
                #ifdef DEBUG_EVENTS
                printf("WriteFile error (not pending)\n");
                #endif DEBUG_EVENTS
            }
            #ifdef DEBUG_EVENTS
            else
                printf("WriteFile pending\n");
            #endif DEBUG_EVENTS
        }
        #ifdef DEBUG_EVENTS
        else
        {
            printf("WriteFile immediate success\n");
        }
        #endif
    break;
/* ############################################################ */
case SIG_WRITER:
    // WriteFile has terminated
    // checking the result of the operation
    if (GetOverlappedResult(serial_handle, &ovWriter,
        &result_nbr, FALSE))
    {
        // Write operation completed successfully
        ResetEvent(serial_events[SIG_WRITER]);
        // further write are now allowed
        tx_in_progress = 0;
```

```
        // telling it to the manager
        if (manager!=0)
            manager((uint32) this, SERIAL_DATA_SENT);
    }
    else
    {
        if(GetLastError() != ERROR_IO_PENDING )
            done = 1;  // failure
    }
    break;
/* #                    MODEM_EVENTS EVENTS                     # */
case SIG_MODEM_CHECKED:
    if ((!WaitCommEventInProgress) && check_modem)
    // if no wait is in progress I start a new one
    {
        WaitCommEventInProgress=1;
        success = (char) WaitCommEvent(serial_handle,&dwCommEvent,
                                &ovWaitEvent);
        // reading one byte only to have immediate answer on each byte
        if (!success)
        {
            // ouups, failure
            if(GetLastError() != ERROR_IO_PENDING )
            {
                // real failure => quiting
                done = true;
                #ifdef DEBUG_EVENTS
                printf("WaitCommEvent error (not pending)\n");
                #endif DEBUG_EVENTS
```

```
            }
            #ifdef DEBUG_EVENTS
            else
                printf("WaitCommEvent pending\n");
            #endif DEBUG_EVENTS
        }
        #ifdef DEBUG_EVENTS
        else
        {
            printf("WaitCommEvent immediate success\n");
        }
        #endif
    }
    break;
/* ############################################################ */
case SIG_MODEM_EVENTS:
    // reading the result of the terminated wait
    if (GetOverlappedResult(serial_handle, &ovWaitEvent,
        &result_nbr, FALSE))
    {
        // Wait operation completed successfully
        ResetEvent(serial_events[SIG_MODEM_EVENTS]);
        WaitCommEventInProgress = 0;
        // if incoming data, I process them
        OnEvent(dwCommEvent);
        // automatically starting a new check
        SetEvent(serial_events[SIG_MODEM_CHECKED]);
    }
    else
```

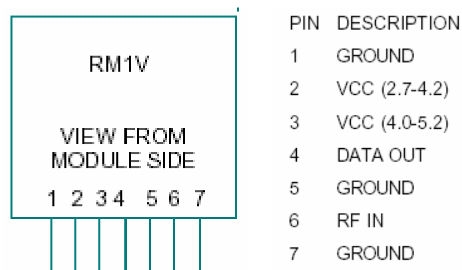
```
                {
                    // GetOverlapped didn't succeed !
                    // What's the reason ?
                    if(GetLastError() != ERROR_IO_PENDING )
                        done = 1;  // failure
                }
                break;
                    }
        }
    };

    // --------------------- Disconnecting ----------------
    ready = false;
    if (serial_handle!=INVALID_HANDLE_VALUE)
        CloseHandle(serial_handle);
    serial_handle = INVALID_HANDLE_VALUE;

    if (manager!=0)
        manager((uint32) this, SERIAL_DISCONNECTED);
}
/* ------------------------------------------------------------------ */
```

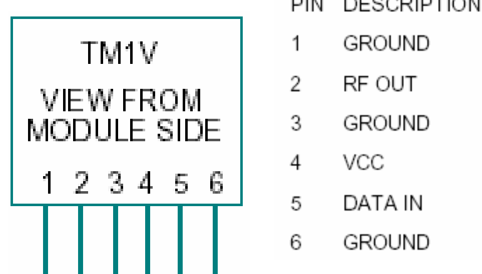
APPENDIX V

A. Specifications of RM1V Receiver



Pin descriptions:		
1	Ground	Connect to ground plane
2	Vcc (2.7-4.2 vdc)	Use this pin for supply voltages of less than 4.2 vdc
3	Vcc (4.0-5.2 vdc)	Use this pin for supply voltages of 4.0 vdc or greater
4	Data out	Recovered data, voltage during high bit = Vcc
5	Ground	Connect to ground plane
6	RF in	Receiver antenna input. Capacitively isolated from circuit.
7	Ground	Connect to ground plane

B. Specifications of TM1V Transmitter



Pin descriptions:		
1	Ground	Connect to ground plane
2	RF out	Connect to 50 ohm antenna
3	Ground	Connect to ground plane
4	Vcc	Positive supply 2.7 to 5.2 vdc (<= 20mv pp noise)
5	Data	Serial data input pin CMOS and TTL compatible
6	Ground	Connect to ground plane

VITA

Sriram T. Vengalathur received his B.E. degree in mechanical engineering from B.M.S.C.E., Bangalore in 1999. He joined the master’s program at Texas A&M University in August 2000 and graduated in August 2003. He can be contacted through the Department of Mechanical Engineering, Texas A&M University, College Station, TX-77843-3123.