THE MANAGEMENT AND IMPLEMENTATION OF A DATA TERMINAL PROJECT ON A GLOBAL SYSTEM FOR MOBILE COMMUNICATIONS

Tjaart van der Walt, B. Ing.

Thesis submitted in the Department of Electrical and Electronic Engineering of the Potchefstroomse Universiteit vir Christelike Hoër Onderwys in partial fulfilment of the requirements for the degree Magister in Electronic Engineering.

Study leader : Prof. C.P. Bodenstein

POTCHEFSTROOM
1999

DIE BESTUUR EN IMPLEMENTERING VAN 'n DATA TERMINAAL OP 'n GLOBALE STELSEL VIR MOBIELE KOMMUNIKASIE

Tjaart van der Walt, B. Ing.

Verhandeling voorgelê vir gedeeltelike nakoming van die vereistes van die graad Magister in Elektroniese Ingenieurswese in the Departement Elektriese en Elektroniese Ingenieurswese aan die Potchefstroomse Universiteit vir Christelike Hoër Onderwys.

Studieleier : Prof. C.H. Bodenstein

POTCHEFSTROOM
1999

Preface

This project was a personal challenge to myself because there were no other products to measure or compare it with. I was also given *carte blanche* by my employer (Wasp International) to design the data terminal in any way that I saw fit, for which I am thankful.

I would also like to thank my supervisor, Prof. C.P. Bodenstein for his much appreciated advice and guidance throughout this project.

I would mostly like to thank my wife Julie, for the tons of patience and love, and helping me to 'bite the bullet'. I am not sure if I always deserved it!

Summary

With the demand for more control over remote systems and access to remote data increasing, there existed the need for a more efficient and low cost means of transmitting this data. Traditionally, small telemetry data packets have been transmitted using unreliable direct radio frequency links, or expensive copper connections or satellite communications. With the international infrastructure of the Global System for Mobile communication (GSM) networks and it's ever-improving coverage, GSM has become a viable solution as the data channel for data telemetry. With a data terminal, remote monitoring of processes in a factory plant with real-time access to the parameters will be possible. If the per-unit cost is low enough, there would be a use for it in pre-paid energy metering, or even in the control of peak energy demand.

The goal of this project was twofold : firstly to apply a structured project management approach to the design cycle of a data terminal, and secondly to design such a terminal in the shortest possible time. It was anticipated that the use of a structured management process would reduce the development time.

The data terminal was not designed for a specific customer, but was to be as versatile and flexible as possible. It is understood that only a percentage of all telemetry applications will be able to use such a device, and the trade-off was to balance the cost of the unit (and functionality) with too much complexity.

The project was completed in a relatively short time, and the terminal found to be extremely flexible. The unit cost was kept low as compared to other systems. In fact, the complete unit will cost the end user less than just the communication devices in the trunk radio and satellite systems it can replace.

The short message service (SMS) as a communication channel has proven itself to be more reliable and of lower cost than other competing technologies. The ongoing expansion and improvement of the GSM networks by their operators, also ensure a stable platform for future use of the data terminal.

Opsomming

Met die verhoogde behoefte aan afstandbeheer oor nie ter plaatse stelsels en die toegang tot sulke data, het daar 'n behoefte ontstaan vir 'n meer effektiewe en goedkoper metode om die data te vervoer. Tot nou was klein telemetrie datapakkies vervoer deur middel van onbetroubare radiofrekwensiekoppelings, of deur duurder koperkoppelings of satellietkommunikasie. Met die internasionale infrastruktuur van die Global System for Mobile communications (GSM) netwerke en die voortdurenede verbetering van die diensarea, het GSM 'n goeie kandidaat geword vir die datakanaal vir telemetriestelsels. Met so 'n dataterminaal kan nie ter plaatse monitering van prosesse in fabrieke met intydse toegang tot die parameters gedoen word. Indien die eenheidkoste laag genoeg gehou kan word, sou dit ook in vooraf-betaal energiemeting of vir beheer oor piek energiegebruik gebruik kon word.

Die doel van die projek was tweevoudig : eerstens om 'n gestruktureerde projekbestuur benadering te gebruik in die ontwikkelingsiklus en tweedens om die dataterminaal in die kortste tyd moontlik te ontwerp. Daar is voorsien dat die gebruik van gestruktureerde projekbestuur die ontwikkelingstyd sou verkort.

Die dataterminaal was nie vir 'n spesifieke kliënt ontwikkel nie, maar moes so bruikbaar en buigsaam moontlik wees. Daar is aanvaar dat slegs 'n persentasie van moontlike toepassings van die terminaal gebruik sou kon maak en die koste (en funksionaliteit) van die eenheid moes opgeweeg word teen onnodige kompleksiteit.

Die projek was binne 'n baie kort tydperk afgehandel en daar is gevind dat die terminaal hoogs buigsaam is vir verskillende toepassings. Die eenheidkoste was laag gehou indien vergelyk word met ander stelsels. Die koste vir 'n volledige data teminaal is selfs laer as slegs die kommunikasietoerusting van die radio- en satellietstelsels wat dit kan vervang.

Die kort boodskap stelsel (SMS) as 'n kommunikasie kanaal is bewys om meer betroubaar en goedkoper te wees as ander kompeterende stelsels. Die voortdurende verbetering en uitbreiding van die GSM netwerkoperateurs verseker 'n stabiele basis vir die toekomstige gebruik van die dataterminaal.

Table of Contents

CHAPTER 1

1. THE NEED FOR A DATA TERMINAL 1

1.1 MOBILE ASSET MANAGEMENT 1
1.2 SECURITY MONITORING AND ARMED RESPONSE 2
1.3 ENERGY/WATER METERING AND CONTROL 2
1.4 INDUSTRIAL PROCESS AND EXCEPTION MONITORING AND CONTROL 3

2. COMMUNICATIONS CHANNELS 4

2.1 RADIO TRUNKING 4
2.2 SATELLITE COMMUNICATION 5
Low Earth Orbit 5
Medium Earth Orbit 5
Geostationary Earth Orbit 5
2.3 GLOBAL SYSTEM FOR MOBILE COMMUNICATIONS (GSM) 6

3. PRODUCT SPECIFICATIONS 7

4. THE DEVELOPMENT AND PRODUCTION PHILOSOPHY 7

5. TESTING AND VALIDATION 7

5.1 TERMINATOR 7
5.2 PASSWORD 7
5.3 MESSAGE TYPE 8
5.4 SEQUENCE NUMBER 8

6. THE DEVELOPMENT CYCLE 8

CHAPTER 2

7. NEW PRODUCT REQUEST 14

7.1 PRODUCT DESCRIPTION 14
Description 14
Functionality 14
Interface 14
Power Supply 14
7.2 PRODUCT TARGET MARKET 14
Product Positioning 14
Budget 15
7.3 MILESTONE 1 EVALUATION 15

8. PRODUCT DEFINITION 15

8.1 COMPONENT SELECTION 15
Communication Medium 15
Microcontroller 15
Power Supply 15
Input Protection 16
Real Time Clock 16
Memory 16
Revised Interface Specifications 16
8.2 MESSAGE STRUCTURES 18
Communication Medium 18
COMMUNICATION PROTOCOL 18
FIRMWARE STRUCTURE 19

ENCLOSURE 19
THIRD PARTY DEVELOPMENTS 19

9. MILESTONE 2 EVALUATION 19

CHAPTER 3

10. PROTOTYPE DEVELOPMENT 20

10.1 DEVELOPMENT JIGS 20
Programming Jig 20
Downloading Jig 20
Breakout Monitor Jig 21
10.2 HARDWARE 21
Schematic Design 21
PCB Design 21
Prototype PCB 21
10.3 DEVELOPMENT SOFTWARE 21
Prototype Set-up Software 21
10.4 FIRMWARE & CUSTOMER SOFTWARE 22
Detail Protocol Design - Message Path 22
Detail Protocol Design - Mobile Originating (MO) 22
Detail Protocol Design - Mobile Terminating (MT) 27
10.5 FIRMWARE MODULE DESIGN 34
UART Character Received (ISR.1) 36
UART Character Transmitted (ISR.2) 38
Time Base : 20ms (ISR.3) 38
Time Base : 100ms (ISR.4) 40
Time Base : 1s (ISR.5) 41
Set up Micro (SC.1) 42
Download set-up values (SC.2) 42
Load String (OPM.1) 43
Save String (OPM.2) 43
Load IO Parameters (OPM.3) 44
Save IO Parameters (OPM.4) 45
Set-up RTC (OPM.5) 46
Get Time (OPM.6) 46
Issue Command (OPM.7) 48
Resolve Message Header (OPM.8) 51
Get Next Command (OPM.9) 52
Send the Log (OPM.10) 54
ASCII to HEX (OPM.11) 56
Binary to BCD (OPM.12) 56
BCD to Binary (OPM.13) 56
10.6 CUSTOMER SOFTWARE CONCEPT DESIGN 57
10.7 IN-CIRCUIT EMULATOR SET-UP 58
10.8 FIRMWARE MODULE INTEGRATION 58
State Based Operations 59
Non - State Based Operations 60

11. PRODUCT DEVELOPMENT 71

11.1 PRODUCTION HARDWARE 71
Bed-of-Nails 71
Soak Test 71
Repair/Rework 72
11.2 PRODUCTION SOFTWARE 73
Bed-of-Nails 73
Soak Test 75
Repair/Rework 75

12. MILESTONE 3 EVALUATION 76

CHAPTER 4

13. SAMPLE RUN 77

13.1 TESTING FOR MANUFACTURABILITY OR ASSEMBLY PROBLEMS 77
13.2 REVISION 77

14. PRODUCTION QUALITY CONTROL (QC) PLAN 77

14.1 PROCEDURES FOR QUALITY CONTROL 77
Batch Procedure Development 77
Component Manufacturer Specification 77

15. PRODUCTION TOOLS 77

15.1 FAILURE MODE EFFECT AND ANALYSIS (FMEA) 77
Hardware 77
Software 77
15.2 REVISION 77
15.3 CORRECTIVE MEASURES 77

16. PRODUCTION MANUAL 78

16.1 FINAL COPY 78

17. REPAIR MANUAL 78

17.1 FINAL COPY 78

18. PRODUCT FAILURE MODE EFFECT AND ANALYSIS (FMEA) 79

18.1 HARDWARE 79
Hardware FMEA 79
Hardware Corrective Action 80
18.2 FIRMWARE 81
Firmware FMEA 81
Firmware Corrective Action 83

19. PACKAGING 85

19.1 DESIGN 85
19.2 DIE-CUTTING TOOLS 85
19.3 CORRECTIVE MEASURES 85

20. ENCLOSURE 85

20.1 TOOLING 85
20.2 EFFECT ON ASSEMBLY 85
20.3 OFF-TOOL SAMPLES 86
20.4 CORRECTIVE MEASURES 86

21. CUSTOMER SOFTWARE 86

21.1 FAILURE MODE EFFECT AND ANALYSIS (FMEA) 86

22. CUSTOMER MANUAL 86

22.1 INFORMATION PACKAGE 86
User Manual 86
Pocket Guide 86

23. INSTALLATION MANUAL 86

23.1 INFORMATION PACKAGE 86
Installation Manual 86
Trouble Shooting Guide 86
23.2 CORRECTIVE MEASURES 86

24. BAR CODED LABELS 87

24.1 DESIGN 87

Serial Number ... 87
24.2 CHECK FOR DURABILITY ... 87
Glue/Temperature/Colour fading/solvents/water ... 87

25. FACE PLATES ... 87

25.1 DESIGN ... 87
Legend ... 87
Tooling ... 87
25.2 CHECK FOR DURABILITY ... 88
Temperature ... 88
Solvents ... 88
Water ... 88
25.3 CORRECTIVE MEASURES ... 88

26. HELP DESK ... 88

26.1 PRODUCT FAMILIARISING ... 88
Documentation ... 88
Information Sessions ... 88
Hands-on Training ... 88
Frequently Asked Questions (FAQ) ... 88
26.2 WEB SITE DOWNLOADS ... 88
Public Information ... 88
Customer Information ... 88

27. MARKETING ... 88

27.1 STRATEGY ... 89
Markets ... 89
27.2 DESIGN ... 89
Photography ... 89
Web site ... 89

28. MILESTONE 4 EVALUATION ... 89

CHAPTER 5

29. SAMPLE RUN ... 90

29.1 TEST FOR MANUFACTURABILITY OR ASSEMBLY PROBLEMS ... 90
Final Testing ... 90

30. PRODUCTION QUALITY CONTROL (QC) PLAN ... 90

30.1 PROCEDURES FOR QUALITY CONTROL IMPLEMENTED ... 90

31. PRODUCTION MANUAL ... 90

31.1 FINAL COPY ... 90

32. REPAIR MANUAL ... 90

32.1 FINAL COPY ... 90

33. INSTALLER TRAINING ... 90

33.1 TRAINING SESSIONS ... 90
33.2 TRAIN BY EXAMPLE ... 90
33.3 Q&A SESSION ... 90

34. BUREAU OPERATOR TRAINING ... 91

34.1 TRAINING SESSIONS ... 91
34.2 Q&A SESSION ... 91

35. COSMETIC CHECKS ... 91

35.1 PRODUCTION MARKS ... 91

35.2 LABELS AND LEGEND 91
35.3 PACKAGING 91

36. MILESTONE 5 EVALUATION 91

CHAPTER 6

37. PROTOCOL ROBUSTNESS AND SYSTEM PERFORMANCE 92

37.1 TESTING METHODOLOGY 92
37.2 RESULTS 93

38. INTERFACE ACCURACY 94

38.1 TESTING METHODOLOGY 94
38.2 RESULTS 94
Digital Inputs 94
Power Sense 94
Analogue Inputs 94
Frequency Counter Input 94
Outputs 94

39. FUNCTIONALITY 94

39.1 TESTING METHODOLOGY AND RESULTS 94

CHAPTER 7

40. THE MANAGEMENT AND IMPLEMENTATION OF A GLOBAL SYSTEM FOR MOBILE COMMUNICATIONS (GSM) BASED DATA TERMINAL 95

40.1 COMPETING IN INFORMATION TECHNOLOGY PRODUCT MARKETS 95
40.2 MANAGEMENT STRATEGY 95
40.3 THE DATA TERMINAL 95
The GSM engine may be replaced or augmented by a satellite communicator 95
The implementation of MODbus 96
The implementation of a proprietary protocol 96
40.4 CONCLUSION 96

BIBLIOGRAPHY

APPENDIX A

ELECTRICAL SPECIFICATIONS ..... ATTACHED

APPENDIX B

TEST RESULTS - LOG FILE ..... ATTACHED

APPENDIX C

PHOTOGRAPH OF FINISHED PRODUCT ..... ATTACHED

APPENDIX D

CIRCUIT DESIGN - SCHEMATIC ..... ATTACHED

APPENDIX E

PRINTED CIRCUIT BOARD DESIGN ..... ATTACHED

APPENDIX F

CDROM ..... ATTACHED

INDEX OF FIGURES

1. CHAPTER 1 1

FIGURE 1. THE USE OF A GSM TERMINAL IN MOBILE ASSET MANAGEMENT. 1
FIGURE 2. THE USE OF A DATA TERMINAL AS A SECURITY DEVICE.. 2
FIGURE 3. THE USE OF A DATA TERMINAL IN ENERGY AND WATER CONTROL. 3
FIGURE 4. THE USE OF A DATA TERMINAL IN PLANT MANAGEMENT 4
FIGURE 5. THE OUTING OF SMS MESSAGES.. 7

2. CHAPTER 2 14

FIGURE 6. THE RELATIVE COSTS OF DIFFERENT CHANNELS OF COMMUNICATIONS OVER THE GSM 18

3. CHAPTER 3 20

FIGURE 7. THE PROGRAMMING JIG. 20
FIGURE 8. THE JIG USED TO DOWNLOAD THE SETUP DATA INTO THE NON-VOLATILE MEMORY 20
FIGURE 9. THE BREAKOUT MONITOR JIG USED TO MONITOR THE COMMUNICATIONS 21
FIGURE 10. THE HOST COMPUTER DOWNLOADING APPLICATION 22
FIGURE 11. THE CUSTOMER SOFTWARE CONCEPT 57
FIGURE 12. THE EMULATOR DER DELOPMENT ENVIRONMENT 58
FIGURE 13. THE DESIGN STRATEGY OF THE FIRMWARE. 58
FIGURE 14. THE STATE BASED TASKS IN THE MAIN LOOP OF THE FIRMWARE 60
FIGURE 15. THE NON - STATE BASED (LOW PRIORITY) TASKS IN THE MAIN LOOP OF THE FIRMWARE 61
FIGURE 16. THE EVENT FLAGS ARE CHECKED FOR TRIGGERS 62
FIGURE 17. READING THE MESSAGES ON THE SIM CARD 64
FIGURE 18. THE MANUFACTURING PROCESS 71
FIGURE 19. THE SOAK TEST JIG 72
FIGURE 20. THE REWORK PROCESS 72
FIGURE 21. THE COMPONENT POPULATION PROCESS 73

FIGURE 22. THE ASSENBLY PROCESS. 74
FIGURE 23. THE SOAK TEST PROCESS 75
FIGURE 24. THE REWORK/REPAIR SOFTWARE 75

4. CHAPTER 4 77

FIGURE 25. THE SNAP-ON CLAM-TYPE ENCLOSURE 85
FIGURE 26. THE BAR-CODE LABEL 87
FIGURE 27. THE LEGEND TO BE PAD-PRINTED ON THE ENCLOSURE. 87

5. CHAPTER 5 90

6. CHAPTER 6 92

FIGURE 28. THE TEST CONFIGUTATION 92
FIGURE 29. THE DELIVERY TIMES OF SMS MESSAGES OVER A 100 MINUTE PERIOD 93
FIGURE 30. THE DISTRIBUTION OF DELIVERY TIMES 93

7. CHAPTER 7 96

INDEX OF TABLES

1. CHAPTER 1 1

TABLE 1. STAGE 1 OF THE DEVELOPMENT CYCLE 9
TABLE 2. STAGE 2 OF THE DEVELOPMENT CYCLE 10
TABLE 3. STAGE 3 OF THE DEVELOPMENT CYCLE 12
TABLE 4. STAGE 4 OF THE DEVELOPMENT CYCLE 13

Chapter 1 Introduction

1. The Need For a Data Terminal

The need was identified for a product that would enable one to interact with a remote site. This interaction includes the measurement and limited processing of electrical parameters and the switching of outputs.

The interaction must be controlled from a remote computer. The communication channel must be wireless, thus enabling the user to exercise control from virtually anywhere. The GSM was chosen as communications medium, as it is a proven medium with operational installations world-wide. It is also very cost-effective when compared to other wireless means of communication, like satellite and radio trunking.

1.1 Mobile Asset Management

Some companies specialise in renting out equipment that is uneconomical for the user to purchase. This implies that these assets are usually very expensive and may also require regular or scheduled maintenance. Other companies may want to keep track of the status of their mobile assets, like whether a vending machine needs to be re-filled with a particular product. Examples of such assets, are :

- Photocopier machines
- Vending machines
- Rented computers, fax machines and peripheral equipment
- Construction and building equipment (air compressors, generators, hammers etc)
- Television and video recorder equipment
- Electronic test and measurement equipment

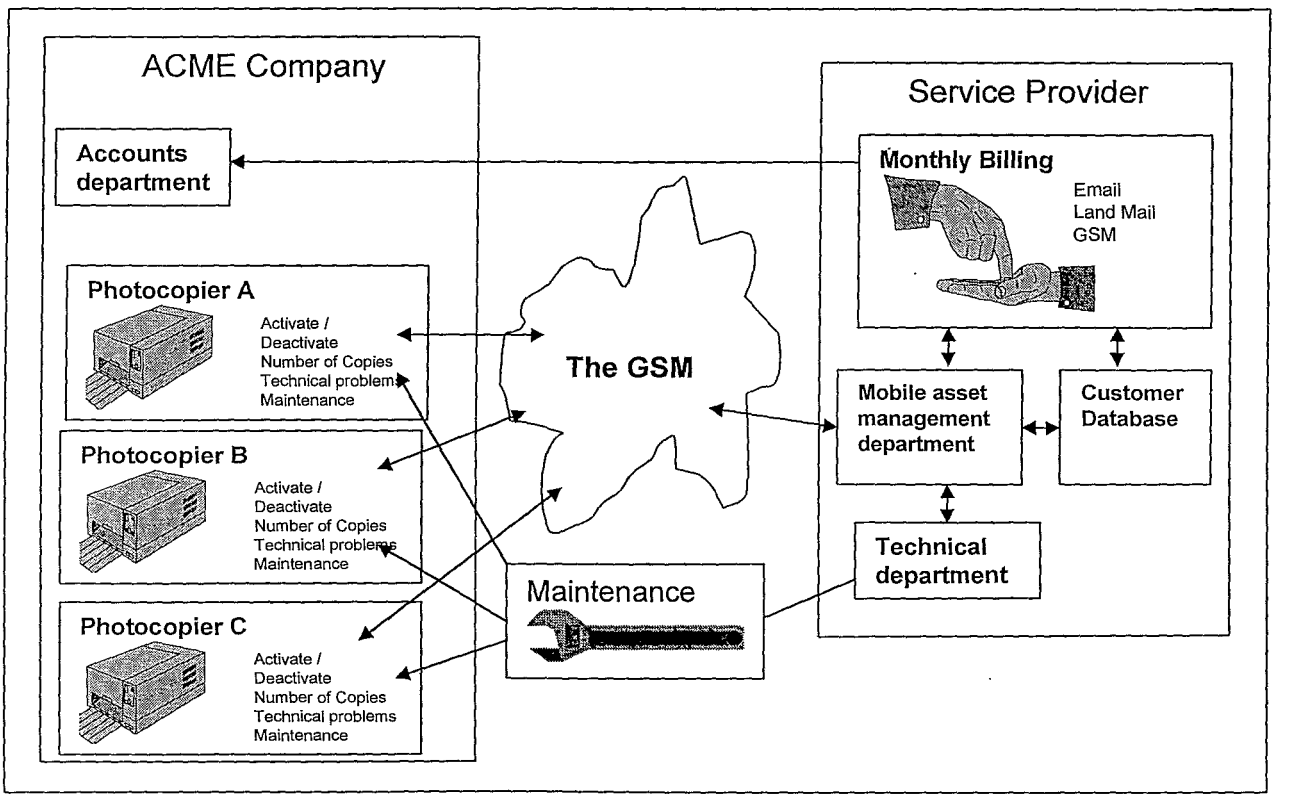


Figure 1. The use of a GSM terminal in mobile asset management.

1.2 Security Monitoring and Armed Response

Financial institutions can monitor high-security areas and distress calls made by way of panic buttons. Redundant or second-level control can be added where intervention from a control room operator is needed to gain access to high-security areas. In the illustrated example, more than one company may use a shared security monitoring and response firm to save on costs.

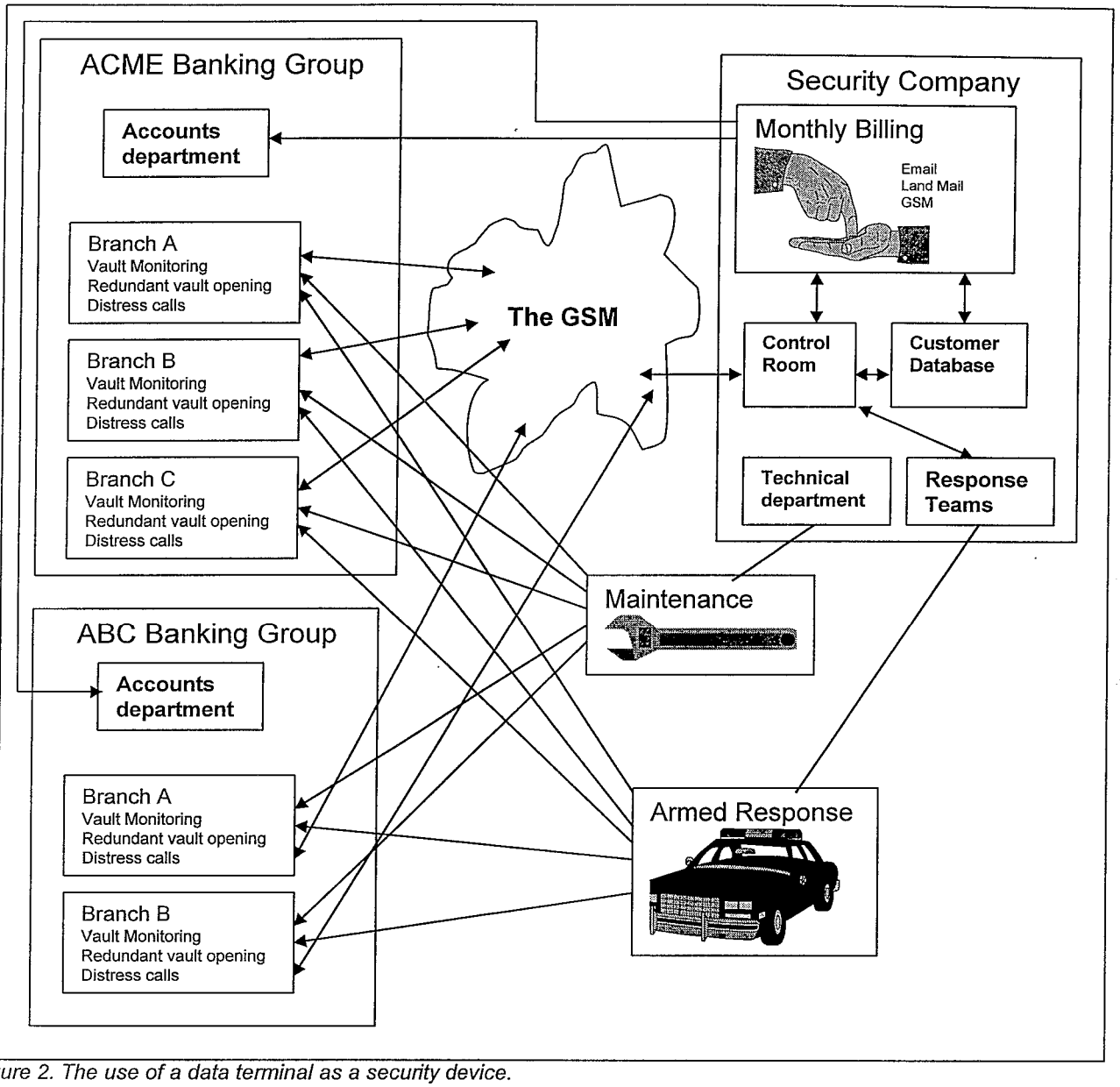


Figure 2. The use of a data terminal as a security device.

1.3 Energy/Water Metering and Control

Most large municipalities use peak energy control to reduce the peak-rate charges from the electricity supplier. This control is in the form of an electric current interruption of high-current domestic devices like water heating systems. The communication is effected through the energy distribution network, and limited to the switching of these devices.

A system is proposed that will offer full control over the user's energy and water utilistation. Not only must the municipality be able to interrupt the high-current devices during peak times, but also be able to know how much water and electricity was used for a month period, and be able to disconnect it in cases of non-payment. Pre-paid accounts will also be able to use such a system.

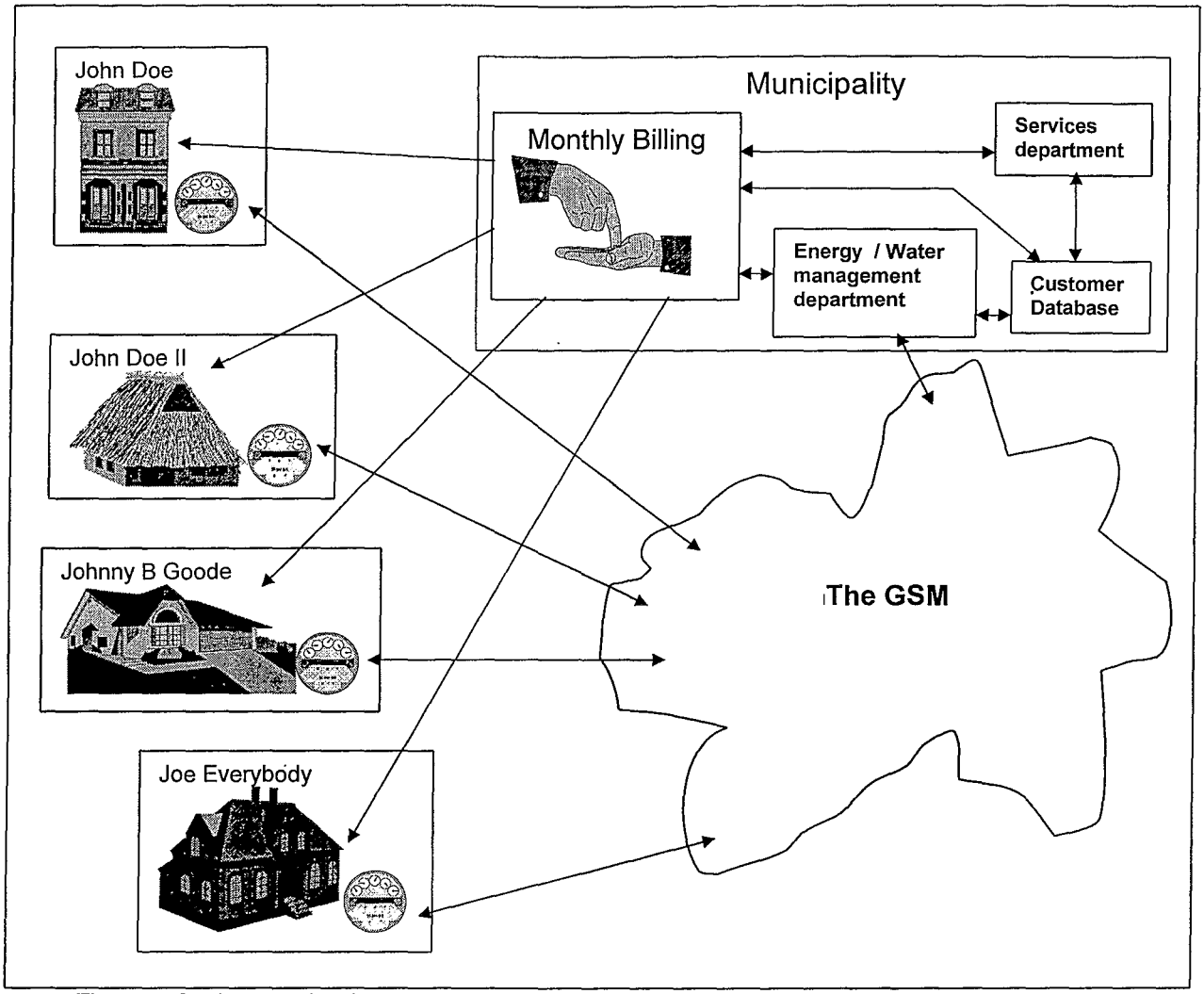


Figure 3. The use of a data terminal in energy and water control.

1.4 Industrial Process and Exception Monitoring and Control

Most factories or industrial processing plants will have some or other process or status that is critical in the successful operation. This process or status may even pose a threat to the workers, close-by communities, or the environment. To monitor these processes, the GSM data terminal may be connected to measuring devices to enable the engineer charged with the responsibility to monitor one or more processes or states in the plant. The engineer will also be able to intervene in the case of a runaway process by disabling some of its subsystems, or even granting access to certain areas to on-site personnel for emergency maintenance.

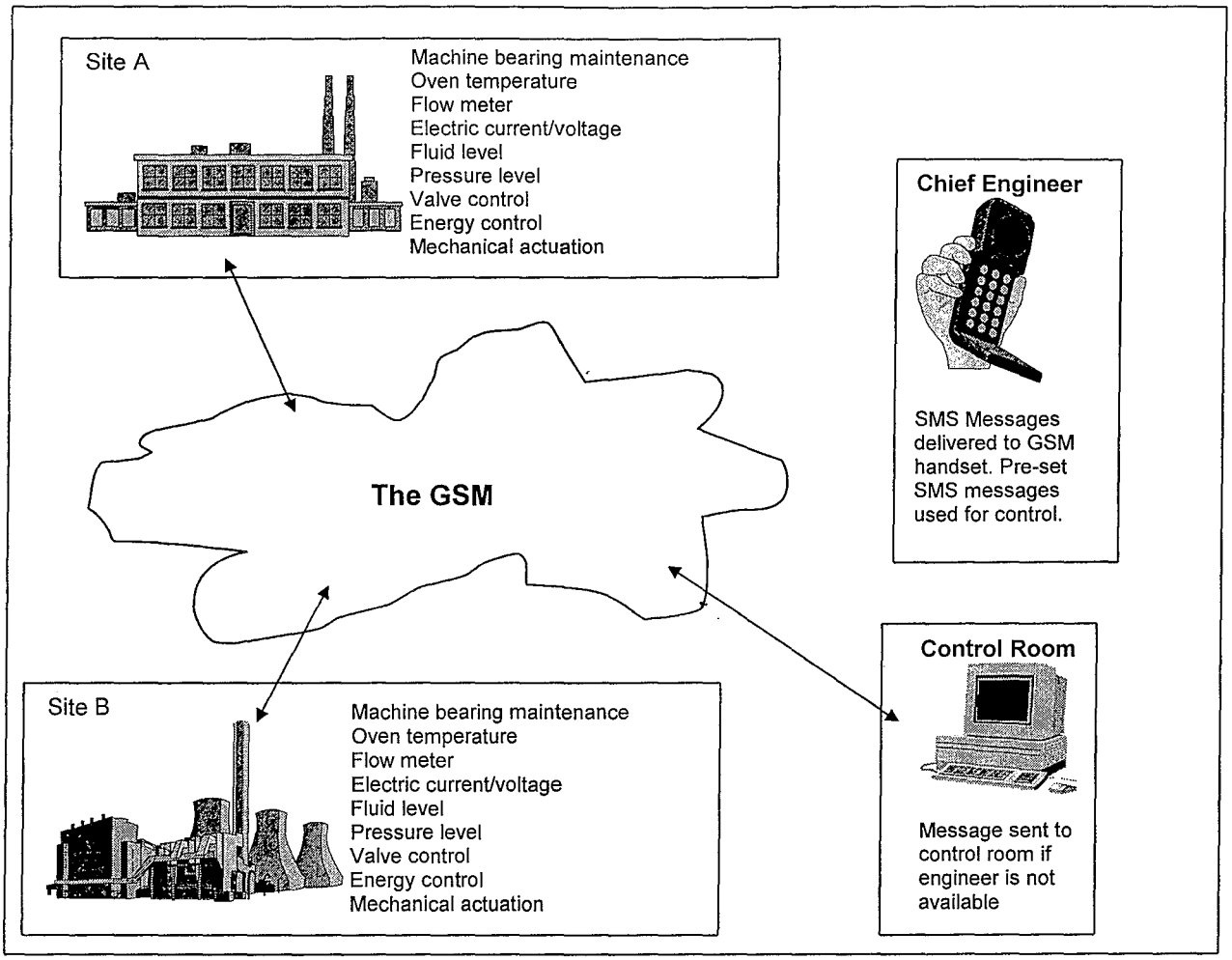


Figure 4. The use of a data terminal in plant management.

2. Communications Channels

The data from such a terminal may be transferred by a number of radio frequency technologies.

2.1 Radio Trunking

Radio trunking is a terrestrial cellular system by which data from a user may be transmitted to another user in the same cell, or to another cell in the network. The communication usually takes a digital form and may carry voice, data, or short messages (27bytes) depending on the network and system. Some trunking networks also connect to PABX's for integration into the telephone network.

There are many different standards and protocols from different vendors and different frequencies used in different countries. This makes radio trunking a very closed system with regards to roaming between networks or countries. ETSI (European Telecommunications Standards Institute) standards are defined for some of the systems like TETRA. The predominant systems are :

- MPT1327
- EDACS
- TETRA
- TETRAPOL
- LTR
- EASYCOM
- NEXTEL
- MULTINET-II
- Motorola SmartZone

There are currently four large trunking networks in South Africa :

- Fleetcall operate a MPT1327 trunked network using the frequencies 254MHz-263MHz. The network covers Cape Town and the Eastern Cape.
- One-2-One also operate a MPT1327 trunked network using the frequencies 254MHz-263MHz. The network covers the Western Cape and Bloemfontein.
- Q-Trunk operate a national MPT1327network, also using the frequencies 254MHz-263MHz.
- The Police and Security Forces in the Western Cape use a Motorola SmartZone system in the UHF band.

2.2 Satellite Communication

Low Earth Orbit

These low-earth orbit satellites are usually better suited for messaging, because the fly-by times are short due to the low orbit. It they were to carry voice or data communication, complex handover schemes would have to be used. A typical fly-by time could be in the order of 11 minutes.

The point of these satellites is that they are relatively cheap to manufacture and to launch. Instead of an expensive rocket launch, these satellites are launched in groups from a small airplane-based rocket. An operator would typically launch a complete orbital plane of satellites at once. Small rockets on the satellites spread them across their orbital plane over a period of a few months.

Because these satellites are so close to earth (500km – 2000km), transmission power levels to and from these space vehicles are dramatically lower than with the geostationary satellites. The LEO systems also have to rely on much more extensive constellations for proper coverage. A typical constellation will have more than thirty space vehicles, with other constellations up to 48 (Leo One). Most systems use a store-and-forward message system.

The major players are :
Orbcomm
The complete constellation will have 36 satellites in orbit. The uplink is maximum 2400bps (FSK) at 148.0MHz - 149.9MHz, and the downlink is 4800bps (FSK) at 137.0MHz – 138.0MHz. The typical cost of communication, is US$0.01 per character with monthly charges of around US$30.00.

Leo One
The constellation will consist of 48 satellites in eight planes 950km above the earth

KITComm
This is a very low-cost one-way message system that covers Australia. It was designed to track cows, trucks, power lines, tanks etc.

E-Sat
This constellation will have only six space vehicles in orbit at 1260km above earth. The coverage will focus mainly on North America.

Medium Earth Orbit

These satellite communication systems fall outside of the scope of this paper because of its higher communications and equipment cost, and power requirements. The major players are :

- Teledesic
- SkyBridge
- Orblink
- Pentriad
- M-Star

Geostationary Earth Orbit

These space vehicles occupy that orbit which Arthur C Clarke described decades ago where the orbital speed of the satellite, and the rotational speed of the earth would be identical. This means the satellite won't move in relation to the earth. The costs associated with communication over

these networks make them totally unacceptable for low-cost telemetry. For completeness' sake, the major players in this area are mentioned :

- Inmarsat
- ACeS
- APMT
- AMSC
- Iridium
- Globalstar
- AstroLink
- Cyberstar
- Euroskyway
- GESN
- KaStar
- Aster
- V-Stream
- Worldspace
- WEST

2.3 Global System for Mobile Communications (GSM)

The GSM (Global System for Mobile communications) is a standard for wireless communication in a cellular radio environment. The mobile terminal does not have to transmit over long distances, resulting in the unit being much smaller in size and using less power.

The specification responsibility was given to ETSI (European Telecommunications Standards Institute) in 1989, and the Phase I specification was released in1990. By 1993, there were 36 GSM networks in 22 countries. Currently, there are more than 200 GSM networks operational in more than 110 countries.

There are three variants of GSM, namely the standard GSM900, DCS1800, and PCS1900. The dominant GSM900 uses a combination of Time Division Multiple Access (TDMA) and Frequency Division Multiple Access (FDMA) at an uplink frequency of 890-915MHz, and a downlink frequency of 935MHz – 960MHz.

There are a number of communication services available on the GSM, with the voice service being the most well known and widely used. There is also a data service for data calls with rates up to 9600bps to users on POTS (Plain Old Telephone System), ISDN and other systems. No modem is necessary between the user and the network, although an analogue modem is used in the GSM network to connect to POTS users. Group 3 facsimile services are also supported.

The service we will use, is the Small Message Service (SMS). SMS is a service designed for small data packets that are transmitted over the control channel of the communication layer. This means that SMS messages are not affected as much by network congestion due to large volumes of voice calls.

The data packet length of an SMS message depends on the encoding scheme used. In 8 bit mode (also called PDU mode for Packet Data Unit), the maximum packet length is 140 bytes. In 7 bit mode (also called Text mode), the maximum packet length is 160 characters (7 bits each).

The routing of an SMS message does not follow the data path of the other traffic, but is rather routed by a service centre. This service centre handles the delivery scheduling and retrying, as well as the network routing for a message.

The Service Centre is a computer (or network of computers) that routes the SMS message to its destination. The cloud represents the GSM network and the BTS towers the base transceiver stations that we see all over the country. A message can, but does not have to return data via the same service centre (SMSC) that was used to deliver the message. An SMS message can be sent to any GSM device on any GSM network in the world, provided they have a subscriber roaming agreement with the network that is hosting the originating device.

The path usually travelled by an SMS message, is as follows :

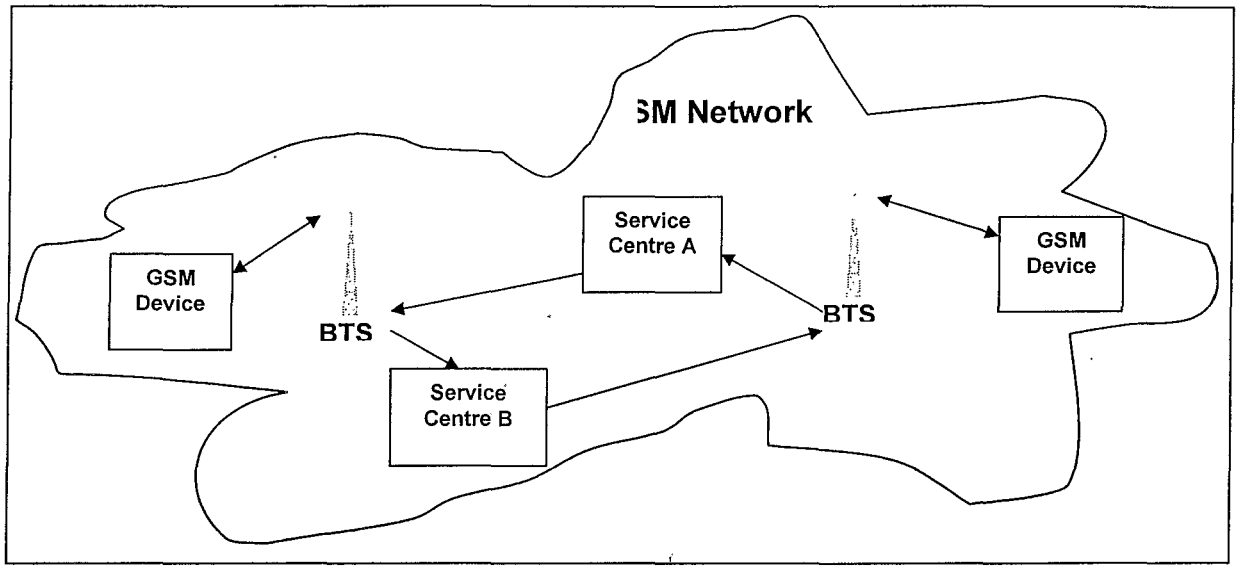


Figure 5. The routing of SMS messages.

3. Product Specifications

Because the data terminal will be designed for general market applications, and not for a specific customer, the specifications are not as rigid as would be the case if it were designed to meet a customer's requirement. Some minimum specifications would be defined, after which any additional functionality can be investigated.

4. The Development and Production Philosophy

The design will take into account not only the actual use of the product, but also the manufacturability of the units. Failures must be prevented rather than minimised. By designing the product with ease of manufacturing and testing in mind, the general reliability increases because of the decrease in manufacturing defects.

5. Testing and Validation

After development the data terminal will be tested for communication throughput and interface accuracy. Its communications protocols will be tested for robustness and speed of communication, whilst the interface will be tested for accuracy. It is expected that the data terminal will satisfy the need for communication speed and low error rate. The protocol will be designed for high integrity. This may be achieved through :

5.1 Terminator

A known character is inserted after the data string. There will be a length byte to specify the location of this character. If this character is not present, the message must be ignored.

5.2 Password

A 16 bit password is used to gain access to the terminal. If the password is incorrect, the terminal must ignore the message. Corrupt messages will almost certainly not have the correct password.

5.3 Message Type

A specific message type number is inserted in a specific position. This number can only take one of a few values. Invalid message types are also ignored.

5.4 Sequence Number

To allow the base station to keep track of all the outgoing messages, it must be able to attach a numeric value to each outgoing message. This number is contained in the return message.

6. The Development Cycle

The development will follow an adaptation of the industry standard development flow. The development flow is governed by a series of milestone meetings. At these meetings the work preceding the meeting is discussed and the decision to proceed to the next stage or not, is taken. Because this project was undertaken and managed by one person, the meetings will be replaced by reflections on the completed tasks.

In the development flow diagram, there are thick vertical bars. These bars represent a completion of all the tasks marked by it. This usually precedes a milestone meeting, or the transition to the next logical step. The flow of time is from left to right. The flow diagrams used are presented at the end of this chapter, and the rest of the structure of this document will follow the contents of the diagrams as closely as possible.

Stage 1 - Product Planning

Time →

New Product Request

Product Description	**Functionality** Minimum functionality required to meet market expectations. A more comprehensive list must also be compiled. **Interface** Inputs - digital, analogue, voltage range, protection Outputs - digital, analogue, drive level, protection Intra-communication interfaces - bus types, protocol Communication media and interface **Power Supply** Voltage Range Polarity Battery back-up
Product Target Market	**Product positioning** Product positioning in market. **Budget** Manufactured price calculated backwards from realistic market price.

Milestone 1 Meeting

Product Definition

Component Selection	Component Selection → Functionality, Interface, Power Supply Specifications, Other : EMI, Manufacturability, Reworks, Repairs, Budget Review
Message Structures	Message Structures → Complexity, Flexibility, Robustness, CPU Overheads, Packet size Review
Firmware Structure	Firmware Structure → Real Time OS, Scheduler, Task Engine Review
Enclosure material and size	
3rd Party developments - definitions	

Revised Interface specifications

Milestone 2 Meeting

Table 1. Stage 1 of the development cycle

Stage 2 - Design					
Time →					
Prototype Development					
Production Hardware					
Production Software					
Development jigs	Programming jig, downloading jig, RS232-TTL converters				
Hardware	Schematic design	PCB design	Prototype PCB	ICE development setup	
Development Software	Prototype Set-up Software			Prototype Exercise Software	
Firmware	Detail Protocol Design	Firmware Module Design		Firmware Module Integration	
Customer Software		Software Component Design		User Software	

Table 2. Stage 2 of the development cycle.

Stage 2 - Design					
Time →					
Product Development					
	Bed-of-nails (BON), Soak Test jig, Repair/Rework Jig				**Milestone 3 Meeting**
Production Software	Bed-of-nails (BON) PCB testing & part programming software, Soak Test Server software, Repair/Rework Fault finding software				
Development jigs					
Hardware	Schematic design revised	PCB design revised	Pre-production prototype	Prototype testing	
Development Software					
Firmware	Testing		Firmware Revision	Integrated Testing	
Customer Software	User Software Testing		Software Revision		

Table 2 (continued). Stage 2 of the development cycle.

Stage 3 - Design Revision

Time →

Sample Run	Test for manufacturing or assembly problems	Revision	Corrective Measures
Production QC Plan	Procedures for quality control. Batch testing. Random sample testing.	Revision	Corrective Measures
Production	FMEA - Hardware, Software	Revision	Corrective Measures
Production Manual	Final changes	Check for accuracy	Corrective Measures
Repair Manual	Final changes	Check for accuracy	Corrective Measures
Product	FMEA - Hardware, Firmware	Final Tests	Corrective Measures
Packaging	Design. Die-cut tools made.	Off-tool samples. Revision	Corrective Measures
Enclosure Tooling	Compatibility with assembly process.	Revision	Corrective Measures
Customer	FMEA - Software	Revision	Corrective Measures
Customer Manual	User manual, pocket guides, flyers, pamphlets	Check for accuracy	Corrective Measures
Installation Manual	Installer manual. Some minimal troubleshooting.	Check for accuracy	Corrective Measures
Bar Coded Labels	Design labels for major components as well as product itself.	Check for glue stability	Corrective Measures
Faceplates	Design. Die-cutting tools made.	Check for glue stability and legend correctness.	Corrective Measures
Marketing	Photography, artwork, helpdesk operator training, Web Site FAQ/Downloads		

Milestone 4 Meeting

Table 3. Stage 3 of the development cycle.

Stage 4 - Pre-Production				
Time →				
Off-tool Sample Run	Environmental testing, Functional testing, Failure mode testing.	Revision	Corrective Measures	**Milestone 5 Meeting**
Production QC Plan	Implementation.	Revision	Corrective Measures	
Production Manual	Wall charts, colour code charts.			
Repair Manual	Wall charts, test point charts, fault finding flow diagrams.			
Installation Manual	Complete Installation manual			
Installer Training	Product familiarity. Troubleshooting.			
Bureau Operator Training	Product and services familiarity. Troubleshooting.			
Cosmetic Checks	Barcodes, faceplates, burrs, other production marks. Packaging and shrink-wrapping.	Revision	Corrective Measures	

Table 4. Stage 4 of the development cycle.

Chapter 2 Product Planning

7. New Product Request

7.1 Product Description

Description

The product must be a wireless telecommunications device with an interface that would enable it to interact with its surroundings. The medium for communication will be the GSM.

Functionality

The minimum functionality is required. The main functions of the unit must be :

1) To maintain a stable communications medium.
2) To enable a user to change any unit parameter remotely.
3) To enable a user to read the user status.
4) To measure and condition certain input signals.
5) To effect certain output signals.
6) To store and report the unit status at selectable intervals under selectable conditions.

Interface

The unit must have, as a minimum specification, the following inputs :

1) A minimum of 5 analogue to digital converters with 0-5V input range and 8 bit resolution. Minimum sample period must be no longer than 1s.
2) The analogue inputs must each have individual high alarms as well as low alarms with their respective debounce times.
3) At least two digital inputs with TTL levels.
4) The digital inputs must be configurable to react to a low-going-high, or a high-going-low signal. The inputs must be debounced with configurable debounce times.
5) Mains power detection.

As outputs, the following is required :

1) A minimum of 2 TTL outputs capable of a drive of at least 10mA.
2) The outputs must be able to stream a selectable pattern.

The following additional specifications will increase the value of the product :

1) The analogue input values could be smoothed to reduce the effects of noise.
2) More digital inputs and outputs.
3) An event counter or frequency input.

Power Supply

The unit must be able to tolerate a wide range of input voltages without undue heat build-up. The unit must also be able to deliver a regulated voltage at a minimum current of 1A.
The unit must be protected against reverse polarities.
The unit must be able to operate from a battery back-up. An external charging circuit may be assumed.

7.2 Product Target Market

Product Positioning

This product is not intended to be a process controller or for large-scale sensing. It is instead, intended to be used where other products are too complicated and expensive. It will therefore, also be used by non-technical people.

Budget

In a distributor survey, it was found that a retail price of around R2500 would still be acceptable. Assuming a total mark-up of around 40%, this leaves a manufactured cost of around R1750. Some of the once-off costs, like development tools and production tools, can be amortised over the first 5000 units.

7.3 Milestone 1 Evaluation

The minimum specifications for the unit were compiled. These specifications include the interface specifications, and the unit power requirements. The functional and financial boundaries within which the unit must be designed were defined by the market requirements.

8. Product Definition

8.1 Component Selection

Communication Medium

The device to be used will be a GSM engine. This is the most expensive part in the product.

Microcontroller

The lowest cost microcontroller able to fulfil the task will be used. Since Wasp already owns the full development suites for both the Microchip and Motorola microcontrollers and some library modules from other existing products may be used as is, only these two manufacturers will be considered. As bulk buying can lead to considerable cost savings, versions of microcontrollers already used in large quantities by Wasp are preferred.

The two most used microcontrollers by Wasp are the Motorola 68HC916X1, and the Microchip PIC16C77. These two controllers will be compared to see if they are suitable for the purpose.

1) Functionality :

	Motorola 68HC916X1	Microchip PIC16C77
ROM	50k Flash	8k ROM
RAM	2k	392 bytes

2) Interface :

	Motorola 68HC916X1	Microchip PIC16C77
Analogue Inputs	Yes	Yes
Digital Inputs	Yes	Yes
TTL Outputs	Yes	Yes
UART	yes	yes

3) Power Supply :

	Motorola 68HC916X1	Microchip PIC16C77
5V	Yes	Yes
12V	Yes (flash programming)	No

4) Other Considerations :

	Motorola 68HC916X1	Microchip PIC16C77
EMI	Low	Low
IO Protection	Yes	Yes
Manufactureability	Difficult because of fine lead pitch on surface mount parts.	Surface mount quad flat pack
Reworks	Complicated by lead pitch	No problem
Repairs	Complicated by lead pitch	No problem
Upgradeablility	Yes - flash	No
Budget	~R80	~R50

There is a cost advantage of at least R30 in using the Microchip part.

Power Supply

The only way to ensure low heat dissipation over a wide range in input voltages, is by using a switch mode supply. A Linear Technologies LT1076 can supply up to 2A continuously. This part is already used in

another product that has passed FTA and CE testing, and has proven itself to be ideal for use with the GSM engine. To take advantage of bulk pricing, the same through-hole package style will be used.

Input Protection

The SP720 by Harris Semiconductor offers protection against voltage transients for up to 14 inputs (analogue or digital). It can clamp to both positive and negative supplies. It is also used in another product in a surface-mount package, so it will be beneficial to use it in higher quantities.

Real Time Clock

The PIC 16C77 has only 392 byte of RAM, which makes it very difficult to parse commands and messages. The Philips PCF8583 is a real time clock with an extra 250 bytes of RAM. This memory may be used as a scratch pad by the microcontroller. Communication with the chip is by means of the I2C bus, which only uses two lines. Its price is comparable to other real time clocks.

Memory

The unit will need some non-volatile memory to store data and its setup parameters. A 64kb I^2C E^2PROM part will have sufficient space for this application.

Revised Interface Specifications

With the selection of components as chosen in the previous selection exercise, the following revised interface specification is chosen :

1) 5 Analogue to digital inputs sampled at a resolution of 8 bits every 100ms. Full scale is 5V. A high, and a low alarm with separate debounce periods will be implemented. A four value rolling average for each channel will also be implemented as a software option.
2) 3 Digital inputs (TTL) with selectable trigger levels.
3) 1 Mains present input (TTL).
4) A 16 bit event counter with the option of a 1s clearing interval for frequency measurements up to 64kHz will be implemented. A high, and a low alarm with separate debounce periods will be used. A synchronising pulse will indicate the start of a measurement for ranging applications.
5) 4 TTL outputs with an 8 bit pattern, bit repeat number, and bit period will be implemented.
6) All inputs are to be configurable to govern the unit's action in the event of a trigger. These options include :
    - Report trigger immediately.
    - Log trigger and report with next message, or if buffer is full.
    - Report status on a timed basis (linked to the real time clock).
    - Log status on a timed basis, or if buffer is full (linked to the real time clock).

The parameters now associated with the interface, are

- *Analogue Inputs*

Parameter	Type	Range	Units
Analogue Value	Byte	0 - 255	(5 / 255) * (Input Voltage Divider Factor)
High Alarm	Byte	0 - 255	(5 / 255) * (Input Voltage Divider Factor)
High debounce Time	Byte	1 - 255	78ms
Low Alarm	Byte	0 - 255	(5 / 255) * (Input Voltage Divider Factor)
Low Debounce Time	Byte	1 - 255	78ms
Trip Event Value	Bit	0,1	1
High Alarm Enable	Bit	0,1	1
Low Alarm Enable	Bit	0,1	1
Averaging Enable	Bit	0,1	1
Regular Logging Enable	Bit	0,1	1
Regular Reporting Enable	Bit	0,1	1
Trigger Logging Enable	Bit	0,1	1

Digital Inputs

Parameter	Type	Range	Units
Debounce Time	Byte	1 - 255	40ms
Input Status Value	Bit	0,1	1
Input Event Value	Bit	0,1	1
Non-active polarity	Bit	0,1	1
Regular Logging Enable	Bit	0,1	1
Regular Reporting Enable	Bit	0,1	1
Trigger Logging Enable	Bit	0,1	1

Counter Input

Parameter	Type	Range	Units
Frequency/Counter Value	Word	1 - 65535	1 Hz / 1 count
High frequency alarm/ Max count alarm	Word	1 - 65535	1 Hz / 1 count
High frequency debounce Time	Byte	1 - 255	1s
Low frequency alarm	Word	1 - 65535	1 Hz
Low frequency debounce Time	Byte	1 - 255	1s
Frequency/Counter select	Bit	0,1	1
High frequency alarm enable/Max count alarm enable	Bit	0,1	1
Low frequency alarm enable	Bit	0,1	1
Regular Logging Enable	Bit	0,1	1
Regular Reporting Enable	Bit	0,1	1
Trigger Logging Enable	Bit	0,1	1

Power Detect

Parameter	Type	Range	Units
Power Status Value	Bit	0,1	1
Power Fail Event	Bit	0,1	1
Power Fail Alarm enable/disable	Bit	0,1	1
Debounce Time	fixed	5	5s fixed
Regular Logging Enable	Bit	0,1	1
Regular Reporting Enable	Bit	0,1	1
Trigger Logging Enable	Bit	0,1	1

TTL Outputs

Parameter	Type	Range	Units
Output Bit Pattern	Byte	0-255	1
Output Repeats	Byte	0 (infinite), 1-255	1
Output Bit Period	Byte	0-255	100ms
Output State	Bit	0,1	1
Regular Logging Enable	Bit	0,1	1
Regular Reporting Enable	Bit	0,1	1
Trigger Logging Enable	Bit	0,1	1

8.2 Message Structures

Communication Medium

There are currently four generally used methods to communicate using the GSM. These are :

1) Voice channel
2) Data channel
3) SMS (short message service) in Text Mode (7 bit data) with a maximum of 160 characters.
4) SMS (short message service) in PDU (packet data unit) mode (8 bit data) with a maximum of 140 bytes.

To compare these in terms of use as a data channel :

	Voice channel	Data channel	Text mode SMS	PDU mode SMS
Data rate	Very low	Very high	High for short messages	High for short messages
Error correction	None	Yes	Yes	Yes
Security	High	High	High	High
Effect of congested network	High	High	Very low	Very low
Maximum length	None	None	160 characters	140 bytes
Complexity	Low	Medium	High	Medium

To compare these in terms of cost per byte :

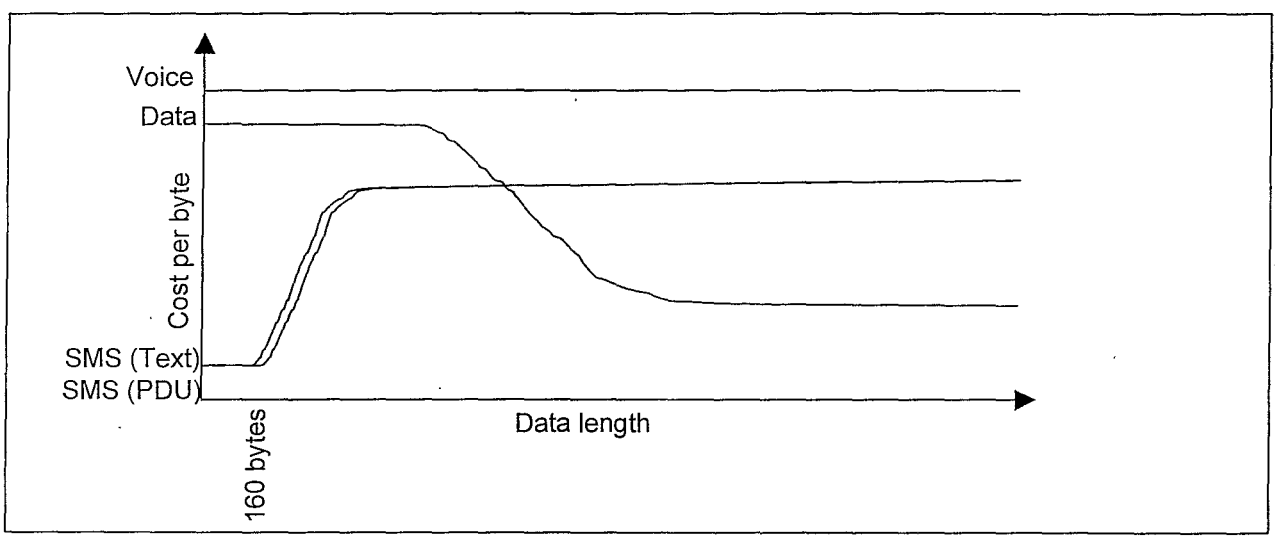


Figure 6. The relative costs of different channels of communications over the GSM.

For short data packets, the best solution is to use SMS in PDU mode.

Communication Protocol

The basic message structure for the outbound (mobile originating, or MO) and the inbound messages (mobile terminating, or MT) must be identical as far as identical message fields are concerned. This simplifies the message structure from a development point of view.

In order to keep the message structure as flexible as possible, a protocol version and command number is added. To keep the microcontroller code as simple as possible, all the different commands will also use the same basic structure. Fixed length messages introduce some level of data integrity, and also reduces software overhead.

All messages (both directions) will start with a header consisting of a message identifier (to identify this message as a GTel message), a version identifier (to identify the protocol version), and a sequence number (to identify messages on a per-message basis).

Standard Header				..Data..	Filler (32)
Message Identifier	Version Identifier	Sequence Number	Transmission Retries		
MI	VI	SN	TR	D	Filler

MI : MI will always be **TE** (two bytes) for GTel.
VI : The high nibble of VI (1 byte) contains a number from 0 - 15 that identifies the specific protocol version.
SN : The sequence number is an 8 bit value that will be increased on each transmission. It will roll over to zero after reaching the maximum value.
TR : This 8 bit value is the number of times the mobile unit has tried to send a message. In mobile terminated messages, its value is always 0.
D : Message data.
Filler : The rest of the message is filled with value 32 (ASCII space)

After the header (and the extra header parameters in the case of an MT message), a command or response record may be inserted. The structures of the records differ for mobile originated (MO) messages, and mobile terminated (MT) messages.

Firmware Structure

The architecture of the Microchip PIC microcontroller does not lend itself to a complicated operating system. To best accommodate the requirements, a combination of a scheduler and task engine will be implemented. An interrupt based time base schedules different tasks to be completed in the background. Command parsing is also done in the background. Tasks are to be issued by a state machine in a round robin style. If there are no tasks to be completed, other lower priority code must be executed. Tasks may also be activated by the background command parser.

A software watchdog function must monitor the different tasks completed, and reset the unit if any of the mission-critical tasks were not completed in good time.

Enclosure

The enclosure must be low cost, but also versatile from a board evolution point of view. Removable side plates enable one to change tooling for holes relatively cheaply since it only involves a new punch die - not a complete tool. Mild steel and aluminium enclosures are considered to be too expensive for the target market.

Dust and waterproofing for hostile environments will be achieved by encasing the unit inside an enclosure made for such purposes.

Third Party Developments

Third parties may contribute greatly to the volume in sales if they are allowed to develop applications for niche markets. These markets may include anything from industrial and vehicle security, moveable asset management, remote monitoring to home automation and security.

9. Milestone 2 Evaluation

The basic building blocks were defined. All the critical components were defined in an iterative manner. The message transmission medium was defined (GSM SMS in PDU mode) and the basic message protocol was defined for mobile originating (MO) messages as well as mobile terminating (MT) messages.

The required embedded software (firmware) structure was identified as a combination of a scheduler and a task engine.

A low-cost enclosure was chosen. It was decided that third parties may contribute to the value of the product. These parties will be invited to develop applications to increase their sales.

Chapter 3 Design

10. Prototype development

10.1 Development Jigs

Programming Jig

This jig connects the PCB with the PIC to a programmer for in-circuit programming after assembly. This jig will only be used for the pilot run. After the final changes to the PCB layout have been made, the board-specific insert for the bed-of-nails will be ordered.

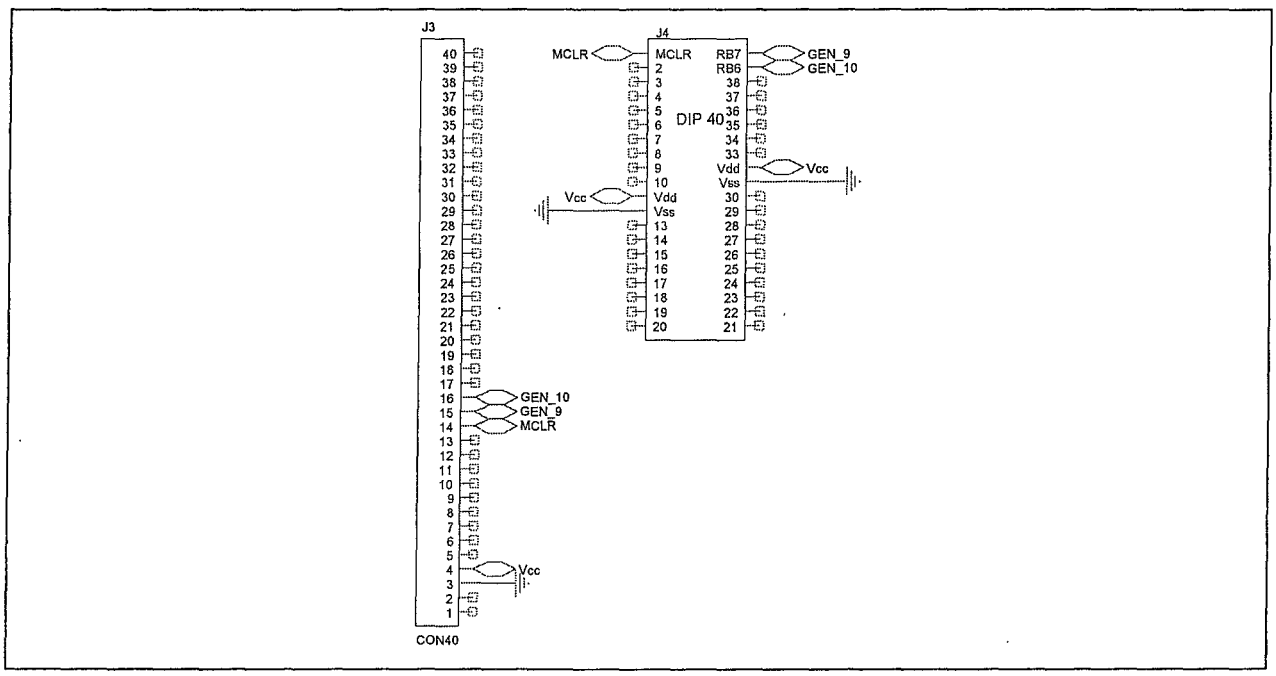


Figure 7. The programming jig.

Downloading Jig

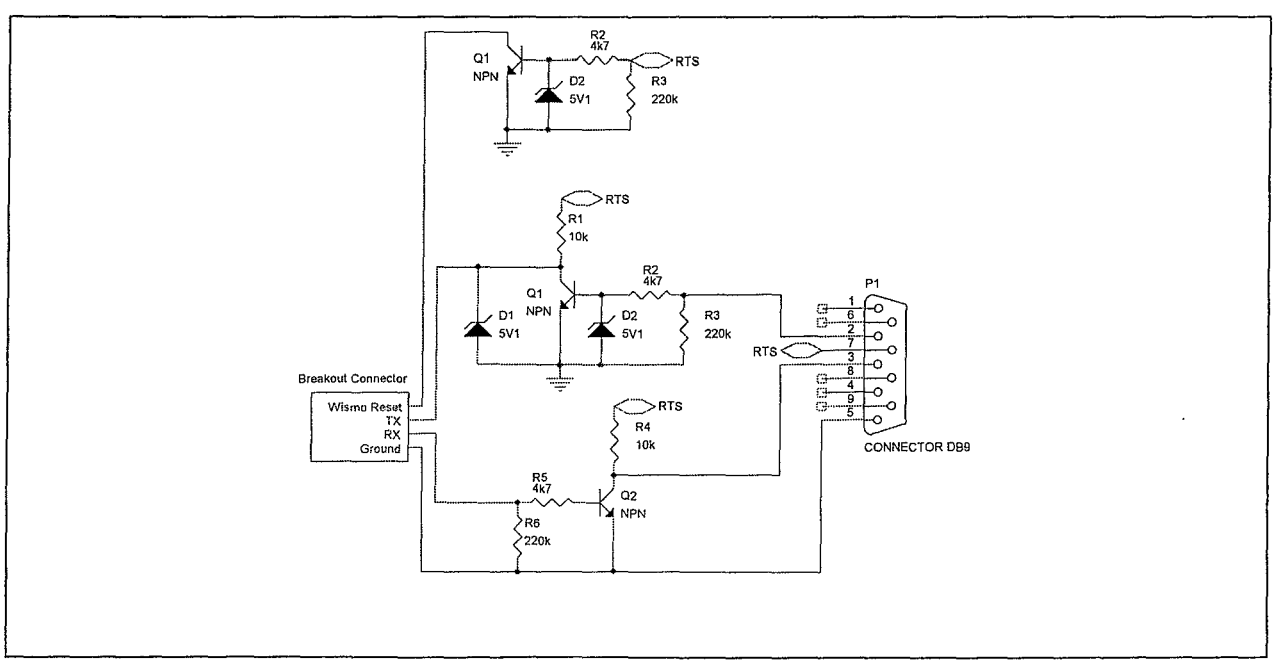


Figure 8. The jig used to download the setup data into the non-volatile memory.

This jig is used during development to program the non-volatile memory on board the unit. This memory will contain all the default set-up values, as well as other semi-permanent information like constants factors, constant arrays (like text strings), serial numbers, and start-up modes. This jig is a simple RS232<->TTL level converter that allows a host computer to interface with the PIC.

The downloading kernel in the PIC will be one of the first modules to be written.

Breakout Monitor Jig

To monitor the communication between the PIC and the GSM engine, a breakout jig is built with level conversion. This jig allows two lines (the RX and TX lines) to be monitored simultaneously on a host computer.

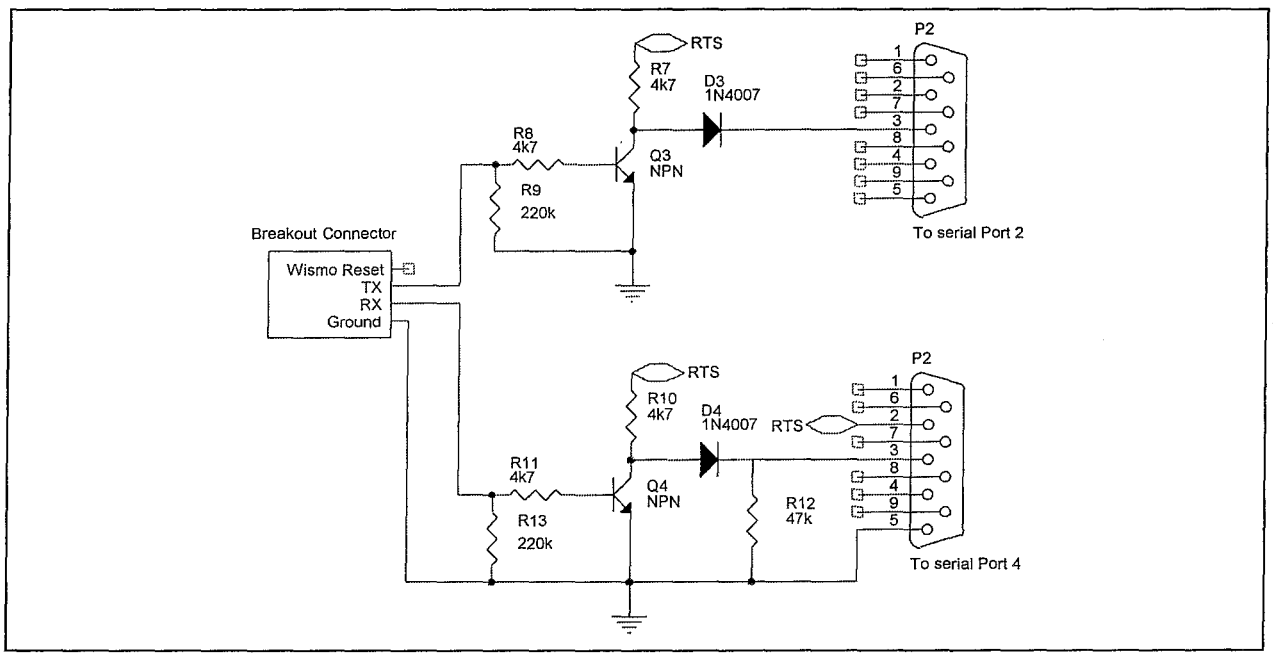


Figure 9. The breakout monitor jig used to monitor the communications.

10.2 Hardware

Schematic Design

See Appendix D.

PCB Design

See Appendix E.

Prototype PCB

A prototype PCB was made and tested to be true to the schematic.

10.3 Development Software

Prototype Set-up Software

This Windows-based software allows the user to download a standard profile into the non-volatile memory. During the code development, this memory may become corrupted, so an efficient way to refresh this memory can save considerable time.

A text file is created with the data arranged as 512 rows of 16 decimal values. After reading this file, the user establishes communications with the unit. This is done by sending the string "Start<CR>" to the unit. If the unit responds with "Ready<CR>", it is ready to receive the data.

The next byte sent by the host to the unit, is the byte value of the first decimal value in the data file. This value is written to address 0. The unit responds with "Ok<CR>" after the write operation is complete.

Because of the low data rate and the non-critical nature of the communications, there are no checksums. The address automatically increases after each write operation.

The user interface on the host computer is self-explanatory :

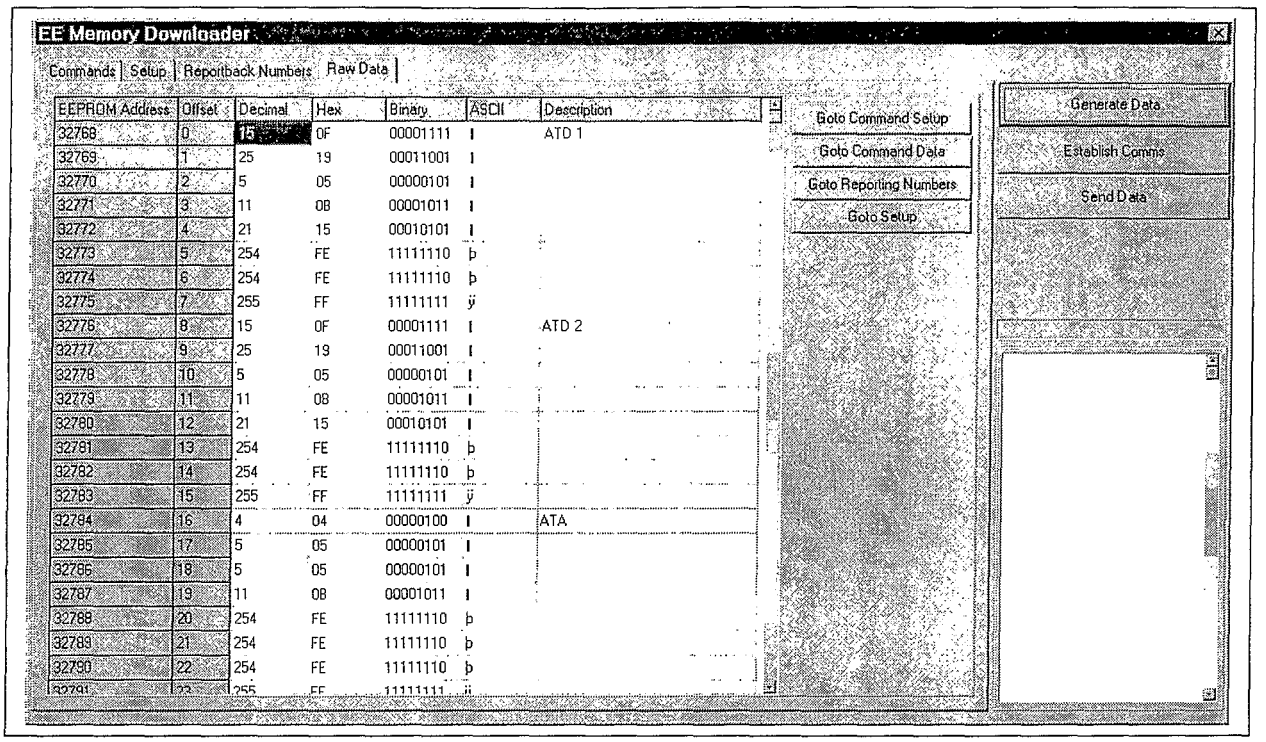


Figure 10. The host computer downloading application.

10.4 Firmware & Customer Software

Detail Protocol Design - Message Path

The unit must be able to intelligently manage its outbound communications path. Two issues are important : the service centre to be used (service centre address), and the target number (usually the base unit). In the event of a failed transmission, the unit will try a different service centre, as well as target number.

One of four options may be set on the mobile terminating (MO) message :

- Return data to Number 1, using Service Centre 1
- Return data to Number 2, using Service Centre 2
- Return data to Number 3, using Service Centre 3
- Return data to Sender, using Service Centre 1

In the event of a failure with the return option being one of the first three, the next pair will be selected. After roll-over to the first pair, the unit will wait 1 minute before starting on the first pair. If the message is to be returned to the sender but fails, the unit will try all the Service Centres from Service Centres 1 to 3, and roll over if it still fails to Service Centre 1. After a successful transmission, the unit will default to Number 1, Service Centre 1 for unsolicited responses.
The return path is specified.

Detail Protocol Design - Mobile Originating (MO)

Information contained in the header, includes :

- Tell the base station what the product type is ('TE' means GTel.)
- Identify the version number of the protocol. This would allow for protocol enhancements in future.
- Return the sequence number used to keep track of multiple messages.
- Return the number of re-transmissions the unit had to effect to send the message successfully. This could identify poor installations or areas of bad coverage.
- Identify the type of message.

A device identification field (ID) leaves room for future expansion. The high 5 bits in this byte give a total of 32 different devices, while the low 3 bits allow 8 identical devices to respond. The interface on the GTel is device 1 (0x08). A three byte date/time stamp is inserted after the command number :

These three bytes take the format :

Byte 1								Byte 2								Byte 3							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Weekday				Month of Year				Day of Month								Minute of the day							

Weekday : 0 - 6
Month of year : 1 - 12
Day of Month : 1 - [29,30,31]
Minute of the Day : 1 - 1441

The only messages to be expected from a unit, are :

1) *Memory Read - Command 2*
This command is used to read any location in the non-volatile memory. The maximum number of bytes that can be read at one time, is 64 bytes.

The structure is :

Standard Header					Record Data									Filler
MI		VI	SN	TR	RL	ID	Cmd	DTS			Address		Length	Data	T	
'T'	'E'	0h00	0h26	0h07	0h08	0h02	0h02	DTS 1	DTS 2	DTS 3	0h01	0h15	0h01	X	'*'	0h20

The protocol version (VI) is 0h00.
The sequence number (SN) is 0h26.
The message was re-transmitted (TR) 4 times
The record length (RL) is 0h0B.
The target device (ID) is Device 1, number 0 (0h08).
The command (Cmd) is 0h02 (Memory read)
The Date/Time stamp is DTS
The target address to read from (Address), is 0h0115.
Four bytes (Length) must be read.
The Terminator (T), is '*'.

2) *Status Request - Command 4*
This command is used to get the status of the interface. Three request masks determine which of Device 1's values or parameters are returned.

Request Mask 1	
7	Return Analogue 1 Value
6	Return Analogue 1 Setup
5	Return Analogue 2 Value
4	Return Analogue 2 Setup
3	Return Analogue 3 Value
2	Return Analogue 3 Setup
1	Return Analogue 4 Value
0	Return Analogue 4 Setup

Request Mask 2	
7	Return Analogue 5 Value
6	Return Analogue 5 Setup
5	Return Digital Input Setup 1
4	Return Digital Input Setup 2
3	Return Digital Input States / Events
2	Return Digital Input 1 Debounce Time
1	Return Digital Input 2 Debounce Time
0	Return Digital Input 3 Debounce Time

Request Mask 3	
7	Return Frequency/Counter Value
6	Return Frequency/Counter Setup
5	Return Output States
4	Return Output 1 Setup

3	Return Output 2 Setup
2	Return Output 3 Setup
1	Return Output 4 Setup

Analogue Setup Bytes Returned (if Requested)	
bit	Function
7	High Alarm
6	High Debounce Time
5	Low Alarm
4	Low Debounce Time
3	Setup Bit Values : 7 : Analogue High Alarm Enable / Disable 6 : Analogue Low Alarm Enable / Disable 5 : Analogue Average Enable / Disable 4 : Analogue Trip Logging Enable / Disable 3 : Analogue RTC Logging Enable / Disable 2 : Analogue RTC Reporting Enable / Disable

Digital Input States/Events Bits Returned	
bit	Function
7	Current Power State
6	Digital Input 3 State
5	Digital Input 2 State
4	Digital Input 1 State
3	Power Down Events
2	Digital Input 3 Events
1	Digital Input 2 Events
0	Digital Input 1 Events

Frequency/Counter Setup Bytes Returned	
bit	Function
7	High frequency alarm/ Max count alarm (high byte)
6	High frequency alarm/ Max count alarm (low byte)
5	High frequency debounce Time
4	Low frequency alarm (high byte)
3	Low frequency alarm (low byte)
2	Low frequency debounce Time
1	Setup Bit Values : 7 : Counter High Alarm Enable 6 ; Counter Low Alarm Enable 5 : Counter Mode 4 : Counter Trip Logging Enable 3 : Counter RTC Reporting Enable 2 : Counter RTC Logging Enable

Output States Bit Values	
bit	Function
7	
6	
5	
4	
3	Output 4 State
2	Output 3 State
1	Output 2 State
0	Output 1 State

Output Setup Values Returned for each Output
Output Bit Pattern
Output Bit Period
Output Bit Repeats

An example of a status returned message is :

Standard Header								Record Data								Filler
MI		VI	SN	TR	RL	ID	Cmd	DTS			RM 1 - RM 3			Data	T	
'T'	'E'	0h00	0h00	0h00	0h08	0h08	0h04	DTS1	DTS2	DTS3	RM1	RM2	RM3	Data	'*'	0h20

3) *Unsolicited Response - Command 9*
This message is sent by the unit without being requested by the user.

It is received when :

- An input was triggered with the 'Trip Reporting' parameter enabled for that input.
- The input was to be logged because the 'Trip Logging' parameter is enabled, but the message space was too small to store the message. After the log is sent, the data is stored.
- An input's state was reported on a timed basis because the 'RTC reporting' parameter for that input is set.
- An input was to be logged on a timed basis because the 'RTC logging' parameter was set, but the message space was too small for the log. After the log is sent, the data is stored.

If one input is reported, all the inputs with 'Trip Logging' enabled will also be sent in that message.

Bytes 1 through 6 will always be present in this message. The other bytes will be present as indicated by the relevant masks.

Byte	Outputs	
1	Input Trip Mask 1	7 : Reserved 6 : Reserved 5 : Frequency/Counter Event 4 : Analogue 5 Event 3 : Analogue 4 Event 2 : Analogue 3 Event 1 : Analogue 2 Event 0 : Analogue 1 Event
2	Input Trip Mask 2	7 : Power State 6 : Input 3 State 5 : Input 2 State 4 : Input 1 State 3 : Power Down Event 2 : Input 3 Event 1 : Input 2 Event 0 : Input 1 Event
3	Input Trip Report Enable Mask	7 : Reserved 6 : Reserved 5 : Frequency/Counter Trip Reporting Enabled 4 : Analogue 5 Trip Reporting Enabled 3 : Analogue 4 Trip Reporting Enabled 2 : Analogue 3 Trip Reporting Enabled 1 : Analogue 2 Trip Reporting Enabled 0 : Analogue 1 Trip Reporting Enabled
4	Input Trip Log Enable Mask	7 : Reserved 6 : Reserved 5 : Frequency/Counter Trip Logging Enabled 4 : Analogue 5 Trip Logging Enabled 3 : Analogue 4 Trip Logging Enabled 2 : Analogue 3 Trip Logging Enabled 1 : Analogue 2 Trip Logging Enabled 0 : Analogue 1 Trip Logging Enabled

5	Digital Input Setup Mask 1	7 : Input 3 Trip Polarity 6 : Input 2 Trip Polarity 5 : Input 1 Trip Polarity 4 : Digital Inputs Trip Logging 3 : Digital Inputs RTC Reporting 2 : Digital Inputs RTC Logging 0 : Frequency/Counter mode (1 = frequency mode)
6	Digital Input Setup Mask 2	7 : Power Down Trip Reporting 6 : Power Down Trip Logging 5 : Power Down RTC Reporting 4 : Power Down RTC Logging 3 : Input 1 Trip Reporting 2 : Input 2 Trip Reporting 1 : Input 3 Trip Reporting
7	Analogue 1	Value
8	Analogue 2	Value
9	Analogue 3	Value
10	Analogue 4	Value
11	Analogue 5	Value
12	Frequency Counter Value	High Byte
13	Frequency Counter Value	Low Byte

Example of an Unsolicited Response

Standard Header					Record Data														Filler
MI		VI	SN	TR	RL	ID	Cmd	ITM1	DTS			ITM2	TREM	TLEM	DISM1	DISM2	AD2	T	
'T'	'E'	0h20	0h00	0h0:	0h08	0h08	0h09	0h03	DTS1	DTS2	DTS3	0h0A	0h08	0h00	0h1E	0h00	0h4E	'*'	0h20

Symbol	Name	Hex	Binary	Detail
RL	Record Length :	0h0B		Record length 11 bytes total
ID	Device ID :	0h08		The target device is Device 1, number 0
Cmd	Command Number	0h09		
DTS	Date/Time Stamp			
ITM1	Input Trip Mask 1 :	0h03	0b00000011	7 : 6 : 5 : Frequency/Counter - No event 4 : Analogue 5 - No event 3 : Analogue 4 - No event 2 : Analogue 3 - No event 1 : Analogue 2 - No event 0 : Analogue 1 - Trip Event
ITM2	Input Trip Mask 2 :	0h0A	0b00001010	7 : Power State - Power OK 6 : Input 3 State - Low 5 : Input 2 State - High 4 : Input 1 State - Low 3 : Power Down - Event 2 : Input 3 - No event 1 : Input 2 - Event 0 : Input 1 - No event
TREM	Trip Reporting Enable Mask	0h08	0b00000001	7 : Reserved 6 : Reserved 5 : Frequency/Counter Trip Reporting Disabled 4 : Analogue 5 Trip Reporting Disabled 3 : Analogue 4 Trip Reporting Disabled 2 : Analogue 3 Trip Reporting Disabled 1 : Analogue 2 Trip Reporting Disabled 0 : Analogue 1 Trip Reporting Enabled

TLEM	Trip Logging Enable Mask	0h00	0b00000000	7 : Reserved 6 : Reserved 5 : Frequency/Counter Trip Logging Disabled 4 : Analogue 5 Trip Logging Disabled 3 : Analogue 4 Trip Logging Disabled 2 : Analogue 3 Trip Logging Disabled 1 : Analogue 2 Trip Logging Disabled 0 : Analogue 1 Trip Logging Disabled
DISM1	Digital Input Setup Mask 1	0h1E	0b00011100	7 : Input 3 Polarity - Trip on High 6 : Input 2 Polarity - Trip on High 5 : Input 1 Polarity - Trip on High 4 : Digital Inputs Trip Logging - Enabled 3 : Digital Inputs RTC Reporting - Enabled 2 : Digital Inputs RTC Logging - Enabled 0 : Frequency/Counter mode (1 : counter mode)
DISM2	Digital Input Setup Mask 1	0h00	0b11111111	7 : Power Down Trip Reporting - Enabled 6 : Power Down Trip Logging - Enabled 5 : Power Down RTC Reporting - Enabled 4 : Power Down RTC Logging - Enabled 3 : Input 1 Trip Reporting - Enabled 2 : Input 2 Trip Reporting - Enabled 1 : Input 3 Trip Reporting - Enabled
AD2	Analogue 2 Value	0h4E	0b01001110	Analogue value : 1.53V
T	Terminator	'*'		

Detail Protocol Design - Mobile Terminating (MT)

Information contained in the header, includes :

- Tell the base station what the product type is ('TE' means GTel.)
- Identify the version number of the protocol. This would allow for protocol enhancements in future.
- Issue a sequence number used to keep track of multiple messages. The next message with this number, is matched to this transmission.
- Return the number of re-transmissions the unit had to effect to send the message successfully. This could identify poor installations or areas of bad coverage.
- Identify the type of message.

A device identification field (ID) leave room for future expansion. The high 5 bits in this byte give a total of 32 different devices, while the low 3 bits allow 8 identical devices to respond. The interface on the GTel is device 1 (0x08).

The only messages to be expected from a unit, are :

1) Memory Write - Command 1

This command is used to read any location in the non-volatile memory. The maximum number of bytes that can be written at one time, is 64 bytes.

The structure is :

Standard Header					Extra Header Data			Record Data									Filler
MI		VI	SN	TR	PW		RT	RL	ID	Cmd	Address		Length	Data			T	
'T'	'E'	0h00	0hAA	0h00	0hA3	0h3F	0h01	0h0A	0h08	0h01	0h01	0h15	0h04	0h01	0h02	0h03	'*'	0h20

The protocol version (VI) is 0h00.
The sequence number (SN) is 0hAA.
The password (PW) is 0hA33F.
The target number (RT.1 and RT.0) is number 2, SMSC 2
The record length (RL) is 0h0A (10 decimal).

The target device (ID) is Device 1, number 0 (0h08).
The command (Cmd) is 0h01 (Memory write)
The target address to read from (Address), is 0h0115.
Three bytes (Length) must be read.
The Terminator (T), is '*'.
Filler characters (0h20) are used to fill up the message to a total length of 140 bytes.

2) Reset Unit - Command 3

This command will be used to reset different variables, counters, or flags. It could also be used to reset the unit itself. The V-values contains the bitmask for the reset targets. The X-value is always the XOR'ed value of the V-value.

Bit Mask 1		Bit Mask 2		Bit Mask 3		Bit Mask 4	
V1	X1	V2	X2	V3	X3	V4	X4

The targets of the reset bits are :

Bit	Bit Mask 1	Bit Mask 2	Bit Mask 3	Bit Mask 4
7	TBA	TBA	TBA	TBA
6	TBA	TBA	TBA	TBA
5	TBA	TBA	TBA	TBA
4	TBA	TBA	TBA	TBA
3	Reset Unit	TBA	TBA	TBA
2	Reset GSM engine	TBA	TBA	TBA
1	Digital & power triggers	TBA	TBA	TBA
0	Analogue & counter triggers	TBA	TBA	TBA

** Shaded areas contain compulsory fields*

Standard Header					Extra Header Data			Record Data												Filler
MI		VI	SN	TR	PW		RT	RL	ID	Cmd	Mask 1		Mask 2		Mask 3		Mask 4		T	
'T'	'E'	0h00	0hAA	0h00	0hA3	0h3F	0h01	0h0A	0h08	0h03	0h00	0hFF	0h00	0hFF	0h00	0hFF	0h00	0hFF	'*'	0h20

The protocol version (VI) is 0h00.
The sequence number (SN) is 0hAA.
The password (PW) is 0hA33F.
The target number (RT.1 and RT.0) is number 2, SMSC 2
The record length (RL) is 0h0A (10 decimal).
The target device (ID) is Device 1, number 0 (0h08).
The command (Cmd) is 0h03 (Unit Reset)
The Terminator (T), is '*'.
Filler characters (0h20) are used to fill up the message to a total length of 140 bytes.

3) Status Request - Command 4

The 3 request masks determine which of Device 1's values or parameters are returned.

Request Mask 1	
7	Return Analogue 1 Value
6	Return Analogue 1 Setup
5	Return Analogue 2 Value
4	Return Analogue 2 Setup
3	Return Analogue 3 Value
2	Return Analogue 3 Setup
1	Return Analogue 4 Value

Request Mask 2	
7	Return Analogue 5 Value
6	Return Analogue 5 Setup
5	Return Digital Input Setup 1
4	Return Digital Input Setup 2
3	Return Digital Input States / Events
2	Return Digital Input 1 Debounce Time
1	Return Digital Input 2 Debounce Time

0	Return Analogue 4 Setup

0	Return Digital Input 3 Debounce Time

Request Mask 3	
7	Return Frequency/Counter Value
6	Return Frequency/Counter Setup
5	Return Output States
4	Return Output 1 Setup
3	Return Output 2 Setup
2	Return Output 3 Setup
1	Return Output 4 Setup

Analogue Setup Bytes Returned (if Requested)	
bit	Function
7	High Alarm
6	High Debounce Time
5	Low Alarm
4	Low Debounce Time
3	Setup Bit Values : 7 : Analogue High Alarm Enable / Disable 6 : Analogue Low Alarm Enable / Disable 5 : Analogue Average Enable / Disable 4 : Analogue Trip Logging Enable / Disable 3 : Analogue RTC Logging Enable / Disable 2 : Analogue RTC Reporting Enable / Disable

Digital Input States/Events Bits Returned	
bit	Function
7	Current Power State
6	Digital Input 3 State
5	Digital Input 2 State
4	Digital Input 1 State
3	Power Down Events
2	Digital Input 3 Events
1	Digital Input 2 Events
0	Digital Input 1 Events

Frequency/Counter Setup Bytes Returned	
bit	Function
7	High frequency alarm/ Max count alarm (high byte)
6	High frequency alarm/ Max count alarm (low byte)
5	High frequency debounce Time
4	Low frequency alarm (high byte)
3	Low frequency alarm (low byte)
2	Low frequency debounce Time
1	Setup Bit Values : 7 : Counter High Alarm Enable 6 ; Counter Low Alarm Enable 5 : Counter Mode 4 : Counter Trip Logging Enable 3 : Counter RTC Reporting Enable 2 : Counter RTC Logging Enable

Output States Bit Values	
bit	Function
7	
6	
5	
4	
3	Output 4 State
2	Output 3 State
1	Output 2 State

0	Output 1 State

Output Setup Values Returned for each Output
Output Bit Pattern
Output Bit Period
Output Bit Repeats

An example of a complete message is :

Standard Header					Extra Header Data			Record Data								Filler
MI		VI	SN	TR	PW		RT	RL	ID	Cmd	RM 1 - RM 3			T		
'T'	'E'	0h00	0h42	0h00	0hAB	0hCD	0h01	0h08	0h08	0h04	RM1	RM2	RM3	'*'		0h20

Filler characters (0h20) are used to fill up the message to a total length of 140 bytes.

The protocol version (VI) is 0h00.
The sequence number (SN) is 0h42.
The password (PW) is 0hABCD.
The target number (RT.1 and RT.0) is number 2, SMSC 2
The record length (RL) is 0h0A (10 decimal).
The target device (ID) is Device 1, number 0 (0h08).
The command (Cmd) is 0h04 (Status request)
The Terminator (T), is '*'.
Filler characters (0h20) are used to fill up the message to a total length of 140 bytes.

4) Status Modify 1 - Command 5

This message is used to change any of the parameters relating to :

- Analogue inputs set-up.
- Digital inputs set-up.

The other parameters may be changed through the Modify Status 2 command.
Note that it is not fixed length. All the bytes represented by the shaded boxes must always be present.

The format is as follows :

Byte	Analogue inputs			
1	Analogue 1 Change Mask	Which parameters (next 6bytes) must change Bit 7 : byte 2 Bit 6 : byte 3 Bit 5 : byte 4 Bit 4 : byte 5 Bit 3 : bytes 6&7		Analog 1
2	High Alarm	Upper limit		
3	High debounce Time	Upper limit debounce time		
4	Low Alarm	Lower limit		
5	Low Debounce Time	Lower limit debounce time		
6	Setup bits ON	Settings to be enabled	7 : High Alarm enable 6 : Low Alarm Enable 5 : Averaging Enable 4 : Regular Logging Enable 3 : Regular Reporting Enable 2 : Trigger Logging Enable	
7	Setup bits OFF	Settings to be disabled		

29	Analogue 5 Change Mask	Which parameters (next 6bytes) must change Bit 7 : byte 30 Bit 6 : byte 31 Bit 5 : byte 32 Bit 4 : byte 33 Bit 3 : bytes 34&35		Analog 5
30	High Alarm	Upper limit		
31	High debounce Time	Upper limit debounce time		
32	Low Alarm	Lower limit		
33	Low Debounce Time	Lower limit debounce time		
34	Set up bits ON	Settings to be enabled	7 : High Alarm enable 6 : Low Alarm Enable 5 : Averaging Enable 4 : Regular Logging Enable 3 : Regular Reporting Enable 2 : Trigger Logging Enable	
35	Set up bits OFF	Settings to be disabled		

Byte	Digital Inputs	
36	Digital Change Mask 1	Setting to Change : 7 : Change Digital Input 3 Debounce Time 6 : Change Digital Input 2 Debounce Time 5 : Change Digital Input 1 Debounce Time 4 : Change Digital Input 3 Trip Reporting 3 : Change Digital Input 2 Trip Reporting 2 : Change Digital Input 1 Trip Reporting 1 : Change Digital Inputs Trip Logging 0 : Value of Digital Inputs Trip Logging
36	Digital Change Mask 2	Bit Values : 7 : Change Digital Input 3 Trigger Polarity 6 : Change Digital Input 2 Trigger Polarity 5 : Change Digital Input 1 Trigger Polarity 4 : Change Digital Input RTC Reporting 3: Change Digital Input RTC Logging
36	Digital Change Mask 3	Bit Values : 7 : Value of Digital Input 3 Polarity 6 : Value of Digital Input 2 Polarity 5 : Value of Digital Input 1 Polarity 4 : Value of Digital Input 3 Trip Reporting 3 : Value of Digital Input 2 Trip Reporting 2 : Value of Digital Input 1 Trip Reporting 4 : Value of Digital Input RTC Reporting 3 : Value of Digital Input RTC Logging
37	Digital Input 1 Debounce	Debounce Time for Input 1 (if change bit is set)
38	Digital Input 2 Debounce	Debounce Time for Input 2 (if change bit is set)
39	Digital Input 3 Debounce	Debounce Time for Input 3 (if change bit is set)

** Shaded areas contain compulsory fields*

An example of a complete message is :

Standard Header					Extra Header Data		Record Data																Filler
MI		VI	SN	TR	PW		RT	RL	ID	Cmd	Data											T	
'T'	'E'	0h00	0h42	0h00	0hAB	0hCD	0h01	0h0F	0h08	0h05	A1	X1	A2	X2	A3	X3	A4	X4	A5	D1	D2	'*'	0h20

The protocol version (VI) is 0h00.
The sequence number (SN) is 0h42.
The password (PW) is 0hABCD.
The target number (RT.1 and RT.0) is number 2, SMSC 2
The record length (RL) is 0h0A (10 decimal).
The target device (ID) is Device 1, number 0 (0h08).
The command (Cmd) is 0h05 (Status Modify 1)

With :

A1 = 128
A2 = 128
A3 = 128
A4 = 128
A5 = 0
D1 = 0
D2 = 0, the high alarms for Analogue inputs 1 through 4 are changed to :

New Analogue 1 High alarm : X1
New Analogue 2 High alarm : X2
New Analogue 3 High alarm : X3
New Analogue 4 High alarm : X4

The Terminator (T), is '*'.
Filler characters (0h20) are used to fill up the message to a total length of 140 bytes.

5) Status Modify 2 - Command 6

This message is used to change any of the parameters relating to :

- Outputs set-up.
- Mains power sensing set-up.
- Frequency meter/event counter set-up

The other parameters may be changed through the Modify Status 1 command.
Note that it is fixed length. All the bytes represented by the shaded boxes must always be present.

The format is as follows :

Byte	Outputs	
1	Output 1 Change mask	Which parameters (next 3 bytes) must change 7 : Change Output 1 Bit Pattern 6 : Change Output 1 Bit Repeats 5 : Change Output 1 Bit Period 4 : Change Digital Input 1 State to bit 3's value 3 : Digital Output 1 State (only valid if bit 4 is set)
2	Output 1 Bit Pattern	The pattern to be clocked out (8 bits max)
3	Output 1 Bit Repeats	The number of bit repeats 1-254 / forever
4	Output 1 Bit Period	The period for each bit (100ms resolution)
13	Output 4 Change mask	Which parameters (next 3 bytes) must change 7 : Change Output 4 Bit Pattern 6 : Change Output 4 Bit Repeats 5 : Change Output 4 Bit Period 4 : Change Digital Output 4 State to bit 3's value 3 : Digital Output 4 State (only valid if bit 4 is set)
14	Output 4 Bit Pattern	The pattern to be clocked out (8 bits max)
15	Output 4 Bit Repeats	The number of bit repeats 1-254 / forever
16	Output 4 Bit Period	The period for each bit (100ms resolution)

Byte	Power Sense	
17	Power Sense Change mask	Which parameters must change (with values) 7 : Change Power Down Trip Reporting 6 : Change Power Down Trip Logging 5 : Change Power Down RTC Reporting 4 : Change Power Down RTC Logging 3 : New value for Power Down Trip Reporting 2 : New value for Power Down Trip Logging 1 : New value for Power Down RTC Reporting 0 : New value for Power Down RTC Logging

Byte	Counter Input	
18	Frequency/Counter Change Mask 1	Which reporting/logging parameters must change 7 : Change Frequency/Counter Mode 6 : Change Frequency/Counter RTC Reporting 5 : Change Frequency/Counter RTC Logging 4 : Change Frequency/Counter Trip Logging 3 : New value for Frequency/Counter Mode 2 : New value for Frequency/Counter RTC Reporting 1 : New value Frequency/Counter RTC Logging 0 : New value for Frequency/Counter Trip Logging
18	Frequency/Counter Change Mask 2	Which parameters (next 6 bytes) must change 7 : Change Frequency/Counter High Alarm (2 bytes) 6 : Change Frequency/Counter High Alarm Debounce Time 5 : Change Frequency/Counter High Alarm Reporting 4 : New Value for Frequency/Counter High Alarm Reporting 3 : Change Frequency/Counter Low Alarm (2 bytes) 2 : Change Frequency/Counter Low Alarm Debounce Time 1 : Change Frequency/Counter Low Alarm Reporting 0 : New Value for Frequency/Counter Low Alarm Reporting
19	Frequency/Counter High Alarm	High byte of 16 bit value
20	Frequency/Counter High Alarm	Low byte of 16 bit value
21	Frequency/Counter High Alarm Debounce Time	High Alarm Debounce Time
22	Frequency/Counter Low Alarm	High byte of 16 bit value
23	Frequency/Counter Low Alarm	Low byte of 16 bit value
24	Frequency/Counter Low Alarm Debounce Time	Low Alarm Debounce Time

** Shaded areas contain compulsory fields*

An example of a complete message is :

Standard Header					Extra Header Data			Record Data					Filler
MI		VI	SN	TR	PW		RT	RL	ID	Cmd	Data	T	
'T'	'E'	0h00	0h42	0h00	0hAB	0hCD	0h01	0h0F	0h08	0h06		'*'	0h20

The protocol version (VI) is 0h00.
The sequence number (SN) is 0h42.
The password (PW) is 0hABCD.
The target number (RT.1 and RT.0) is number 2, SMSC 2
The record length (RL) is 0h0A (10 decimal).
The target device (ID) is Device 1, number 0 (0h08).
The command (Cmd) is 0h06 (Status Modify 2)
The 24 data bytes contain the changed instructions as explained in the table
The Terminator (T), is '*'.
Filler characters (0h20) are used to fill up the message to a total length of 140 bytes.

10.5 Firmware Module Design

All the major functions of the unit were coded in modules and tested separately. These modules are to be used in a program structure that will combine a scheduler as well as a task engine. The simplified structure will be :

		Subsection	Function	Input Flags	Return Flags	Return values
Interrupt Service Routine (ISR)	1	UART Character received	Handles character from input buffer and clears any errors. The character may follow one of two paths : • Command parsing (normal program flow) • Circular buffer (Temporary storage of data in the real time clock scratch memory)		* Character received	* Command / Reply from GSM engine.
	2	UART Character transmitted	Handles the outgoing data stream. If the outgoing buffer is not empty, the next character is transmitted.	* Start transmit-ting	* Busy flag	
	3	Time Base : 16ms	Handles all the shorter time-base tasks, like sampling : • Select next A/D channel for sampling, sample input, smooth input, and debounce if out of limits. • Sample and debounce digital inputs		* Digital input trigger	* Digital input events and states A/D values and events
	4	Time Base : 100ms	Handles the output bit streams. Shifts the patterns, sets delay periods, and switches the outputs.	* Output active		
	5	Time Base : 1s	Sample mains power supply and debounce if not present. Gives a synch pulse for ranging. Reads the 16 bit event counter/frequency counter. Checks if the reading is within bounds, and debounces it if out of limits. Handles some second-based time-outs : • Transmission time-out • Response time-out Handles some delays : • Message delay • GSM engine housekeeping cycle • State machine delays		* Transmission time out * Response time-out * Message delay expired * GSM engine housekeeping required * State machine delay expired	* Event counter / frequency counter values and events * Mains power status and event
Start-up Code	1	Set-up micro	Handles the set-up of : • Microcontroller options • Interrupts • I/O Port direction • Timer pre-scaler • A/D options • Communications parameters Also resets all the variables into a known state.			

	2	Load default values	Loads default values for : • Digital inputs set-up • Analogue inputs set-up • Event counter/frequency counter set-up • Output set-up • Password • Real time clock reporting times			
	3	Download set-up values	If compiled in, this code will download the set-up data from a host with the download jig.			

		Procedure	Function	Variables Passed	Variables Returned
Other Program Modules (OPM)	1	Load_String()	Loads a block of data from the real time clock, or the non-volatile memory	Address Length	Data
	2	Save_String()	Saves a block of data to the real time clock, or the non-volatile memory	Address Length Data	
	3	Load_IO_Parameters()	Load default values for set-up from non-volatile memory		
	4	Save_IO_Parameters()	Saves new default values for set-up in non-volatile memory		
	5	Set up_RTC()	Sets up the real time clock for proper operation		
	6	Get_Time()	Gets the time from the real time clock.		Processed time and date stamp (three bytes)
	7	Issue_Command(Command_Number)	Sends a command to the GSM engine	Command number	Response from GSM engine
	8	Resolve_Message_Header()	Checks incoming message for validity and integrity		Validity
	9	Get_Next_Command()	Gets the next command in the message from the memory.		Reads the command into RAM. Calculates response length.
	10	Send_The_Log()	Sends the logged data using the path specified by the data		Transmission outcome
	11	ASCII_2_HEX()	Changes an ASCII BCD value into its absolute equivalent	Two HEX characters	Absolute value
	12	BIN2BCD(BCDBIN_In)	Changes an absolute value into 3 digit BCD value.	Absolute value	3 characters
	13	BCD2BIN(BCDBIN_In)	Changes a double digit BCD value into an absolute one.	8 bit BCD	Absolute value

UART Character Received (ISR.1)

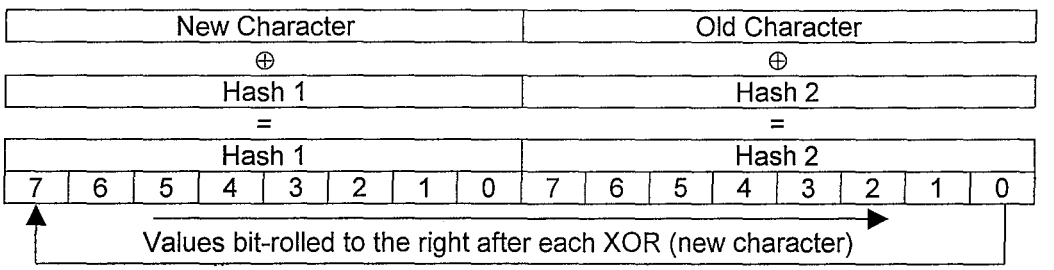
1) Error Handling

```
if (RCSTA.1==1)                                          // If an error occurs,
        {RCSTA.CREN = 0;                                 // reset it
        RCSTA.CREN  = 1;}                                //
```

2) Calculate Hash function

This section of code calculates a hash function from the incoming characters. This function's output is used for command parsing. This particular function has been found to be very fast and accurate.

With every new character received, the new and old characters are XOR'ed with hash values 1 and 2. The hash values are then bit-rolled to the right with roll-over to the MSB of Hash 1.




```
if (PIR1.RCIF)
        {
        PIR1.RCIF    = 0;                                // Reset interrupt
        GotChar      = 1;                                // Send flag to MPL

        RX_Older  = RX_Old;                              // Store the new character,
        RX_Old    = RCREG;                               // but keep the older one too.
        RRF(RX_Hash_1);                                  // Calculate the new hash
        RRF(RX_Hash_2);                                  // function with the new and
        RX_Hash_1bits.7 = STATUS.0;                      // the previously received character
        RX_Hash_1 ^= RX_Old;                             // Complete the
        RX_Hash_2 ^= RX_Older;                           // hash function.

        RX_Char_Counter++;                               // This counter ensures that the hash
                                                         // function is limited to a certain
                                                         // number of characters.

switch(RX_Hash_1)                                        // To save time, we only check for a
        {                                                // match on the first byte. The second
                                                         // byte is checked if the first matched.

        case 87 :                                        // +CREG: 0,1
                {                                        // This response indicates that the
                if (RX_Hash_2 == 66)                     // unit is logged onto the network
                        {                                // and ready to transmit or receive
                        Response_Number = _Response_NetWorkOk;  // messages at any time.
                        break;                           //
                        }
                }
        case 63 :                                        // >
                {                                        // This response indicates that
                if (RX_Hash_2 == 59)                     // the header for the
                        {                                // transmission command has
                        Response_Number = _Response_SMS_Header_OK; // been accepted.
                        break;                           // The data may now be sent.
                        }
                }
        case 135 :                                       // +CMGS: XX
                {                                        // This response indicates
                if (RX_Hash_2 == 74)                     // that the message has been
                        {                                // successfully transmitted
                        Response_Number = _Response_SMSOk;  //
                        break;                           //
                        }
                }
        case 82 :                                        // +CMS ERROR: 38
                {                                        // This response indicates
                if (RX_Hash_2 == 89)                     // a possible problem with the
                        {                                // service centre.
                        Response_Number = _Response_SMSC_Is_Sick;  //
                        break;                           //
                        }
                }
        case 108 :                                       // OK
                {                                        // The "OK" response was received.
                if ((RX_Hash_2 == 202)&&(Response_Number != _Response_Message))
                        {                                // in the middle of a response
                        Response_Number = _Response_Ok;  //
                        break;                           //
                        }
```

```
                }
        case 171 :                                                      // OK
                {                                                       // The "OK" response was received
                if (RX_Hash_2 == 146)                                   //
                        {                                               //
                        Response_Number = _Response_Ok;                 //
                        break;                                          //
                        }
                }
        case 124 :                                                      // +CME ERROR: 3
                {                                                       // This response is received when
                if (RX_Hash_2 == 35)                                    // a pin number is issued to a
                        {                                               // sim card with its pin disabled
                        Response_Number = _Response_PinNotNeeded;       // or if the pin has already been
                        break;                                          // accepted
                        }
                }
        case 174 :                                                      // ERROR
                {                                                       // An unspecified error prevented
                if (RX_Hash_2 == 48)                                    // the command from being successful.
                        {                                               //
                        Response_Number = _Response_Error;              //
                        RX_Hash_2 = 0;                                  //
                        RX_Hash_1 = 0;                                  //
                        break;                                          //

                        }
                }
        case 199 :                                                      // +CMTI: "SM",X
                {                                                       // A new message was received
                if (RX_Hash_2 == 50)                                    //
                        {                                               //
                        Response_Number = _Response_NewMessage;         //
                        RX_Hash_2 = 0;                                  //
                        RX_Hash_1 = 0;                                  //
                        break;                                          //

                        }
                }
        case 157 :                                                      // +CMS ERROR: 321
                {                                                       // There is no message in
                if (RX_Hash_2 == 30)                                    // the position just read
                        {                                               //
                        Response_Number = _Response_NoMessage;          //
                        RX_Hash_2 = 0;                                  //
                        RX_Hash_1 = 0;                                  //
                        RX_Char_Counter = 0;                            //
                        break;                                          //

                        }
                }
        case 7 :                                                        // +CMGR:
                {                                                       // The first part of the new
                if (RX_Hash_2 == 10)                                    // message has been received by
                        {                                               // the unit. The rest of the characters
                        RX_Char_Counter = 0;                            // will be the body of the message.
                        Response_Number      = _Response_Message;       // This data must be saved in the
                        MessageReadingStatus = _Starting;               // real time clock memory
                        break;                                          //
                        }                                               //
                }
        }

if ((RX_Old==10)&&(RX_Older==13))                                       // A new line will start
        {                                                               // after <CR><LF>
        RX_Hash_1 = 0;                                                  // Reset the hash functions
        RX_Hash_2 = 0;                                                  // and the received
        RX_Char_Counter = 0;                                            // character counter.
        }                                                               //

if (RX_Char_Counter>15)                                                 // If there is still
        {                                                               // no match after 15 characters,
        RX_Hash_1 = 0;                                                  // there must be something wrong,
        RX_Hash_2 = 0;                                                  // so reset the counter.
        RX_Char_Counter = 0;                                            //
        }

}
```

UART Character Transmitted (ISR.2)

1) Transmitting from the transmission buffer

```
if ((Ignore_CTS)||(GISMO_CTS == 0))                                    // Hardware flow control
        {                                                              //
        PIR1.TXIF = 0;                                                 // Clear interrupt

        if ((I2C_Buffer_Index<Total_bytes_to_Transmit)&&(I2C_Buffer_Index<I2C_Buffer_Max))
                {                                                      // If there are still characters
                TXREG = I2C_Buf[I2C_Buffer_Index];                     // left over in the buffer,
                I2C_Buffer_Index++;                                    // take the next one and start
                }                                                      // the transmission
        else
                {
                while (TXSTA.TRMT==0);                                 // If the buffer is empty,
                TXSTA.TXEN = 0;                                        // make sure the last
                PIE1.TXIE = 0;                                         // character from the FIFO is
                Transmitting = 0;                                      // also transmitted.
                }                                                      // Clear the busy flag when done
        }
```

Time Base : 20ms (ISR.3)

```
T2CON.2          = 0;                                                  //
PIR1.TMR2IF      = 0;                                                  // Reset Interrupt
T2CON.2          = 1;                                                  // Reset Timer 2

_20mseconds++;
PR2              = 211;
```

1) Analogue Input Sampling

Analog_Channel (0-4) is the current channel being sampled.
Analog_Control (0,1) is an interleaving control variable. When its value is 1, the following operations are done :

- The current channel will be converted and the value read.
- If averaging is enabled, the last four values will be averaged.
- The high alarm and low alarm are checked and debounced if necessary.
- The relevant events will be set if necessary.

When its value is 0, the following operations are done :

- The next channel is selected.
- The input sampling is started.

```
if (Analog_Control==1)
        {
        ADCON0 |= 0b00000100;                                                  // Start Conversion

        Analog_ValA[Analog_Channel] = Analog_ValB[Analog_Channel];             // The oldest value
        Analog_ValB[Analog_Channel] = Analog_ValC[Analog_Channel];             // is discarded
        Analog_ValC[Analog_Channel] = Analog_ValD[Analog_Channel];             // while the newest
        Analog_ValD[Analog_Channel] = ADRES;                                   // four values are kept

        if (Testbit(Analog_Config[Analog_Channel],Analog_Average_Enable))      // If the averaging
                {                                                              // option is set, the
                Analog_Temp = Analog_ValA[Analog_Channel];                     // unit will add the
                Analog_Temp = Analog_Temp + Analog_ValB[Analog_Channel];       // last four values and
                Analog_Temp = Analog_Temp + Analog_ValC[Analog_Channel];       // divide it by four to
                Analog_Temp = Analog_Temp + Analog_ValD[Analog_Channel];       // get a running average
                Analog_Temp = Analog_Temp>>2;                                  // that will be used as
                Analog_Value[Analog_Channel] = Analog_Temp_Low;                // the valid value.
                }
        else
                {
                Analog_Value[Analog_Channel] = ADRES;                          // Use it without averaging
                }

        ADCON0 = 0b00000000;                                                   // Done


        if                                                                     // Check if high alarm was
exceeded
((Analog_Value[Analog_Channel]>Analog_High_Alarm[Analog_Channel])&&(Analog_Config[Analog_Channel]&0b10000000))
                {
                Analog_Debounce_Count_Low[Analog_Channel] = 0;                 // Clear low debouncing
                Analog_Debounce_Count_High[Analog_Channel]++;                  // Debounce high
                if (Analog_Debounce_Count_High[Analog_Channel]==Analog_Debounce_High[Analog_Channel])
                        {
                        switch (Analog_Channel)                                // Set relevant event
                                {                                              // It must trigger once only
                                case 0 : {if (LevelWasOK1.0) {Analog1_Trip; LevelWasOK1.0=0;} break;}
                                case 1 : {if (LevelWasOK1.1) {Analog2_Trip; LevelWasOK1.1=0;} break;}
                                case 2 : {if (LevelWasOK1.2) {Analog3_Trip; LevelWasOK1.2=0;} break;}
```

```
                                case 3 : {if (LevelWasOK1.3) {Analog4_Trip; LevelWasOK1.3=0;} break;}
                                case 4 : {if (LevelWasOK1.4) {Analog5_Trip; LevelWasOK1.4=0;} break;}
                                }
                        }
                }
        else if                                                         // Check if low alarm was violated
((Analog_Value[Analog_Channel]<Analog_Low_Alarm[Analog_Channel])&&(Analog_Config[Analog_Channel]&0b01000000))
                {
                Analog_Debounce_Count_High[Analog_Channel] = 0;         // Clear high debouncing
                Analog_Debounce_Count_Low[Analog_Channel]++;            // Debounce low
                if (Analog_Debounce_Count_Low[Analog_Channel]==Analog_Debounce_Low[Analog_Channel])
                        {
                        switch (Analog_Channel)                         // Set relevant event
                                {                                       // It must trigger once only
                                case 0 : {if (LevelWasOK1.0) {Analog1_Trip; LevelWasOK1.0=0;} break;}
                                case 1 : {if (LevelWasOK1.1) {Analog2_Trip; LevelWasOK1.1=0;} break;}
                                case 2 : {if (LevelWasOK1.2) {Analog3_Trip; LevelWasOK1.2=0;} break;}
                                case 3 : {if (LevelWasOK1.3) {Analog4_Trip; LevelWasOK1.3=0;} break;}
                                case 4 : {if (LevelWasOK1.4) {Analog5_Trip; LevelWasOK1.4=0;} break;}
                                }
                        }

                }
        else
                {
                Analog_Debounce_Count_High[Analog_Channel] = 0;         // No alarm violations, so
                Analog_Debounce_Count_Low[Analog_Channel] = 0;          // both debounce counters
                        switch (Analog_Channel)                                 // cleared
                        {                                               //
                        case 0 : {LevelWasOK1.0=1; break;}              // The level is normal,
                        case 1 : {LevelWasOK1.1=1; break;}              // so an alarm may
                        case 2 : {LevelWasOK1.2=1; break;}              // trigger again.
                        case 3 : {LevelWasOK1.3=1; break;}              //
                        case 4 : {LevelWasOK1.4=1; break;}              //
                        }
                }
        Analog_Control = 0;                                             // Sample next time…
        }
else if (Analog_Control==0)                                             // Sample the next input
        {                                                               //
        Analog_Channel++;                                               // Get next channel
        if (Analog_Channel>4)                                           // from all the channels
                {                                                       // 0 to 4
                Analog_Channel = 0;                                     //
                }                                                       //
        ADCON1  = 0b00000010;                                           // Analog inputs
        ADCON0  = Analog_Channel<<3;                                    // Desired channel
        ADCON0 |= 0b10000000;                                           // Clock
        ADCON0 |= 0b00000001;                                           // Start Sampling
        Analog_Control = 1;                                             // Convert next time…
        }
```

2) Digital Input Sampling

```
Digital1_State = PORTD.5;                                               // Digital Input 1
Digital2_State = PORTD.4;                                               // Digital Input 2
Digital3_State = PORTD.3;                                               // Digital Input 3
PORTD = PORTD_SHADOW;                                                   // Shadow register to overcome
                                                                        // the red-modify-write effect

if (Digital1_State==Digital_Input_1_Polarity)                           // Check if the input
{                                                                       // is at trigger polarity
if (Digital_Debounce_Count[0]<Digital_Debounce[0])                      // if it is, then debounce
        {                                                               // this input. Check if it was ok
        Digital_Debounce_Count[0]++;                                    // before triggering now
        if
((Digital_Debounce_Count[0]==Digital_Debounce[0])&&(LevelWasOK2.0)&&(Digital_Input_1_Trip_Reporting))
{Digital1_Trip;LevelWasOK2.0 = 0;}                                      // An event generated - don't
        }                                                               // trigger again
}                                                                       //
else {Digital_Debounce_Count[0] = 0;LevelWasOK2.0 = 1;}                 // Level is ok - ok to trigger
too.

if (Digital2_State==Digital_Input_2_Polarity)                           // Check if the input
{                                                                       // is at trigger polarity
if (Digital_Debounce_Count[1]<Digital_Debounce[1])                      // if it is, then debounce
        {                                                               // this input. Check if it was ok
        Digital_Debounce_Count[1]++;                                    // before triggering now
        if
((Digital_Debounce_Count[1]==Digital_Debounce[1])&&(LevelWasOK2.1)&&(Digital_Input_2_Trip_Reporting))
{Digital2_Trip;LevelWasOK2.1 = 0;}                                      // An event generated - don't
        }                                                               // trigger again
}                                                                       //
else {Digital_Debounce_Count[1] = 0;LevelWasOK2.1 = 1;}                 // Level is ok - ok to trigger
too.

if (Digital3_State==Digital_Input_3_Polarity)                           // Check if the input
{                                                                       // is at trigger polarity
if (Digital_Debounce_Count[2]<Digital_Debounce[2])                      // if it is, then debounce
        {                                                               // this input. Check if it was ok
        Digital_Debounce_Count[2]++;                                    // before triggering now
        if
((Digital_Debounce_Count[2]==Digital_Debounce[2])&&(LevelWasOK2.2)&&(Digital_Input_3_Trip_Reporting))
{Digital3_Trip;LevelWasOK2.2 = 0;}                                      // An event generated - don't
        }                                                               // trigger again
}                                                                       //
```

```
else {Digital_Debounce_Count[2] = 0;LevelWasOK2.2 = 1;}                    // Level is ok - ok to trigger
too.
```

Time Base : 100ms (ISR.4)

1) Output Bit Stream

```
Outputs_100ms_Delay++;                                                    // Count the 20ms cycles
if (Outputs_100ms_Delay>5)                                                // 20ms * 5 = 100ms
        {
        Outputs_100ms_Delay = 0;

        if (Output1_Counter) Output1_Counter--;                           // Bit delay counter
        if ((Outputs_State.0)&&(Output1_Counter==0))                      // Next bit only shifts
                {                                                         // if delay is 0
                Output1_Counter = Output1_Delay;                          // Reload the bit delay counter
                RLF(Output1_Pattern);                                     // Roll the pattern
                Output1 = STATUS.0;                                       // Get the next state...
                Output1_Pattern.0 = STATUS.0;                             // Check if it must be
                if (Output1_Repeats<255) Output1_Repeats--;               // output, or not
                if (!Output1_Repeats)    Outputs_State.0 = 0;             //
                }
        if (Output2_Counter) Output2_Counter--;                           // Bit delay counter
        if ((Outputs_State.1)&&(Output2_Counter==0))                      // Next bit only shifts
                {                                                         // if delay is 0
                Output2_Counter = Output2_Delay;                          // Reload the bit delay counter
                RLF(Output2_Pattern);                                     // Roll the pattern
                Output2 = STATUS.0;                                       // Get the next state...
                Output2_Pattern.0 = STATUS.0;                             // Check if it must be
                if (Output2_Repeats<255) Output2_Repeats--;               // output, or not
                if (!Output2_Repeats)    Outputs_State.1 = 0;             //
                }
        if (Output3_Counter) Output3_Counter--;                           // Bit delay counter

        if ((Outputs_State.2)&&(Output3_Counter==0))                      // Next bit only shifts
                {                                                         // if delay is 0
                Output3_Counter = Output3_Delay;                          // Reload the bit delay counter

                RLF(Output3_Pattern);                                     // Roll the pattern
                Output3 = STATUS.0;                                       // Get the next state...
                Output3_Pattern.0 = STATUS.0;                             // Check if it must be
                if (Output3_Repeats<255) Output3_Repeats--;               // output, or not
                if (!Output3_Repeats)    Outputs_State.2 = 0;             //
                }
        if (Output4_Counter) Output4_Counter--;                           // Bit delay counter

        if ((Outputs_State.3)&&(Output4_Counter==0))                      // Next bit only shifts
                {                                                         // if delay is 0
                Output4_Counter = Output4_Delay;                          // Reload the bit delay counter

                RLF(Output4_Pattern);                                     // Roll the pattern
                Output4 = STATUS.0;                                       // Get the next state...
                Output4_Pattern.0 = STATUS.0;                             // Check if it must be
                if (Output4_Repeats<255) Output4_Repeats--;               // output, or not
                if (!Output4_Repeats)    Outputs_State.3 = 0;             //
                }
        if (!Outputs_State.0) Output1 = 0;                                // Switch off if disabled
        if (!Outputs_State.1) Output2 = 0;                                // Switch off if disabled
        if (!Outputs_State.2) Output3 = 0;                                // Switch off if disabled
        if (!Outputs_State.3) Output4 = 0;                                // Switch off if disabled
        }
```

Time Base : 1s (ISR.5)

```
if (_20mseconds > 50)                                                   //
        {                                                               // counts 50 * 20ms
        seconds++;                                                      // = 1 second
        _20mseconds       = 0;                                          //
        PR2               = 196;                                        // Time correction every second.
```

1) Mains Power Status Check

```
Power_State     = PORTA.4;                                              // Power Supply Sampled
        if (!Power_State)                                               // if power fails…
                {                                                       //
                PowerDwn_Counter++;                                     // …debounce it
                if ((PowerDwn_Counter == 20)&&(LevelWasOK2.3))          // If power down for 20 seconds
                        {                                               // and it was OK before,
                        PowerDwn_Trip;                                  // trigger a power down event
                        LevelWasOK2.3 = 0;                              // It must be ok before it will
                        }                                               // trigger again
                }                                                       //
        else                                                            //
                {                                                       //
                PowerDwn_Counter = 0;                                   // The power is ok now, so it
                LevelWasOK2.3 = 1;                                      // can trigger on power fail
                }                                                       //
```

2) Event Counter / Frequency Counter

```
        PORTB.0  = 1;                                                   // Synch pulse for ranging

        Counter_Interrupt_Cycles = 0;
        Counter_Value = Timer1_Value;                                   // Get 16 bit value

        PORTB.0  = 0;                                                   // Synch pulse for ranging

        if (Counter_Mode == Frequency_Mode)                             // Frequency counting mode
                {                                                       // The counter must be reset unit
                Timer1_Value = 0;                                       // next measurment in 1 second
                if ((Counter_Value > Counter_High_Alarm)&&(Counter_High_Alarm_Enable))
                        {                                               // Input exceeded the high alarm
                        Counter_Debounce_Count_Low = 0;                 // Debounce this alarm
                        if (Counter_Debounce_Count_High<255) Counter_Debounce_Count_High++;
                        if (Counter_Debounce_Count_High==Counter_Debounce_High)
                                {                                       // High alarm debounced
                                if (LevelWasOK1.5)                      // and the frequency was ok before
                                        {                               // going too high, so an
                                        Counter_Trip;                   // event is generated
                                        LevelWasOK1.5 = 0;              // No more triggers until level is
ok
                                        }
                                }
                        }                                               //
                else if ((Counter_Value < Counter_Low_Alarm)&&(Counter_Low_Alarm_Enable))
                        {                                               // Input lower than low alarm
                        Counter_Debounce_Count_High = 0;                // Debounce low alarm
                        if (Counter_Debounce_Count_Low<255) Counter_Debounce_Count_Low++;
                        if (Counter_Debounce_Count_Low==Counter_Debounce_Low)
                                {                                       // Low alarm debounced
                                if (LevelWasOK1.5)                      // and the level was ok before
                                        {                               // going too low, so an
                                        Counter_Trip;                   // event is triggered
                                        LevelWasOK1.5 = 0;              // No more triggers until level is
ok
                                        }
                                }
                        }
                else                                                    // The value was within the
                        {                                               // alarm boundaries, so the
                        Counter_Debounce_Count_Low  = 0;                // high and low
                        Counter_Debounce_Count_High = 0;                // debounce counters are reset
                        LevelWasOK1.5 = 1;                              // It can trigger again.
                        }
                }
        else
                {                                                       // It is in event counter mode
                if (Counter_Value > Counter_High_Alarm)                 // Check if event has exceeded
                        {                                               // the high alarm.
                        Counter_Trip;                                   // Yes it has - report
                        Timer1_Value = 0;                               // and start again
                        }
                }
```

3) Timeouts and Delays

```
if (MessageDelayInSeconds)                      // This is the counter that
        {                                       // delays the transmission with
        MessageDelayInSeconds--;                // 1min between roll-overs of
        }                                       // number/service centres

if (Wismo_Housekeep_Timer)                      // This timer initiates a
        {                                       // housekeeping cycle every
        Wismo_Housekeep_Timer--;                // 5 seconds to refresh settings
        }                                       //
```

```
if (TX_timeouts)                                    // This timer signals a
        {                                           // timeout during transmission
        TX_timeouts--;                              //
        }                                           //

if (Response_timeouts)                              // This timer signals a
        {                                           //timeout when waiting for
        Response_timeouts--;                        // the GSM engine to respond
        }                                           //

if (Wismo_State_Timer)                              // This is a general
        {                                           // purpose timer used
        Wismo_State_Timer--;                        // in the main program loop
        }                                           //
```

Set up Micro (SC.1)

```
void Initialize() {

        OPTION          = 0b11001111;               // Set the timer pre-scaler.

        RCSTA           = 0b10010000;               // UART enabled, continuous RX enabled
        TXSTA.BRGH      = 0;                        // Non-high speed mode
        SPBRG           = 17;                       // 9600 baud
        PIE1.RCIE       = 1;                        // RX interrupt enabled

        Transmit_Retries = 0;                       // No TX yet
        Ignore_CTS = 0;                             // Hardware flow control

        RX_Char_Counter  = 0;                       // Clear all the variables
        RX_Hash_1        = 0;                       // related to the command
        RX_Hash_2        = 0;                       // parsing
        TMR2             = 0;                       // Timer 2 used for
        PR2              = 211;                     // interrupt - based
        T2CON            = 0b01111111;              // time base
        PIE1.TMR2IE      = 1;                       //

        T1CON            = 0b00000011;              // Timer 1 used for
        Timer1_Value     = 0;                       // event/frequency
        Counter_Value    = 0;                       // counting

        TRISA            = 0b11111111;              // Port direction : PORTA
        TRISB            = 0b11111100;              // Port direction : PORTB
        TRISC            = 0b10110001;              // Port direction : PORTC
        TRISD            = 0b00111111;              // Port direction : PORTD
        TRISE            = 0b00000111;              // Port direction : PORTE

        Outputs12_Direction_Set;                    // Set TTL outputs 1 and 2 to low impedance
        Outputs34_Direction_Set;                    // Set TTL outputs 3 and 4 to low impedance

        Output1_Counter  = 0;                       // By resetting these counters
        Output2_Counter  = 0;                       // we ensure that the outputs
        Output3_Counter  = 0;                       // will start outputting their
        Output4_Counter  = 0;                       // bitstreams synchronised
        Output_Bit_Counter = 0;                     // Start with bit 0

        Analog_Channel   = 4;                       // The next channel will be 0
        Analog_Control   = 0;                       // Analogue set-up

        Trip1            = 0;                       // Clear all triggers
        Trip2            = 0;                       // Clear all triggers
        ReportTheInput   = 0;                       // Clear all triggers

        LogPosition      = 0;                       // Start at the beginning of the log
        LogFree          = 135;                     // with the full space left

        RTC_Alarm_Event  = 0;                       // Have to check the time first…

        MessageReadingStatus = _NotReading;         // Not busy reading a message

        Current_Service_Centre = __Select_Service_Centre1;
        Response_Targets = _Return_To_Number_1;     // Default path is number 1/service centre 1

        EE_Setup_Write_Enable = 0;                  // For data integrity

        INTCON           = 0b11000000;              // Enable global interrupts
  }
```

Download set-up values (SC.2)

```
#ifdef DOWNLOAD                                     // Only if compiled in

        while ((RX_Hash_1!=30)&&(RX_Hash_2!=165)); // Wait for 'Start' string on the com port
        counter1H=0;                                // Begin at address 0
        while (counter1H<2048)                      // Download 2k for now
                {                                   //
                I2C_Buf[0]              = 'O';      // Acknowledge
                I2C_Buf[1]              = 'k';      // the incoming data
                I2C_Buf[2]              = 13;       // with "Ok"
                Total_bytes_to_Transmit = 3;        //
```

```
                                                    //
        Ignore_CTS                  = 1;            // Send the string
        I2C_Buffer_Index            = 0;            //
        Transmitting                = 1;            //
        TXSTA.TXEN                  = 1;            //
        PIE1.TXIE                   = 1;            //
        while(Transmitting)                         //
                {                                   //
                CLRWDT();                           //
                }                                   //
        TXSTA.TXEN = 0;                             //
        PIE1.TXIE = 0;                              //
        TXSTA.TXEN = 0;                             //
        PIE1.TXIE = 0;                              //
                                                    //
        GotChar    = 0;                             // Wait for next data byte
        while (GotChar==0);                         //
                                                    //
        I2C_Buf[0]  = RX_Old;                       // Take this byte
        I2C_address = counter1H + 32768;            // and write it at the
        I2C_bytes = 1;                              // next address
        EE_Setup_Write_Enable = 1;                  //
        Save_String();                              //
        counter1H++;                                //
        }
#endif
```

Load String (OPM.1)

This is a library module, and the underlying code has been proven. The module uses the redundant bit in the 10 bit I²C addressing scheme (all addresses > 32768) to decide whether to use a 10 bit, or 7 bit address. If a 7 bit address is to be used, only one address byte is sent. If the address is above 32768, two bytes are sent.

```
void    Load_String(void)
        {
        Open_EEPROM_4_Write();                      // The device memory counter is set to the
                                                    // required writing address
        Open_EEPROM_4_Read();                       // The device is set to read mode

        I2C_Buffer_Index = 0;                       // Store the data from the bottom in the
                                                    // RAM buffer (64 max)
        while (I2C_bytes>0)                         // Read the required number of bytes
                {                                   //
                delay_I2C();                        // Inter-character delay
                I2C_bits=8;                         // Read 8 bits
                I2C_data = 0;                       //
                DataUp();                           // Release line
                while (I2C_bits>0)                  //
                        {                           //
                        delay_I2C();                // Inter-bit delay
                        ClockUp();                  // Clock in the data
                        delay_I2C();                // Time for bus to settle
                        I2C_data.0 = PORTD.1;       // Read the value
                        I2C_bits--;                 // Next please
                        RLF(I2C_data);              // Shove the bit into the data byte
                        ClockDown();                // Done with bit
                        }                           //
                RRF(I2C_data);                      // All 8 bits read
                I2C_Buf[I2C_Buffer_Index] = I2C_data;   // Put the data in the RAM buffer
                delay_I2C();                        //
                ACK();                              // Acknowledge the byte
                delay_I2C();                        //
                I2C_Buffer_Index++;                 // Next RAM location
                if (I2C_bytes) {I2C_bytes--;}       // Next byte of data
                }                                   //
        I2C_Stop();                                 // Stop the bus
        }
```

Save String (OPM.2)

This is a library module, and the underlying code has been proven. The module uses the redundant bit in the 10 bit I²C addressing scheme (all addresses > 32768) to decide whether to use a 10 bit, or 7 bit address. If a 7 bit address is to be used, only one address byte is sent. If the address is above 32768, two bytes are sent.

```
void    Save_String(void)
        {                                                   //
        I2C_Buffer_Index = 0;                               // Begin reading the RAM buffer from 0
        while   (  (I2C_bytes>0)                            // There must be something to write…
                &&(  (I2C_address < 254)                    // Writing allowed anywhere in the RTC
                   ||(I2C_address > _End_of_Setup)          // Not writing  in the non-volatile setup…
                   ||(EE_Setup_Write_Enable)))              // …except if expressly stated
                {                                           //
                Open_EEPROM_4_Write();                      // Open the device for writing at the
                                                            // required address
                I2C_data = I2C_Buf[I2C_Buffer_Index];       // Load value
                if (I2C_address==0) I2C_data &= 0b11001111; // Ensures RTC will never enter test mode
                Write_byte();                               // Write it
```

```
            if (I2C_bytes) {I2C_bytes--;}                    //
            I2C_Buffer_Index++;                              // Next byte
            I2C_address++;                                   // Next data position from RAM
            I2C_Stop();                                      // Next address
            }                                                // Stop the bus
                                                             //
                                                             //
        EE_Setup_Write_Enable = 0;                           // Disable the write enable for set-up
        }
```

Load IO Parameters (OPM.3)

```
void    Load_IO_Parameters(void)
        {
        I2C_address = _Setup;                    // Starting address for set-up data
        I2C_bytes   = 63;                        // Read 63 bytes
        Load_String();                           // Load it now

        Analog_High_Alarm[0]     = I2C_Buf[0];   // Analogue 1 High alarm
        Analog_Debounce_High[0]  = I2C_Buf[1];   // Analogue 1 High Alarm Debounce
        Analog_Low_Alarm[0]      = I2C_Buf[2];   // Analogue 1 Low Alarm
        Analog_Debounce_Low[0]   = I2C_Buf[3];   // Analogue 1 Low Alarm Debounce
        Analog_Config[0]         = I2C_Buf[4];   // Analogue 1 Set-up

        Analog_High_Alarm[1]     = I2C_Buf[5];   // Analogue 2 High alarm
        Analog_Debounce_High[1]  = I2C_Buf[6];   // Analogue 2 High Alarm Debounce
        Analog_Low_Alarm[1]      = I2C_Buf[7];   // Analogue 2 Low Alarm
        Analog_Debounce_Low[1]   = I2C_Buf[8];   // Analogue 2 Low Alarm Debounce
        Analog_Config[1]         = I2C_Buf[9];   // Analogue 2 Set-up

        Analog_High_Alarm[2]     = I2C_Buf[10];  // Analogue 3 High alarm
        Analog_Debounce_High[2]  = I2C_Buf[11];  // Analogue 3 High Alarm Debounce
        Analog_Low_Alarm[2]      = I2C_Buf[12];  // Analogue 3 Low Alarm
        Analog_Debounce_Low[2]   = I2C_Buf[13];  // Analogue 3 Low Alarm Debounce
        Analog_Config[2]         = I2C_Buf[14];  // Analogue 3 Set-up

        Analog_High_Alarm[3]     = I2C_Buf[15];  // Analogue 4 High alarm
        Analog_Debounce_High[3]  = I2C_Buf[16];  // Analogue 4 High Alarm Debounce
        Analog_Low_Alarm[3]      = I2C_Buf[17];  // Analogue 4 Low Alarm
        Analog_Debounce_Low[3]   = I2C_Buf[18];  // Analogue 4 Low Alarm Debounce
        Analog_Config[3]         = I2C_Buf[19];  // Analogue 4 Set-up

        Analog_High_Alarm[4]     = I2C_Buf[20];  // Analogue 5 High alarm
        Analog_Debounce_High[4]  = I2C_Buf[21];  // Analogue 5 High Alarm Debounce
        Analog_Low_Alarm[4]      = I2C_Buf[22];  // Analogue 5 Low Alarm
        Analog_Debounce_Low[4]   = I2C_Buf[23];  // Analogue 5 Low Alarm Debounce
        Analog_Config[4]         = I2C_Buf[24];  // Analogue 5 Set-up

        Digital_Config1          = I2C_Buf[25];  // Digital inputs Set-up
        Digital_Config2          = I2C_Buf[26];  // Digital inputs Set-up
        Digital_Debounce[0]      = I2C_Buf[27];  // Digital input 1 debounce
        Digital_Debounce[1]      = I2C_Buf[28];  // Digital input 2debounce
        Digital_Debounce[2]      = I2C_Buf[29];  // Digital input 3debounce

        Counter_High_Alarm_Low   = I2C_Buf[30];  // Counter High alarm low byte
        Counter_High_Alarm_High  = I2C_Buf[31];  // Counter High alarm high byte
        Counter_Debounce_High    = I2C_Buf[32];  // Counter High alarm debounce
        Counter_Low_Alarm_Low    = I2C_Buf[33];  // Counter Low alarm low byte
        Counter_Low_Alarm_High   = I2C_Buf[34];  // Counter Low alarm high byte
        Counter_Debounce_Low     = I2C_Buf[35];  // Counter Low alarm debounce
        Counter_Config           = I2C_Buf[36];  // Counter Set-up

        Output1_Pattern          = I2C_Buf[37];  // Bit pattern to be streamed out for output 1
        Output1_Delay            = I2C_Buf[38];  // Bit period for output 1
        Output1_Repeats          = I2C_Buf[39];  // Bit repeats for output 1
        Output2_Pattern          = I2C_Buf[40];  // Bit pattern to be streamed out for output 2
        Output2_Delay            = I2C_Buf[41];  // Bit period for output 2
        Output2_Repeats          = I2C_Buf[42];  // Bit repeats for output 2
        Output3_Pattern          = I2C_Buf[43];  // Bit pattern to be streamed out for output 3
        Output3_Delay            = I2C_Buf[44];  // Bit period for output 3
        Output3_Repeats          = I2C_Buf[45];  // Bit repeats for output 3
        Output4_Pattern          = I2C_Buf[46];  // Bit pattern to be streamed out for output 4
        Output4_Delay            = I2C_Buf[47];  // Bit period for output 4
        Output4_Repeats          = I2C_Buf[48];  // Bit repeats for output 4
        Outputs_State            = I2C_Buf[49];  // States (active/non-active)

        Password_H               = I2C_Buf[50];  // Password high byte
        Password_L               = I2C_Buf[51];  // Password low byte

        RTC_Start_Time_H         = I2C_Buf[52];  // RTC start time in minutes (high byte)
        RTC_Start_Time_L         = I2C_Buf[53];  // RTC start time in minutes (low byte)
        RTC_Stop_Time_H          = I2C_Buf[54];  // RTC stop time in minutes (high byte)
        RTC_Stop_Time_L          = I2C_Buf[55];  // RTC stop time in minutes (low byte)
        RTC_Intervals            = I2C_Buf[56];  // Logging interval in minutes
        RTC_Weekday_Setup        = I2C_Buf[57];  // Weekdays to log/report

        Output1_Pattern_Reload   = Output1_Pattern; // Get outputs in synch...
        Output2_Pattern_Reload   = Output2_Pattern; // Get outputs in synch...
        Output3_Pattern_Reload   = Output3_Pattern; // Get outputs in synch...
        Output4_Pattern_Reload   = Output4_Pattern; // Get outputs in synch...

        Output1_Counter          = 0;            // Get outputs in synch...
        Output2_Counter          = 0;            // Get outputs in synch...
        Output3_Counter          = 0;            // Get outputs in synch...
```

```
Output4_Counter         = 0;                // Get outputs in synch…

TripLog.0 = Analog_Config0.2;               // These bits will force
TripLog.1 = Analog_Config1.2;               // the status of an input
TripLog.2 = Analog_Config2.2;               // to be logged in the
TripLog.3 = Analog_Config3.2;               // event of any other trigger
TripLog.4 = Analog_Config4.2;               //
TripLog.5 = Counter_Trip_Logging;           //
TripLog.6 = Digital_Input_Trip_Logging;     //
TripLog.7 = PowerDwn_Trip_Logging;          //

TripReport.0     = (Analog_Config0.6||Analog_Config0.7);    // These bits will
TripReport.1     = (Analog_Config1.6||Analog_Config1.7);    // force the unit
TripReport.2     = (Analog_Config2.6||Analog_Config2.7);    // to report the
TripReport.3     = (Analog_Config3.6||Analog_Config3.7);    // state/event of
TripReport.4     = (Analog_Config4.6||Analog_Config4.7);    // the input if any
TripReport.5     = (Counter_High_Alarm||Counter_Low_Alarm); // input is triggered
TripReport.7     = PowerDwn_Trip_Reporting;                 //

RTCLog.0         = Analog_Config0.4;        // These bits will
RTCLog.1         = Analog_Config1.4;        // force the unit
RTCLog.2         = Analog_Config2.4;        // to log the particular
RTCLog.3         = Analog_Config3.4;        // input's status/event
RTCLog.4         = Analog_Config4.4;        // every time a RTC
RTCLog.5         = Counter_RTC_Logging;     // log/report event
RTCLog.6         = Digital_Input_RTC_Logging; // occurs
RTCLog.7         = PowerDwn_RTC_Logging;    //

RTCReport.0      = Analog_Config0.3;        // These bits will
RTCReport.1      = Analog_Config1.3;        // force the unit
RTCReport.2      = Analog_Config2.3;        // to report the particular
RTCReport.3      = Analog_Config3.3;        // input's status/event
RTCReport.4      = Analog_Config4.3;        // every time a RTC
RTCReport.5      = Counter_RTC_Reporting;   // log/report event
RTCReport.6      = Digital_Input_RTC_Reporting; // occurs
RTCReport.7      = PowerDwn_RTC_Reporting;  //
}
```

Save IO Parameters (OPM.4)

```
void    Save_IO_Parameters(void)
        {
        I2C_Buf[0]  = Analog_High_Alarm[0];         // Analogue 1 set-up
        I2C_Buf[1]  = Analog_Debounce_High[0];      //
        I2C_Buf[2]  = Analog_Low_Alarm[0];          //
        I2C_Buf[3]  = Analog_Debounce_Low[0];       //
        I2C_Buf[4]  = Analog_Config[0];             //
        I2C_Buf[5]  = Analog_High_Alarm[1];         // Analogue 2 set-up
        I2C_Buf[6]  = Analog_Debounce_High[1];      //
        I2C_Buf[7]  = Analog_Low_Alarm[1];          //
        I2C_Buf[8]  = Analog_Debounce_Low[1];       //
        I2C_Buf[9]  = Analog_Config[1];             //
        I2C_Buf[10] = Analog_High_Alarm[2];         // Analogue 3 set-up
        I2C_Buf[11] = Analog_Debounce_High[2];      //
        I2C_Buf[12] = Analog_Low_Alarm[2];          //
        I2C_Buf[13] = Analog_Debounce_Low[2];       //
        I2C_Buf[14] = Analog_Config[2];             //
        I2C_Buf[15] = Analog_High_Alarm[3];         // Analogue 4 set-up
        I2C_Buf[16] = Analog_Debounce_High[3];      //
        I2C_Buf[17] = Analog_Low_Alarm[3];          //
        I2C_Buf[18] = Analog_Debounce_Low[3];       //
        I2C_Buf[19] = Analog_Config[3];             //
        I2C_Buf[20] = Analog_High_Alarm[4];         // Analogue 5 set-up
        I2C_Buf[21] = Analog_Debounce_High[4];      //
        I2C_Buf[22] = Analog_Low_Alarm[4];          //
        I2C_Buf[23] = Analog_Debounce_Low[4];       //
        I2C_Buf[24] = Analog_Config[4];             //

        I2C_Buf[25] = Digital_Config1;              // Digital inputs and mains power detect set-up
        I2C_Buf[26] = Digital_Config2;              //
        I2C_Buf[27] = Digital_Debounce[0];          //
        I2C_Buf[28] = Digital_Debounce[1];          //
        I2C_Buf[29] = Digital_Debounce[2];          //

        I2C_Buf[30] = Counter_High_Alarm_Low;       //
        I2C_Buf[31] = Counter_High_Alarm_High;      // Event counter / frequency counter set-up
        I2C_Buf[32] = Counter_Debounce_High;        //
        I2C_Buf[33] = Counter_Low_Alarm_Low;        //
        I2C_Buf[34] = Counter_Low_Alarm_High;       //
        I2C_Buf[35] = Counter_Debounce_Low;         //
        I2C_Buf[36] = Counter_Config;               //

        I2C_Buf[37] = Output1_Pattern_Reload;       // Output 1 set-up
        I2C_Buf[38] = Output1_Delay;                //
        I2C_Buf[39] = Output1_Repeats;              //
        I2C_Buf[40] = Output2_Pattern_Reload;       // Output 2 set-up
        I2C_Buf[41] = Output2_Delay;                //
        I2C_Buf[42] = Output2_Repeats;              //
        I2C_Buf[43] = Output3_Pattern_Reload;       // Output 3 set-up
        I2C_Buf[44] = Output3_Delay;                //
        I2C_Buf[45] = Output3_Repeats;              //
        I2C_Buf[46] = Output4_Pattern_Reload;       // Output 4 set-up
        I2C_Buf[47] = Output4_Delay;                //
        I2C_Buf[48] = Output4_Repeats;              //
```

```
I2C_Buf[49] = Outputs_State;                    //

I2C_Buf[50] = Password_H;                       // High byte of password
I2C_Buf[51] = Password_L;                       // Low byte of password

I2C_Buf[52] = RTC_Start_Time_H;                 // High byte of RTC reporting & logging start
I2C_Buf[53] = RTC_Start_Time_L;                 // Low byte of RTC reporting & logging start
I2C_Buf[54] = RTC_Stop_Time_H;                  // High byte of RTC reporting & logging stop
I2C_Buf[55] = RTC_Stop_Time_L;                  // Low byte of RTC reporting & logging stop
I2C_Buf[56] = RTC_Intervals;                    // Reporting & logging interval in minutes
I2C_Buf[57] = RTC_Weekday_Setup;                // Weekdays enabled
                                                //       bit 7 not defined
                                                //       bit 6 Sunday
                                                //       bit 5 Saturday
                                                //       bit 4 Friday
                                                //       bit 3 Thursday
                                                //       bit 2 Wednesday
                                                //       bit 1 Tuesday
                                                //       bit 0 Monday

Output1_Pattern = Output1_Pattern_Reload;       // Make sure the outputs are synchronised
Output2_Pattern = Output2_Pattern_Reload;       //
Output3_Pattern = Output3_Pattern_Reload;       //
Output4_Pattern = Output4_Pattern_Reload;       //

I2C_address = _Setup;                           // Save the set-p
I2C_bytes   = 63;                               // as the new default
EE_Setup_Write_Enable = 1;                      // in the non-volatile memory
Save_String();                                  //
}
```

Set-up RTC (OPM.5)

```
void    Setup_RTC(void)                         //
        {                                       // It is very important that the RTC is never
        I2C_Buf[0] = 0;                         // set up with 0bxx11xxxx in the first register,
        I2C_address = 0;                        // this puts the RTC in a test mode.
        I2C_bytes = 1;                          // The RTC cannot exit from this test mode.
        Save_String();                          // The value 0h00 sets up the RTC for :
        }                                       //       1) No alarm
                                                //       2) 32kHz crystal input
```

Get Time (OPM.6)

This routine is used to read the time from the RTC. The time is then converted into a 16 bit minute-a-day format. This value is used to start or to stop the logging and reporting when compared to the alarm start and stop values stored in non-volatile memory.

The RTC logging and reporting works on the basis of a fixed daily start and stop time, a weekday enable, and an interval. The following table shows how the number of weekly reports may be calculated. If the unit is set to log at these times, the reporting times will depend on the size of the logs. If more information is logged, the unit will report more often. For this table, a start time of 06:00 (360 minutes) and a stop time of 18:00 (1080 minutes) is assumed. A report interval of 60 minutes is assumed with all 7 weekdays enabled. It will result in 13*7 = 91 weekly reports. If, for instance the reporting on Sundays and Saturdays is disabled, the number of reports may be reduced to 13*5 = 65 reports (a 29% saving in communication costs).

	Date	Weekday Enable						
		Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Time		Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Start Time	06:00							
Interval : 60 minutes	07:00							
	08:00							
	09:00							
	10:00							
	11:00							
	12:00							
	13:00							
	14:00							
	15:00							
	16:00							
	17:00							
Stop Time	18:00							

The format that the time and dates are represented in the RTC, is as follows :

		Bits							
Address	**Register name**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
00	Time configuration			oscillator					
01	Tens of seconds								
02	Seconds								
03	Minutes	Minutes (tens)				Minutes (units)			
04	Hours	Hours (tens)				Hours (units)			
05	Days	Year (0-3) (3 : leap)		Days (tens)		Days (units)			
06	Months	Weekday				Months (units)			
07	Timer								
08	RTC Alarm set-up								

```
void    Get_Time(void)
        {
        I2C_address = 0;                                    // Get all the registers from 0 - 7
        I2C_bytes = 7;                                      // because we want to use I2C[0] as
        Load_String();                                      // a scratch register
                                                            //
        I2C_Buf[0]              = BCD2BIN(I2C_Buf[4]);      // Get absolute version of BCD hours...
        RTC_DateTime_M_L = 0;                               // We multiply the
        while (I2C_Buf[0]>0)                                // hours by 60 and
                {                                           // then add the minutes
                RTC_DateTime_M_L += 60;                     // to get an absolute
                I2C_Buf[0]--;                               // day time based
                }                                           // on minutes from
        RTC_DateTime_M_L += BCD2BIN(I2C_Buf[3]);            // 0 - 1440
        RTC_Time         = RTC_DateTime_M_L;                // Store the minute time
                                                            //
        I2C_Buf[5]              &= 0b00111111;              // Mask off year and then
        I2C_Buf[0]              = BCD2BIN(I2C_Buf[5]);      // get the day count
        RLF(I2C_Buf[0]);                                    // and right
        RLF(I2C_Buf[0]);                                    // justify it
        RLF(I2C_Buf[0]);                                    // into
        I2C_Buf[0]              &= 0b11111000;              // second
        RTC_DateTime_M          |= I2C_Buf[0];              // byte
                                                            //
        I2C_Buf[0]              = (I2C_Buf[6] & 0b00011111); // Mask off weekdays
        RTC_DateTime_H          = BCD2BIN(I2C_Buf[0]);      // Get month count
        I2C_Buf[0]              = (I2C_Buf[6] & 0b11100000); // Mask off months
        RTC_DateTime_H          |= I2C_Buf[0];              // Add weekdays
        }
```

The new compressed format that the time and dates are represented in the unit's logging memory, is as follows :

Byte	**Byte 1**								**Byte 2**								**Byte 3**							
Meaning	Weekday				Month of Year				Day of Month								Minute of the day							
Range	0 - 6				1 - 12				1 - [29,30,31]								0 - 1440							
Bits	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

Issue Command (OPM.7)

This command is used to send a command to the GSM engine. It waits for one of four possible return strings before timing out. The time-out period may be changed. If the command times out with no response, it will return with a timeout error. If an unexpected response is received, it will repeat the command for a certain number of times.

If an incoming message response is detected, the data will automatically be stored in the real time clock's scratch memory for later processing.

The command string, the acceptable responses, the time-out period, and the number of repeats before failing are loaded from the non-volatile memory. The first 256 byte in non-volatile memory store 32 blocks of following parameters :

Byte	Parameter
1	Total string length to send to GSM engine
2	Timeout in seconds before retry
3	Total retries
4	Acceptable result 1
5	Acceptable result 2
6	Acceptable result 3
7	Acceptable result 4
8	Not used

The next 1024 bytes store the actual data in 32 byte blocks. The complete string (including the carriage return) is stored here. If a parameter inside the message must change at run time, the value 254 is inserted in the space. Up to three parameters may be changed inside a command. The wanted parameter/s is/are placed in the array Command_Parameters[]. When *Issue_Command* encounters the value 254, it will substitute it with the value in Command_Parameters[].

The commands that are currently used, are :

1	__Dial1	Dial this number
2	__Dial2	Dial this number
3	__Answer	Answer call
4	__Hangup	Hangup call
5	__DTMF_Duration	DTMF tone duration in 100ms
6	__DTMF_Tone	DTMF tone
7	__Operator_Select	Select network operator
8	__Registration_Status	Get registration status
9	__Get_Cell_Environment	Get information on the cell environment
10	__Microphone_Amplify	Set amplification level of microphone
11	__Speaker_Amplify	Set speaker volume
12	__Side_Tone_Control	Control audio side tone
13	__Echo_Off	Echo off
14	__Get_IMEI_Number	Get GSM engine IMEI number
15	__Pin_Issue	Issue PIN number
16	__Select_Message_Service	Select type of SMS service
17	__Select_Message_Format	Select the read/transmit format
18	__New_Message_Indication	Select the new message indication format
19	__Read_SMS_Message	Read the SMS message
20	__Delete_SMS_Message	Delete the SMS message
21	__Send_SMS_Message	Send a message
22	__Select_Service_Centre1	Select Service Centre 1
23	__Select_Service_Centre2	Select Service Centre 1
24	__Select_Service_Centre3	Select Service Centre 1
25	__Save_Settings	Save the current settings
26	__Select_Baudrate	Select the baudrate
27	__Revert_To_Factory_Defaults	Revert to saved defaults
28	__Select_Error_Codes	Select the error code format
29	__Start	String to signal download start

The return value of this subroutine is the response received from the engine. If the command times out, the value is 0. If it received an unexpected response, the return value is 255.

```
unsigned Issue_Command(Command_Number)
        {
        // To Subroutine
        //      Command_Number    {1-32}
        //      Command_Parameter{'0'-'9'}
        //      Response_Timeout {1s-255s}
        //      Retries           {1-255}
        //
        // From Subrountine
        //      Result            {0-255}

        RLF(Command_Number);                                    // Multiply the command number by 8
        RLF(Command_Number);                                    // and mask to get the start address
        RLF(Command_Number);                                    // of the block where the
        Command_Number &= 0b11111000;                           // parameters for the command are
        I2C_address = Command_Number;                           //
                                                                //
        I2C_address_High = 0b10000000;                          // Set bit to force 10 bit addressing
        I2C_bytes = 7;                                          // to the non-volatile memory
        Load_String();                                          // Get the block
                                                                //
        I2C_bytes_temp          = I2C_Buf[0];                   // This is the total length
        Timeout                 = I2C_Buf[1];                   // ...and the time-out
        Retries                 = I2C_Buf[2];                   // ...and the retries
        Acceptable_Results[0]   = I2C_Buf[3];                   // ...and result 1
        Acceptable_Results[1]   = I2C_Buf[4];                   // ...and result 2
        Acceptable_Results[2]   = I2C_Buf[5];                   // ...and result 3
        Acceptable_Results[3]   = I2C_Buf[6];                   // ...and result 4

        Response_Number  = 0;                                   // Clear previous responses
                                                                //
        while ((Retries) && (!Response_Number))                 // While nothing returned and I
                {                                               // still have to try...
                /***** Get the Command from EEPROM *****/       //
                I2C_address = Command_Number;                   // Get the command number
                RLF(I2C_address_Low);                           // and multiply it
                RLF(I2C_address_High);                          // by 4 to get the
                RLF(I2C_address_Low);                           // start of the 32 byte block
                RLF(I2C_address_High);                          // with the string data
                I2C_address_High &= 0b00000011;                 //
                I2C_address_Low  &= 0b11100000;                 //
                I2C_address += 256;                             // It is offset by 256 bytes
                I2C_address_High |= 0b10000000;                 //
                I2C_bytes = I2C_bytes_temp;                     // This was the string length
                                                                // in the parameter block
                if (I2C_bytes>32)                               // Check if it is a valid length
                        {                                       //
                        I2C_bytes = 0;                          // It's not, so don't send anything
                        }                                       //
                                                                //
                Load_String();                                  // Get the string data


                Total_bytes_to_Transmit = I2C_bytes_temp;       // We have to send the whole string
                I2C_Buf[Total_bytes_to_Transmit] = 0;           // Clear byte after the last
                                                                // string character

                I2C_bytes = Total_bytes_to_Transmit;            // Go through the whole
                Total_bytes_to_Transmit = 0;                    // string and look for
                counter1 = 0;                                   // zero character.
                while (counter1 < I2C_bytes)                    // Such characters may
                        {                                       // indicate invalid data
                        if (I2C_Buf[counter1])                  // If a zero character is
                                {                               // found, the string is
                                Total_bytes_to_Transmit++;      // truncated to that point
                                }                               //
                        counter1++;                             //
                        }                                       // Done with zero checking

                counter1 = 0;                                   // Now check if there are
                counter2 = 0;                                   // any run time parameters
                while ((counter1 < I2C_bytes) && (counter2<3))  // that have to be inserted
                        {                                       // into the string.
                        if (I2C_Buf[counter1]==254)             //
                                {                               // Yes - a 254 was found
                                I2C_Buf[counter1] = Command_Parameters[counter2];
                                counter2++;                     // Insert next parameter
                                }                               //
                        counter1++;                             //
                        }                                       // Done with parameter substitution

                /***** Write the Command from Buffer to Wismo *****/
                TX_timeouts = 25;                               // A maximum of 25 seconds
                while ((GISMO_CTS)&&(TX_timeouts)&&(!Ignore_CTS))   // is waited for the CTS line
                        {                                       // to go high before transmission
                        CLRWDT();                               // will start
                        }                                       //

                I2C_Buffer_Index = 0;                           // Set up the transmission buffer
                Transmitting      = 1;                          // Set the busy flag
                TXSTA.TXEN        = 1;                          // Enable transmission
                PIE1.TXIE         = 1;                          // Enable TX interrupts
```

```
        TX_timeouts       = 15;                                 // This string must be
                                                                // sent in 15s or else
        while((Transmitting)&&(TX_timeouts))                    // there is a problem
                {                                               // Tag the watchdog while
                CLRWDT();                                       // we wait…
                }                                               //
                                                                //
        TXSTA.TXEN = 0;                                         // Transmission completed
        PIE1.TXIE = 0;                                          // Disable TX interrupts
        TXSTA.TXEN = 0;                                         // Disable transmission
        PIE1.TXIE = 0;                                          // Make sure about the interrupts

        Retries--;                                              // This was a valid transmission
                                                                // so we expect a response
        Response_Number = _Response_None;                       // Assume no response so far…
        Response_timeouts = Timeout;                            // Get the timeout period
                                                                // Wait for a timeout or a response
        while ((Response_timeouts>0) && (Response_Number == _Response_None));
                                                                //
        if (Response_timeouts==0)                               // If no response at all, there
                {                                               // must be something wrong,
                return(_Response_TimeOut);                      // so return with a timeout error
                }                                               //
                                                                //
        if (Response_Number == _Response_Message)               // If we are reading a message
                {                                               // we must store it in the RTC memory.
                while (MessageReadingStatus != _Starting);      // Wait for the beginning of the data…
                MessageReadingStatus = _NotReading;             // Wait for a <CR><LF> that will signify
                while ((RX_Hash_1) || (RX_Hash_2));             // the start of the data
                I2C_address = 16;                               // Start saving at address 16 in the RTC
                Open_EEPROM_4_Write();                          // Open it for writing…
                counter1 = 0;                                   //
                RX_Oldest = 100;                                //
                while (RX_Oldest>13)                            // Carry on until the next <CR><LF>

                        {                                       // is received.
                        GotChar = 0;                            //
                        while (!GotChar);                       // Wait for a character
                        MessageTemp1 = RX_Old - 48;             // Get the binary value
                        if (MessageTemp1>9)                     // of the HEX
                                {                               // Get rid of the A-F
                                MessageTemp1 = MessageTemp1 - 7;
                                }                               //
                                                                // Ok - the low nibble was
                        GotChar = 0;                            // received.
                        while (!GotChar);                       // Wait for high nibble
                        RX_Oldest = RX_Old;                     // Got it
                        MessageTemp2 = RX_Old - 48;             // Get the binary value
                        if (MessageTemp2>9)                     // of the HEX
                                {                               // Get rid of the A-F
                                MessageTemp2 = MessageTemp2 - 7;
                                }                               //
                        SWAPF(MessageTemp2);                    // Get the nibble in high
                                                                // position
                        I2C_data = (MessageTemp1|MessageTemp2); // Combine the nibbles
                        Write_byte();                           // Save it
                        }                                       // Now go get the next byte…
                                                                //
                I2C_Stop();                                     // position  - stop the bus
                                                                //
                return(_Response_Message);                      // Return value shows new data
                                                                // received in RTC scratch memory
                }                                               //
                                                                //
        if (  (Response_Number == Acceptable_Results[0])        // Check if the response matches
            ||(Response_Number == Acceptable_Results[1])        // expected results
            ||(Response_Number == Acceptable_Results[2])        //
            ||(Response_Number == Acceptable_Results[3]))       //
                {                                               // Yes - it matches
                Delay(20);                                      // Wait a few millisecond to
                return(Response_Number);                        // make it half duplex
                }                                               // Return the Response number
        }                                                       //
                                                                //
Delay(20);                                                      // No - the expected response was
return(_Response_FatalError);                                   // not received - quit trying.
}                                                               //
```

Resolve Message Header (OPM.8)

This command is used to validate a command. It checks the password, as well as the integrity of the message. It also extracts other data, like the message sequence number and the return path. The return value is the start of the first command address in the RTC scratch memory.

Byte	Parameter
1	Total string length to send to GSM engine
2	Timeout in seconds before retry
3	Total retries
4	Acceptable result 1
5	Acceptable result 2
6	Acceptable result 3
7	Acceptable result 4
8	Not used

```
unsigned Resolve_Message_Header(void)
        {                                                       //
        I2C_address    = 16;                                    // Load the whole RAM
        I2C_bytes      = 64;                                    // buffer with the header
        Load_String();                                          // and the first data
                                                                //
        RM3 = I2C_Buf[1];                                       // Calculate the sender
        SWAPF(I2C_Buf[1]);                                      // telephone number length
        if Testbit(I2C_Buf[1],0) I2C_Buf[1]++;                  // in case we have to return
        I2C_Buf[1] = I2C_Buf[1]/2;                              // the message to this number
        I2C_Buf[1] += 3;                                        //
                                                                //
        Server_Sequence_Number    = I2C_Buf[12+I2C_Buf[1]];     // Get Sequence number
        SWAPF(Server_Sequence_Number);                          // Fix it
                                                                //
        counter1 = I2C_Buf[14+I2C_Buf[1]];                      // Get password high byte
        counter2 = I2C_Buf[15+I2C_Buf[1]];                      // Get password low byte
                                                                //
        if ((counter1!=Password_H)||(counter2!=Password_L))     // Check if it matches...
                {                                               // No - return
                return(255);                                    // to main loop
                }                                               // with error
                                                                // Password was ok
        Response_Targets = I2C_Buf[16+I2C_Buf[1]];              // Check return path
        Response_Targets &= 0x03;                               //
                                                                // If it must be returned to the
        if (Response_Targets==0x00)                             // sender, we must save the number
                {                                               // Yes - save the number for later
                RM4 = I2C_Buf[1];                               // Get number length
                for (counter1 = 0; counter1<RM4; counter1++)    // Move it to another buffer
                        {                                       // so we can work on the
                        RX_Buffer[counter1] = I2C_Buf[counter1];// format of the data before
                        }                                       // saving it
                                                                //
                counter2 = 0;                                   //
                counter1 = 1;                                   //
                RX_Buffer[1] = RM3;                             // Get the original length back
                while (counter1<10)                             // All numbers must be shorter than
                        {                                       // 20 characters (10 BCD)
                        I2C_Buf[0] = RX_Buffer[counter1] & 0x0F; // Split up the BCD
                        I2C_Buf[1] = RX_Buffer[counter1] & 0xF0; // into two bytes
                        SWAPF(I2C_Buf[1]);                      //
                        I2C_Buf[0] += 48;                       // Make it into ASCII
                        I2C_Buf[1] += 48;                       //
                        if (I2C_Buf[0]>57) I2C_Buf[0] += 7;     // and fix the HEX numbers
                        if (I2C_Buf[1]>57) I2C_Buf[1] += 7;     //
                                                                //
                        I2C_address = _SMS_HeaderReturn+counter2; // Save the two bytes in the
                        I2C_bytes   = 2;                        // memory location allocated for
                        EE_Setup_Write_Enable = 1;              // the return path number
                        Save_String();                          //
                        counter1++;                             //
                        counter2+=2;                            //
                        }                                       //
                I2C_Buf[1] = RM4;                               // Restore number length
                Current_Service_Centre = __Select_Service_Centre1; //
                }                                               //
                                                                //
        Command_address = I2C_Buf[1]+17;                        // This is where the command
                                                                // starts, so return the index
        return(Command_address);                                // to the main routine
        }
```

Get Next Command (OPM.9)

```
unsigned Get_Next_Command(void)
        {

        I2C_address       = Command_address+16;                    // Get the length byte to see how
        I2C_bytes         = 1;                                     // many bytes must be loaded from
        Load_String();                                             // the memory
        Command_length    = I2C_Buf[0];                            // This is the total command length
                                                                   //
        if (Command_length>64)                                     // Check for integrity…
                {                                                  // No command will be longer than
                return(_Response_FatalError);                      // 64 bytes. If it is, there is a
                }                                                  // problem - ignore the message
                                                                   //
        I2C_address       = Command_address+16;                    // Now load the message from the
        I2C_bytes         = Command_length+1;                      // specified address with the length
        Load_String();                                             // just determined
                                                                   //
        RecordCommand     = I2C_Buf[2];                            // Get the type of command
                                                                   //
        counter1 = I2C_Buf[Command_length-1];                      // Get the terminator
                                                                   //
        if ((counter1!=EndofLastRecord)&&(counter1!=EndofRecord))  // Check the terminator
                {                                                  // An integrity check - if the terminator
                return(_Response_FatalError);                      // is not present, there is
                }                                                  // an error in the message - ignore it
                                                                   //
                                                                   //


        Command_address += Command_length;                        // Later we will look for the start of the
                                                                   // next command in the memory
        RecordLength = 0;                                          // Begin with 0 length
                                                                   //


                                                                   // Now we examine the individual
                                                                   // commands to determine how long the
                                                                   // response will be. This is necessary
                                                                   // because if the response will be too
long
                                                                   // for the available log space, the log
        switch(RecordCommand)                                      // must be sent first.
                {                                                  //
                case 2 :        {                                  // Memory Read
                                I2C_address_High = I2C_Buf[3];     // This is easy - the response
                                I2C_address_Low  = I2C_Buf[4];     // will be as long as the
                                I2C_bytes        = I2C_Buf[5];     // requested number of bytes plus
                                                                   // a header of 10 bytes
                                I2C_address_temp = I2C_address;    // Save the address and the number of
                        bytes
                                I2C_bytes_temp   = I2C_Buf[5];     // so we can use it to save the data later

                                if (I2C_bytes>64)                  //
                                        {                          // If the number sounds crazy, just
                                        I2C_bytes = 64;            // chop it back to 64
                                        }                          //
                                RecordLength = I2C_bytes + 10;     //
                                break;                             //
                                }                                  //
                case 4 :        {                                  //
                                NOP();                             //
                                RecordLength = 0;                  // The following bytes will be returned :
                                if (RM1.7) RecordLength++;         // Analogue 1 value (1 byte)
                                if (RM1.6) RecordLength += 5;      // Analogue 1 set-up (5 bytes)
                                                                   // High Alarm
                                                                   // High Alarm Debounce
                                                                   // Low Alarm
                                                                   // Low Alarm Debounce
                                                                   // Analogue set-up
                                                                   //       bit 7 Analogue 1 High Alarm Enable

                                                                   //       bit 6 Analogue 1 Low Alarm Enable

                                                                   //       bit 5 Analogue 1 Average Enable
                                                                   //       bit 4 Analogue 1 Trip logging
                                                                   //       bit 3 Analogue 1 RTC Logging
                                                                   //       bit 2 Analogue 1 RTC Reporting
                                if (RM1.5) RecordLength++;         // Analogue 2 value (1 byte)
                                if (RM1.4) RecordLength += 5;      // Analogue 2 set-up (5 bytes)
                                                                   // High Alarm
                                                                   // High Alarm Debounce
                                                                   // Low Alarm
                                                                   // Low Alarm Debounce
                                                                   // Analogue set-up
                                                                   //       bit 7 Analogue 2 High Alarm Enable

                                                                   //       bit 6 Analogue 2 Low Alarm Enable

                                                                   //       bit 5 Analogue 2 Average Enable
                                                                   //       bit 4 Analogue 2 Trip logging
                                                                   //       bit 3 Analogue 2 RTC Logging
                                                                   //       bit 2 Analogue 2 RTC Reporting
                                if (RM1.3) RecordLength++;         // Analogue 3 value (1 byte)
                                if (RM1.2) RecordLength += 5       // Analogue 3 set-up (5 bytes)
                                                                   // High Alarm
                                                                   // High Alarm Debounce
                                                                   // Low Alarm
```

```
                                        // Low Alarm Debounce
                                        // Analogue set-up
                                        //      bit 7 Analogue 3 High Alarm Enable

                                        //      bit 6 Analogue 3 Low Alarm Enable

                                        //      bit 5 Analogue 3 Average Enable
                                        //      bit 4 Analogue 3 Trip logging
                                        //      bit 3 Analogue 3 RTC Logging
                                        //      bit 2 Analogue 3 RTC Reporting
if (RM1.1) RecordLength++;              // Analogue 4 value (1 byte)
if (RM1.0) RecordLength += 5;           // Analogue 4 set-up (5 bytes)
                                        // High Alarm
                                        // High Alarm Debounce
                                        // Low Alarm
                                        // Low Alarm Debounce
                                        // Analogue set-up
                                        //      bit 7 Analogue 4 High Alarm Enable

                                        //      bit 6 Analogue 4 Low Alarm Enable

                                        //      bit 5 Analogue 4 Average Enable
                                        //      bit 4 Analogue 4 Trip logging
                                        //      bit 3 Analogue 4 RTC Logging
                                        //      bit 2 Analogue 4 RTC Reporting
if (RM2.7) RecordLength++;              // Analogue 5 value (1 byte)
if (RM2.6) RecordLength += 5;           // Analogue 5 set-up (5 bytes)
                                        // High Alarm
                                        // High Alarm Debounce
                                        // Low Alarm
                                        // Low Alarm Debounce
                                        // Analogue set-up
                                        //      bit 7 Analogue 5 High Alarm Enable

                                        //      bit 6 Analogue 5 Low Alarm Enable

                                        //      bit 5 Analogue 5 Average Enable
                                        //      bit 4 Analogue 5 Trip logging
                                        //      bit 3 Analogue 5 RTC Logging
                                        //      bit 2 Analogue 5 RTC Reporting

if (RM2.5) RecordLength++;              // Digital Input states & events
                                        //      bit 7 Power status
                                        //      bit 6 Digital input 3 state
                                        //      bit 5 Digital input 2 state
                                        //      bit 4 Digital input 1 state
                                        //      bit 3 Power Down event
                                        //      bit 2 Digital input 3 event
                                        //      bit 1 Digital input 2 event
                                        //      bit 0 Digital input 1 event

if (RM2.4) RecordLength++;              // Input Polarity/RTC R&L/Trigger L
                                        //      bit 7 Digital input 3 polarity
                                        //      bit 6 Digital input 2 polarity
                                        //      bit 5 Digital input 1 polarity
                                        //      bit 4 Digital input 3 trigger log
en
                                        //      bit 3 Digital inputs RTC report en
                                        //      bit 2 Digital inputs RTC log en

if (RM2.3) RecordLength++;              // Power down RTC R&L Trigger R&L
                                        //      bit 7 Power down trigger report en
                                        //      bit 6 Power down trigger log en
                                        //      bit 5 Power down RTC report en
                                        //      bit 4 Power down RTC log en
                                        //      bit 3 Digital input 1 trig report
en                                      //      bit 2 Digital input 2 trig report
en                                      //      bit 1 Digital input 3 trig report
en

if (RM2.2) RecordLength++;              // Digital Debounce 1
if (RM2.1) RecordLength++;              // Digital Debounce 2
if (RM2.0) RecordLength++;              // Digital Debounce 3

if (RM3.7) RecordLength += 2;           // Counter/Frequency Value (high/low)
if (RM3.6) RecordLength += 7;           // Counter/Frequency Set-up
                                        // High alarm high byte
                                        // High alarm low
                                        // High alarm debounce
                                        // Low alarm high byte
                                        // Low alarm low byte
                                        // Low alarm debounce
                                        // Set-up :
                                        //      bit 7 High alarm enable
                                        //      bit 6 Low alarm enable
                                        //      bit 5 Mode (1 = frequency)
                                        //      bit 4 Trigger Log enable
                                        //      bit 3 RTC report enable
                                        //      bit 2 RTC log enable
if (RM3.5) RecordLength++;              // Outputs : States
if (RM3.4) RecordLength += 3;           // Output 1 : Pattern/Delay/Repeats
if (RM3.3) RecordLength += 3;           // Output 2 : Pattern/Delay/Repeats
if (RM3.2) RecordLength += 3;           // Output 3 : Pattern/Delay/Repeats
if (RM3.1) RecordLength += 3;           // Output 4 : Pattern/Delay/Repeats

if (RM3.0) RecordLength++;              // Termination character
break;                                  // End of this reply
```

```
                                }
                        }

        if (I2C_Buf[Command_length-1]==EndofRecord)             // Check the type of termination character
                {                                               // If it is end of message, then
                return(255);                                    // this was the last command
                }                                               // if it was end of command, then
        else return(Command_address);                           // this was not the last command
        }                                                       //
```

Send the Log (OPM.10)

This command is used to send all the logged messages via the specified path. It waits for confirmation before resetting all the triggers.

```
void    Send_The_Log(void)                                      //
        {                                                       //
        I2C_Buf[0]    = 0x54;                                   // The first two characters of a GTel
message
        I2C_Buf[1]    = 0x45;                                   // will always be "TE" for identification.
        I2C_Buf[2]    = Protocol_Version;                       // It is 0 for this phase of GTel
        I2C_Buf[3]    = Server_Sequence_Number;                 // Sequence number is issued by the server
        I2C_Buf[4]    = Transmit_Retries;                       // This number will increase on every TX
fail
        I2C_address   = _SMS_Message_Data;                      // This is where the log begins
        I2C_bytes     = 5;                                      // Update this header
        Save_String();                                          //

        I2C_address_temp = _SMS_Message_Data + LogPosition + 10;// Fill up all the data space with 0h20
        while (I2C_address_temp < (_SMS_Message_Data+160))      // From the start….
                {                                               //
                I2C_Buf[0]    = 32;                             //
                I2C_address   = I2C_address_temp;               //
                I2C_bytes     = 1;                              //
                Save_String();                                  //
                I2C_address_temp++;                             //
                }                                               // …to the finish
                                                                //

        /* First two bytes of number*/                          //
        I2C_Buf[0]    = 0x10;                                   // The message will always
        I2C_Buf[1]    = 0x00;                                   // start with "0100"
        I2C_address   = _SMS_Message_TargetNumber;              //
        I2C_bytes     = 2;                                      //
        Save_String();                                          // So save it
                                                                //

        switch (Response_Targets)                               // The required return path is now used
                {                                               // A path is selected…
                case _Return_To_Sender   : {I2C_address_temp = _SMS_HeaderReturn;break;}
                case _Return_To_Number_1 : {I2C_address_temp = _SMS_Header1;break;}
                case _Return_To_Number_2 : {I2C_address_temp = _SMS_Header2;break;}
                case _Return_To_Number_3 : {I2C_address_temp = _SMS_Header3;break;}
                }

        I2C_address = I2C_address_temp;                         // The selected number will be at
        I2C_bytes   = 20;                                       // this address
        Load_String();                                          // Get the number

                                                                //
        I2C_bytes_temp = I2C_Buf[1]-48;                         // Get the length of the number
        if (I2C_bytes_temp>15){I2C_bytes_temp -= 7;}            // Get rid of 'F' hex character if present
        if Testbit(I2C_bytes_temp,0) {I2C_bytes_temp++;}        // Make sure it is even ('F' makes it
uneven)                                                         //
        I2C_bytes_temp += 4;                                    // To get the number of characters, we
have
        RRF(I2C_bytes_temp);                                    // to multiply by two.
        I2C_bytes_temp &= 0b01111111;                           // Make sure it isn't wrong.
                                                                //
        Header_Length  = I2C_bytes_temp + 5;                    // The 5 first characters must be added
                                                                //
        counter1=0;                                             // Now the SCII characters of
        counter2=0;                                             // the telephone number must
        while (counter1<20)                                     // be swapped into the PDU format
                {                                               // The whole number is swapped…
                Convert1 = I2C_Buf[counter1];                   //
                Convert2 = I2C_Buf[counter1+1];                 //
                I2C_Buf[counter1] = ASCII_2_HEX(void);          //
                I2C_Buf[counter2] = SWAPF(I2C_Buf[counter1]);   //
                counter1 += 2;                                  //
                counter2++;                                     //
                }                                               //
        I2C_address = _SMS_Message_TargetNumber+2;              // …and then saved in the required
position
        I2C_bytes   = I2C_bytes_temp;                           // inside the message
        Save_String();                                          //

        /* Three bytes after number */                          // These three bytes will always
        I2C_Buf[0]    = 0x00;                                   // be present in the GTel messages.
        I2C_Buf[1]    = 0x6F;                                   // It wedges in between the number
        I2C_Buf[2]    = 0xC8;                                   // and the data
        I2C_address   = (_SMS_Message_TargetNumber+2)+I2C_bytes_temp;//
        I2C_bytes     = 3;                                      //
        Save_String();                                          // …so save it
                                                                //
```

```
        Message_Length = Header_Length + 140;                   // The total message length is the total
of
                                                                // the data (140) and the header length
        BIN2BCD(Message_Length);                                // Get the BCD of this total message
length
        Command_Parameters[0] = BCD_Temp1;                      // for the transmission command
        Command_Parameters[1] = BCD_Temp2;                      //
        Command_Parameters[2] = BCD_Temp3;                      //
        Result = Issue_Command(__Send_SMS_Message);             // Send the command
        Wismo_State_Timer = 1;                                  // Wait for 1 second before sending
        while(Wismo_State_Timer);                               // the data portion of the message.
                                                                //
                                                                // Now the header and data can be sent
        for (counter1=0;counter1<Message_Length;counter1++)     // The data and the header are stored
                {                                               // in different parts of the
                                                                // non-volatile memory because of
                                                                // the varying length of the header
                if (counter1<Header_Length)                     //
                        {                                       // This data comes from the header
                        I2C_address_temp = _SMS_Message_TargetNumber;// so get it from the address
                        I2C_address = I2C_address_temp+counter1;// where the header is stored
                                                                //
                        }                                       //
                else                                            //
                        {                                       // This data comes from the logging
                        I2C_address_temp =  _SMS_Message_Data;  // section
                        I2C_address = I2C_address_temp+counter1-Header_Length;
                        }                                       //
                                                                //
                                                                //
                I2C_bytes = 1;                                  // Load the byte from whereever
                Load_String();                                  //
                                                                //
                Convert_Temp1 = I2C_Buf[0]&0b11110000;          //
                Convert_Temp2 = I2C_Buf[0]&0b00001111;          //
                                                                //
                SWAPF(Convert_Temp1);                           // Now the byte must be changed
                                                                // into two hex characters
                Convert_Temp1 += 48;                            // make it ASCII…
                Convert_Temp2 += 48;                            //
                                                                //
                if (Convert_Temp1>'9')                          //
                        {                                       //
                        Convert_Temp1 += 7;                     // Check if > 9 : chance to A,B,C,D,E,F
                        }                                       //
                                                                //
                if (Convert_Temp2>'9')                          //
                        {                                       //
                        Convert_Temp2 += 7;                     // Check if > 9  : chance to A,B,C,D,E,F
                        }                                       //
                                                                //
                I2C_Buf[0] = Convert_Temp2;                     // Get the two characters ready
                I2C_Buf[1] = Convert_Temp1;                     // for transmission.
                I2C_Buf[2] = 13;                                // A carriage return in case of an overrun

                Total_bytes_to_Transmit = 2;                    // Only the first two characters must be
sent
                                                                //
                if (counter1==(Message_Length-1))               // On the last byte, we also want to send
                        {                                       // the carriage return (<CR>) to tell
                        Total_bytes_to_Transmit++;              // the GSM engine to start the
transmission
                        }                                       //
                                                                //
                I2C_Buffer_Index  = 0;                          // Start from index zero
                Transmitting      = 1;                          // Set busy flag
                TXSTA.TXEN        = 1;                          // Enable TX
                PIE1.TXIE         = 1;                          // Enable TX interrupt
                TX_timeouts       = 15;                         // Set up a timeout for
                                                                // in case the previous
                while((Transmitting)&&(TX_timeouts))            // transmission hasn't completed yet.
                        {                                       //
                        CLRWDT();                               // Wait for it to finish
                        }                                       //
                                                                //
                TXSTA.TXEN        = 0;                          // Disable TX
                PIE1.TXIE         = 0;                          // Disable TX interrupt
                TXSTA.TXEN        = 0;                          // Make sure TX is disabled…
                PIE1.TXIE         = 0;                          // Make sure about that interrupt…
                                                                //
                }                                               // Now go and get the next byte for
                                                                // transmission…

                                                                // The message has been sent to the GSM
        Response_Number = 0;                                    // engine. The unit must now wait to see
        TX_timeouts = Timeout;                                  // if the transmission was a success
        while((Response_Number==0)&&(TX_timeouts>0));           // or if it failed.
                                                                // We wait for 'Timeout' seconds.

        if (Response_Number == _Response_SMSC_Is_Sick)          // Aaaarggh! The SMSC is sick!
                {                                               // Plan B : Switch to the next SMSC
                Current_Service_Centre++;                       // If the message was to
                if (Response_Targets>_Return_To_Sender)         // return to a standard
                        {                                       // number/SMCS pair
                        Response_Targets++;                     // try the next pair
                        if (Response_Targets>_Return_To_Number_3)// If we have already tried all 3,
                                {                               // we start again, but
```

```
                                Response_Targets = _Return_To_Number_1; // we have to wait
                                GTEL_State = _GTEL_State_Wait_for_a_minute; // for a minute first
                                MessageDelayInSeconds = 60;         // or else we can crash the SMSC
                                }                                   // with multiple units
                        }                                           //
                else                                                //
                        {                                           //
                        Current_Service_Centre++;                   // The message was supposed to go back
                        if (Current_Service_Centre > __Select_Service_Centre3) // the sender, so we can only
                                {                                   // change the SMSC, not the target number
                                Current_Service_Centre = __Select_Service_Centre1; // If we have tried all 3,
                                GTEL_State = _GTEL_State_Wait_for_a_minute; // we start again with SMSC 1
after
                                MessageDelayInSeconds = 60;         // waiting for 1 minute
                                }                                   //
                        }                                           //
                if (Transmit_Retries<255) Transmit_Retries++;       // Show the failed transmission in the TX
                }                                                   // retry count. Don't roll over to 0
                                                                    //
        else if (Response_Number == _Response_SMSOk)                // If the transmission was successful,
                {                                                   //
                Transmit_Retries = 0;                               // we can reset the retry counter
                Current_Service_Centre = __Select_Service_Centre1;  // we can reset the SMSC to 1
                Response_Targets = _Return_To_Number_1;             // and we can reset the target number to 1
                                                                    //
                ReportTheInput          = 0;                        // Reset the
                Trip1                   = 0;                        // flags that
                Trip2                   &= 0b11110000;              // caused the
                RTC_Alarm_Event  = 0;                               // report in the first place
                LogPosition = 0;                                    // Reset the start of the log
                LogFree = 135;                                      // All the logging memory is free
                }                                                   //
        else                                                        //
                {                                                   // The TX failed because of a radio
                if (Transmit_Retries<255) Transmit_Retries++;       // problem, not a sick SMSC, so try again
                }                                                   // straight away.
                                                                    //
```

ASCII to HEX (OPM.11)

```
unsigned ASCII_2_HEX(void)
        {                                       // The ASCII hex number is represented
        Convert1 = Convert1 - 48;               // by two characters : Convert1 & Convert12
        if (Convert1>15) {Convert1 -= 7;}       // We convert these two
        Convert2 = Convert2 - 48;               // characters into an
        if (Convert2>15) {Convert2 -= 7;}       // absolute number.
        SWAPF(Convert1);                        // This number is represented as a hex number
        return(Convert1|Convert2)               // with the high and low nibbles representing
        }                                       // the high and low hex characters.
```

Binary to BCD (OPM.12)

```
void BIN2BCD(BCDBIN_In)
        {
        BCD_Temp1  = 48;                        // An 8 bit absolute number is taken
        BCD_Temp2  = 48;                        // and broken down into a three character
        BCD_Temp3  = 48;                        // BCD string. The input character is BCDBIN_In
                                                //
        while (BCDBIN_In>99)                    // Calculate the hundreds digit - start with '0'
                {                               //
                BCD_Temp1++;                    //
                BCDBIN_In-=100;                 //
                }                               //
        while (BCDBIN_In>9)                     // Calculate the tens digit - start with '0'
                {                               //
                BCD_Temp2++;                    //
                BCDBIN_In-=10;                  //
                }                               //
        while (BCDBIN_In>0)                     // Calculate the units digit - start with '0'
                {                               //
                BCD_Temp3++;                    //
                BCDBIN_In-=1;                   //
                }                               //
        }                                       //
```

BCD to Binary (OPM.13)

```
unsigned BCD2BIN(BCDBIN_In)                     // A two - digit BCD number is converted
        {                                       // into an 8 bit number
        BCD_Temp1  = (BCDBIN_In&0b00001111);    // Do the low nibble first
        BCDBIN_In    &= 0b11110000;             //
        SWAPF(BCDBIN_In);                       // Now the high nibble
        BCD_Temp2 = BCD2BIN_Lookup[BCDBIN_In];  // Get the tens value of the nibble from a lookup table
        BCD_Temp1 += BCD_Temp2;                 // Add the two values
        return(BCD_Temp1);                      // Return it
        }
```

10.6 Customer Software Concept Design

The customer software will be a modified version of the prototype exercise software. A few extra features on the exercising software for error trapping purposes could be removed from the customer software. This will save the time needed for such a development. In principle, the following facilities should be present :

1) A GSM device capable of sending, or receiving SMS messages. In this case, a GSM modem will be used;
2) The ability to set up the program to work with any serial communications port present, and to be able to set this port up;
3) The ability to build any of the outgoing message type with any of the parameters for that message; and
4) The ability to display the information of any incoming message in a proper manner.

The customer software will be a modified version of the prototype exercise software. A few extra features on the exercising software for error trapping purposes could be removed from out on the customer software.

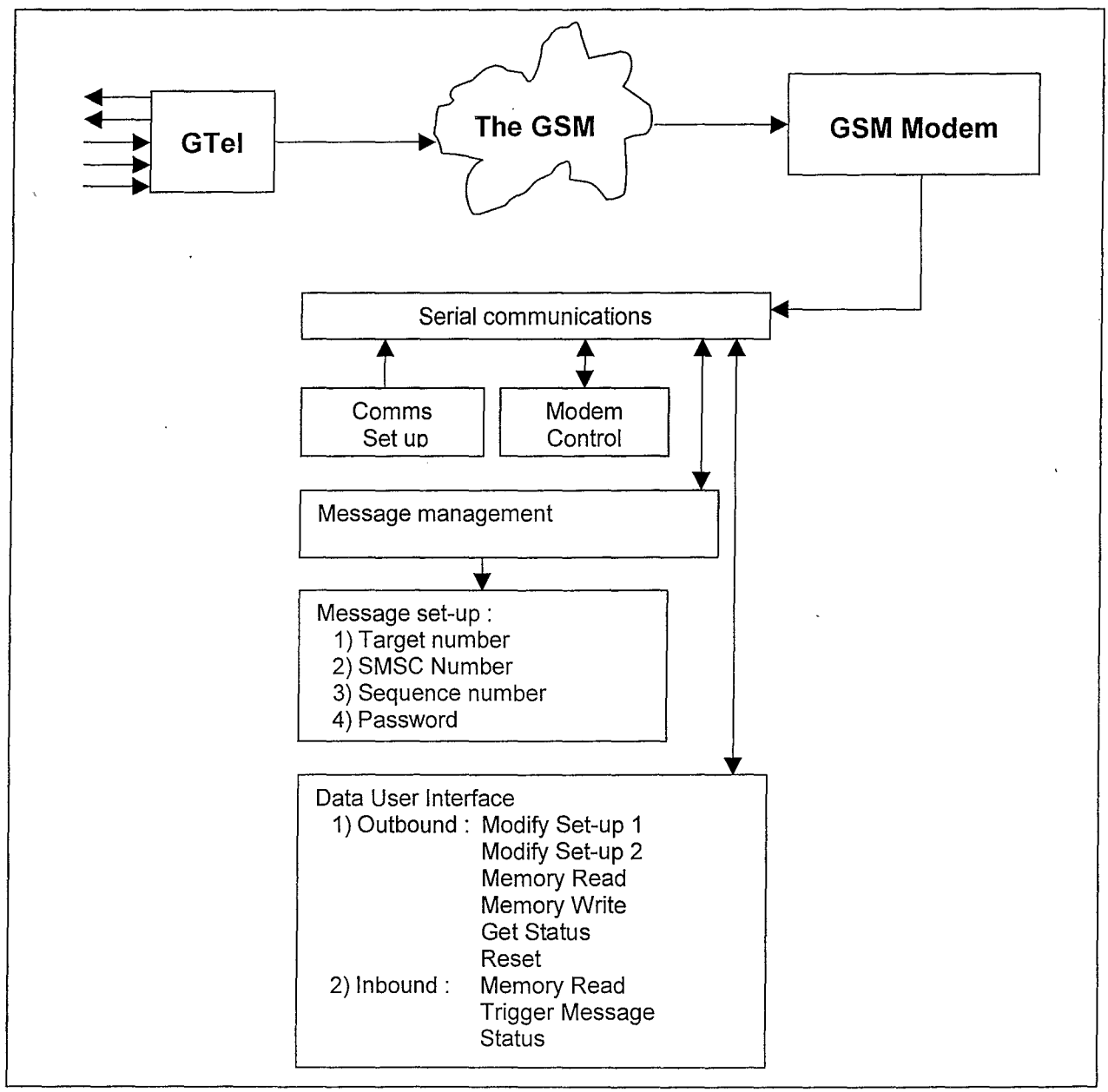


Figure 11. The customer software concept.

10.7 In-Circuit Emulator Set-up

The in-circuit emulator for the PIC16C77 microprocessor has an inverted socket to connect to a socket in a circuit board. Because a surface mount version of the PIC will be used, this socket will not be able to connect directly to the PCB. Almost all the pins of the PIC was brought out to a 40 way header connector on the PCB. An adapter was made to connect the emulator socket to the PCB connector.

Because the wires were relatively long (>10cm), the emulator could not use a crystal mounted on the PCB. Therefore, the on-board oscillator was enabled. Local power was also selected over target power. The Host PC is also used for the downloading of data to the target. An extra PC (Monitoring PC) is used to monitor the data on the breakout jig.

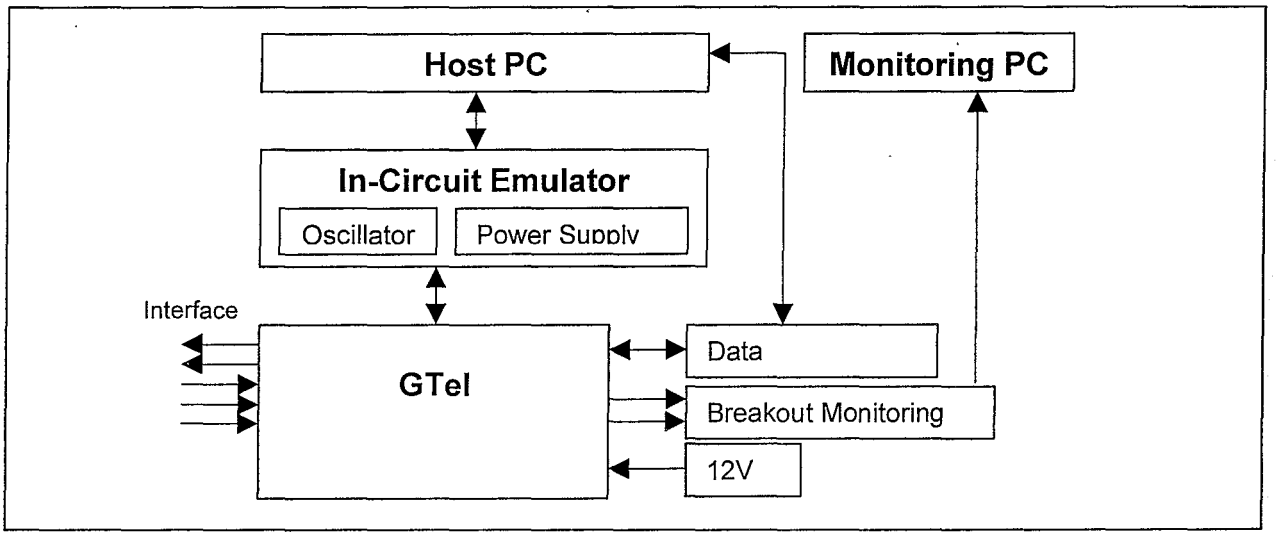


Figure 12. The emulatorder delopment environment.

10.8 Firmware Module Integration

The firmware program flow will now be analysed. The structure will be analysed from a high level, then a lower level.

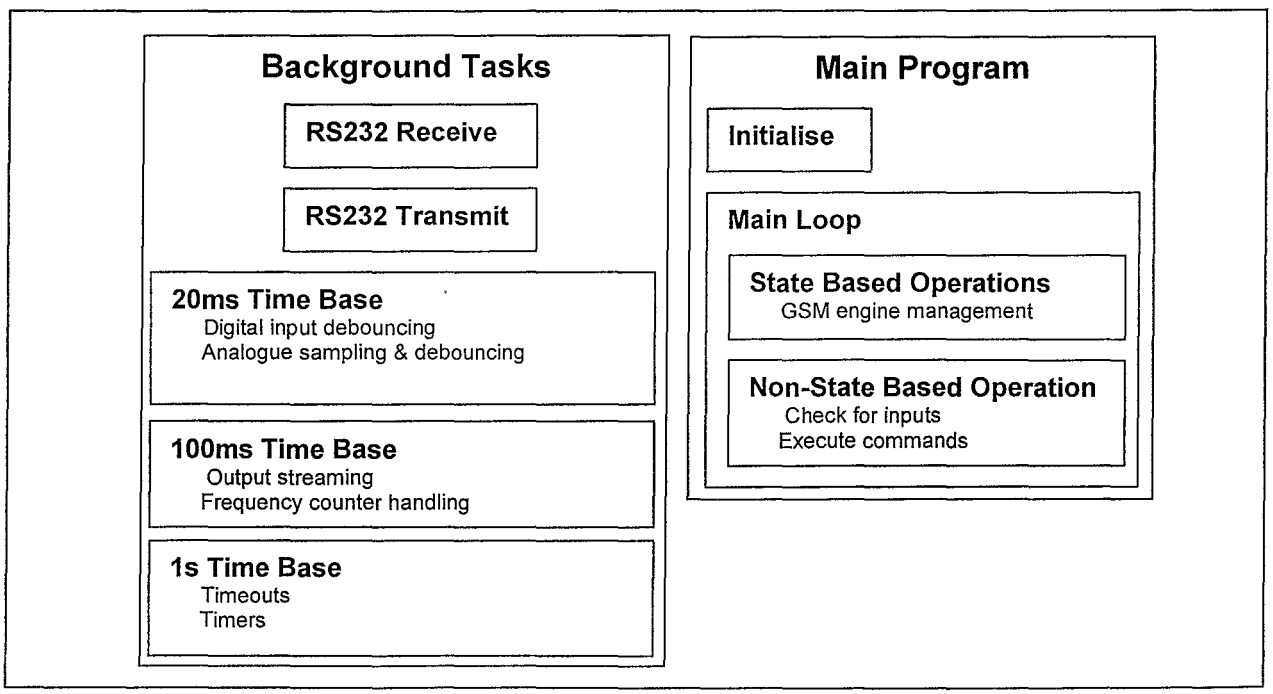


Figure 13. The design strategy of the firmware.

The background tasks run independently from the main loop, so the particular flow is not important, as long as the time spent in the interrupt service routine does not exceed the minimum time base (20ms), or double-buffered serial communications character transmission time (2* 10/9600 = 2ms). This constraint is easily met.

State Based Operations

The state bases code is executed at least every 5s during normal operation. These cycles are called "housekeeping cycles" because it ensure that all the basic set-ups are in order. This is necessary because any loss of set-up on the GSM engine could lead to unpredictable results. The state based program structure makes future upgrades and maintenance very easy. The states that the program can be in, are :

1) *_GTEL_State_Wait_for_a_minute*
   This is a delay state used to delay the next transmission in the case of all the paths failing on transmission. The next state from here is *_GTEL_State_SetErrorCodes*

2) *_GTEL_State_CheckIfOK*
   This state tests the GSM engine to see if it will respond to an "AT" command. If it does respond, the program won't reset the engine (hardware reset), but continue with the set-up.

3) *_GTEL_State_ResetWismo*
   This state will reset the GSM engine, and wait 1s for it to recover

4) *_GTEL_State_SetErrorCodes*
   This state sets the format of the error codes returned.

5) *_GTEL_State_EchoOff*
   This state disables the GSM engine's echo of characters sent to it.

6) *_GTEL_State_IssuePin*
   This state issues the PIN number to the SIM card. It waits for the unit to be logged onto the network.

7) *_GTEL_State_NetworkStatus*
   This state checks the status of the unit on the network. It will return whether the unit is logged, roaming, not logged, or if no networks could be found.

8) *_GTEL_State_MessageFormat*
   This state puts the unit into PDU mode for all messages received or transmitted.

9) *_GTEL_State_ServiceCenter*
   This state will set the GSM engine to use the selected service centre for all transmitted messages.

The flow of the states with entry and exit point, is :

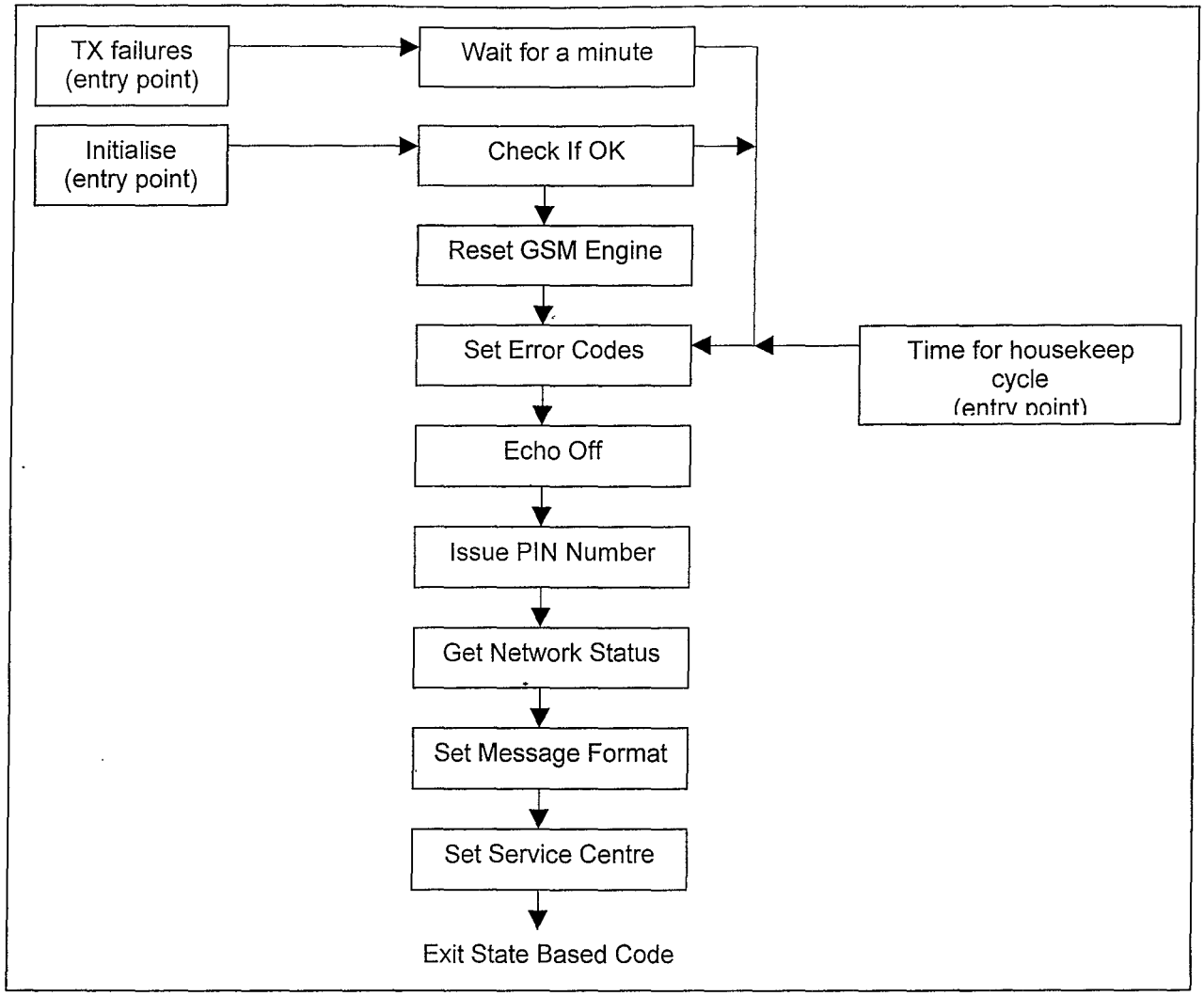


Figure 14. The state based tasks in the main loop of the firmware.

Non - State Based Operations

The non-state based code executes whenever the state is zero. This is code that is not program-flow sensitive, but needs constant servicing. The functions of this code are :

1) Update Time/Date Variables
2) Check for Triggers
3) Read Messages

1) Update Time/Date Variables :

The time is read and processed (Get_Time() : OPM.6). From this procedure, we get the time in terms of day minutes, as well as weekday. This information is used to decided whether the real time reporting or logging should be active or not. In the next section, the time is compared to the stored start and stop times to determine whether an alarm is to be signalled.

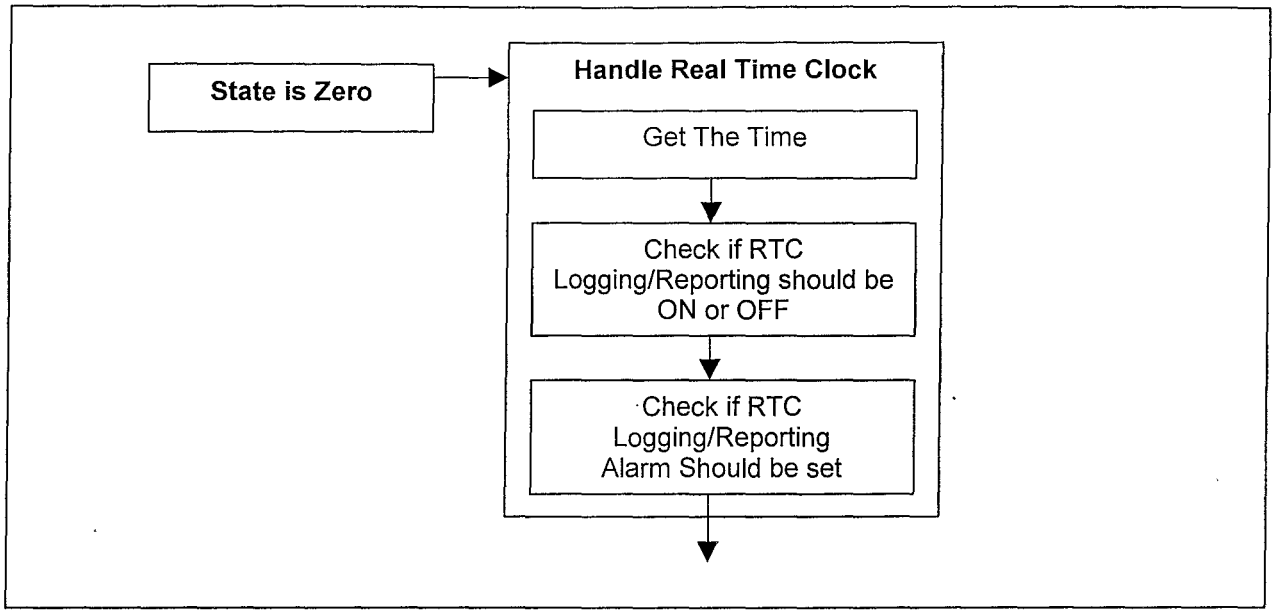


Figure 15. The non - state based (low priority) tasks in the main loop of the firmware.

```
Get_Time();                                                     // Get the Real time Clock Time
                                                                // and weekday.

if ((RTC_Time>=RTC_Start_Time)&&(RTC_Time<=RTC_Stop_Time))      // Check to see if the time is
        {                                                       // within the start and stop times
        RTC_Timing_On = 1;                                      // Yes - Logging/Reporting is active
        }                                                       //
else                                                            //
        {                                                       //
        RTC_Timing_On = 0;                                      // No - Logging/Reporting is not active
        }                                                       //

RTC_New_Minute_State = Testbit(I2C_Buf[0],1);                   // Check for next minute. Unit uses the
RTC
                                                                // bit flag to keep time of
logging/reporting
if (    (RTC_New_Minute_State) &&                               // Check if it is a new minute…
        (!RTC_Old_Minute_State) &&                              // …and not an old one.
        (RTC_Timing_On) &&                                      // Are we within the logging times?
        (RTC_Weekday(I2C_Buf[6]) & RTC_Weekday_Setup))          // Are we in the right day of week?
        {                                                       // Yes
        RTC_Minute_Flag_Counter++;                              // We want to count 10 minutes…
        if (RTC_Minute_Flag_Counter>9)                          // …so we check if we've got ten yet
                {                                               // Yes!
                RTC_Minute_Flag_Counter = 0;                    // Start the 10 minute counter from 0
again
                RTC_10_Minute_Counter++;                        // The interval counter is increased…
                if (RTC_10_Minute_Counter>=RTC_Intervals)       // …and compared with the stored interval
                        {                                       //
                        RTC_10_Minute_Counter = 0;              // Yes - we have waited long enough.
                        if (RTCReport || RTCLog)                // Check if anything will be reported at
all.
                                {                               //
                                RTC_Alarm_Event = 1;            // Yes - send the alarm flag
                                        }                       // to the trigger routine.
                        }                                       //
                }                                               //
        }                                                       //
RTC_Old_Minute_State  = RTC_New_Minute_State;                   // Make it edge-triggered
```

2) Check for Triggers :

This section of code checks whether there are any triggers to report or pending logs. The analogue inputs, event counter/frequency counter, digital inputs and mains power failure detect are checked for events. Then their reporting/logging set-ups are compared to determine the message length.

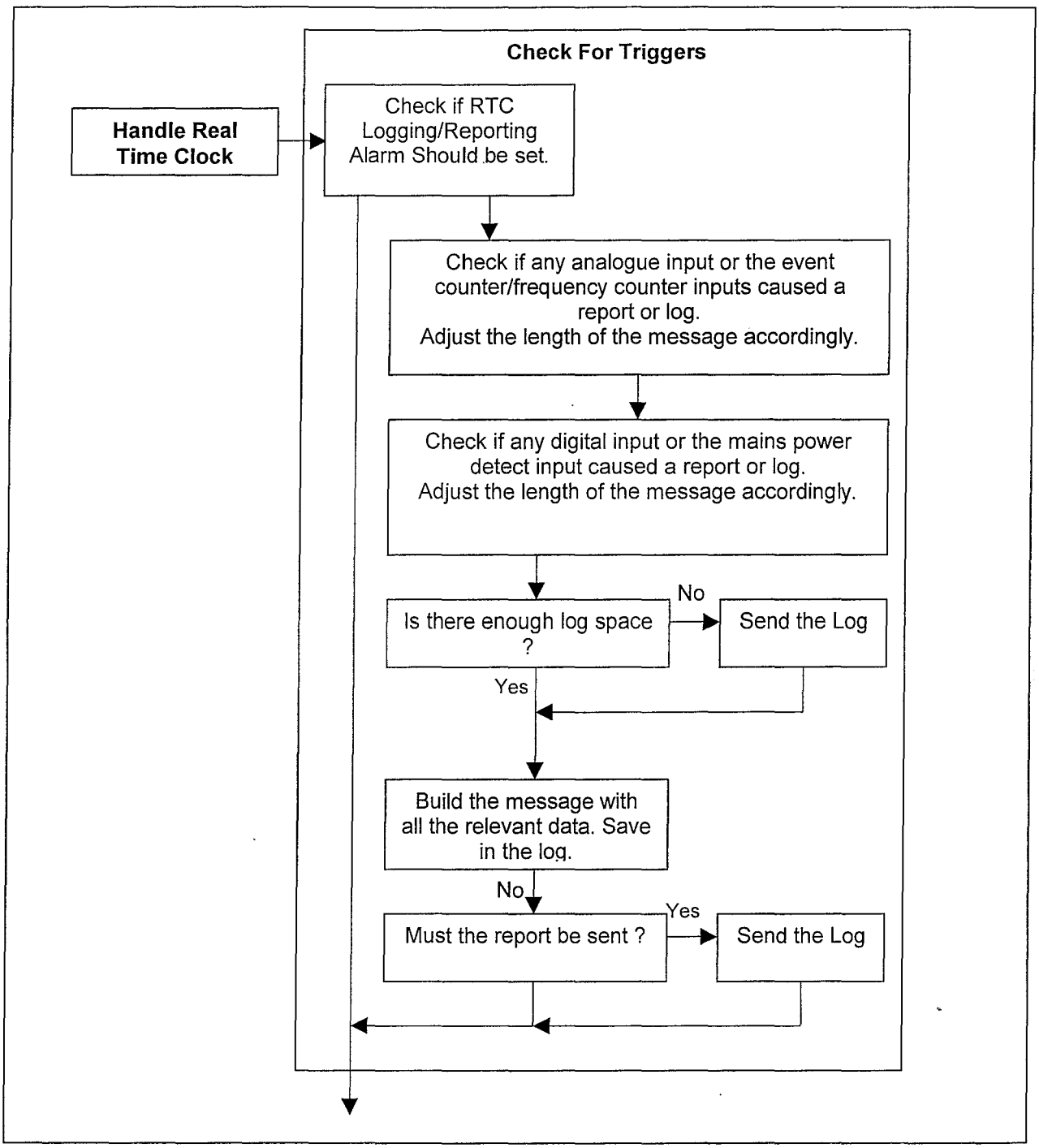


Figure 16. The event flags are checked for triggers.

```
if ((Trip1&0b00111111)||(Trip2&0b00001111)||(RTC_Alarm_Event))      // We are looking for an analogue,
digital,
        {                                                          // mains power, or frequency input
trigger.
                                                                   // A RTC alarm event will also be handled

        RecordLength = 12;                                         // Check to see how long message will be
        RM1 = (Trip1 | TripLog);                                   // (12 with header so far)
        if (RM1.0) {RecordLength++;}                               // Any of the analogues will
        if (RM1.1) {RecordLength++;}                               // contribute 1 byte (the value)
        if (RM1.2) {RecordLength++;}                               // to the length.
        if (RM1.3) {RecordLength++;}                               //
        if (RM1.4) {RecordLength++;}                               //
        if (RM1.5) {RecordLength+=2;}                              // The frequency input contributes 2 bytes
                                                                   // (16 bit value)
        RM1 = (TripReport || RTCReport);                           // Check if the analogues or frequency
inputs
        if (Trip1 && RM1) ReportTheInput = 1;                      // have to report..

        RM1   = 0;                                                 // The Reporting Mask for the digital
inputs
        RM1.0 = Digital_Input_1_Trip_Reporting;                    // Reporting enable for input 1
        RM1.1 = Digital_Input_2_Trip_Reporting;                    // Reporting enable for input 2
        RM1.2 = Digital_Input_3_Trip_Reporting;                    // Reporting enable for input 3
        RM1.3 = PowerDwn_Trip_Reporting;                           // Reporting enable for power fail
        if (Digital_Input_RTC_Reporting && RTC_Alarm_Event) RM1 |= 0b0000111; // RTC reporting for inputs
        if (PowerDwn_RTC_Reporting && RTC_Alarm_Event)      RM1.3 = 1;        // RTC reporting for power fail
        RM1 &= Trip2;                                              // Compare inputs to report mask
        if (RM1) ReportTheInput = 1;                               // Check if inputs have to report..

        RM2   = 0;                                                 // The Logging Mask for the digital inputs
        if (Digital_Input_Trip_Logging)    RM2 |= 0b00000111;      // Trigger Logging for inputs
        if (Digital_Input_RTC_Logging)     RM2 |= 0b00000111;      // RTC Logging for inputs
        if (PowerDwn_Trip_Logging)         RM2.3 = 1;              // Power down Trip logging
        if (PowerDwn_RTC_Logging)          RM2.3 = 1;              // Power down RTC logging
        RM2 &= Trip2;                                              // Check if inputs have to log

        if (RM1 || RM2) {RecordLength++;}                          // Log/Report Inputs
                                                                   //

        if (RecordLength > LogFree)                                // Check to see if there is enough space
                {                                                  //
                Send_The_Log();                                    // No there isn't - send log first
                }                                                  //

        RecordLength = 12;                                         // Log the Trigger Message
                                                                   //
        Get_Time();                                                // Get new time
                                                                   //
        I2C_Buf[1] = 0x08;                                         // Device 1 trigger
        I2C_Buf[2] = 0x09;                                         // Command 9 - unsolicited report
        I2C_Buf[3] = RTC_DateTime_H;                               // RTC stuff
        I2C_Buf[4] = RTC_DateTime_M;                               // RTC stuff
        I2C_Buf[5] = RTC_DateTime_L;                               // RTC stuff
                                                                   //
        I2C_Buf[6] = Trip1;                                        // The analogue/frequency trigger byte
        I2C_Buf[7] = Trip2;                                        // The digital/power fail trigger byte
                                                                   //
        I2C_Buf[8] = TripReport;                                   // What must be reported
        I2C_Buf[9] = TripLog;                                      // What must be logged
                                                                   //
        I2C_Buf[10] = Digital_Config1;                             // Get the polarity & reporting set-up
        if (Counter_Mode) I2C_Buf[10] |= 0b00000001;               // Includes counter mode in message
        I2C_Buf[11] = Digital_Config2;                             // Get the power-down handling & some
                                                                   //
        RM1   = TripReport & Trip1 & 0b00111111;                   // Check if analogues or counter triggered
        RM2   = 0;                                                 // and must be reported
        if (Trip2.0 && Digital_Input_1_Trip_Reporting) RM2.0 = 1;   // The digital inputs are not as easy…
        if (Trip2.1 && Digital_Input_2_Trip_Reporting) RM2.1 = 1;   // …so we check them one-for-one
        if (Trip2.2 && Digital_Input_3_Trip_Reporting) RM2.2 = 1;   // for triggering with the reporting
        if (Trip2.3 && PowerDwn_Trip_Reporting)        RM2.3 = 1;   // enabled
        if (RM1|RM2) ReportTheInput = 1;                           // Yes - at least SOMETHING must be
reported

        RM3   = TripLog & Trip1 & 0b00111111;                      // Check which analogues or frequency must
                                                                   // show that they have been logged
        RM4   = 0;                                                 // Check if the inputs must
        if (Digital_Config2 & 0b00011100) RM4 =  0b00000111;       // show they have been logged
        if (Digital_Config1 & 0b01110000) RM4 |= 0b00001000;       //
        RM4  &= Trip2;                                             //

        Trip1 |= TripLog;                                                    // Must any of the
        if (Trip1.0) {I2C_Buf[RecordLength] = Analogue_1; RecordLength++;}   // analogues or frequency input
        if (Trip1.1) {I2C_Buf[RecordLength] = Analogue_2; RecordLength++;}   // values be included?
        if (Trip1.2) {I2C_Buf[RecordLength] = Analogue_3; RecordLength++;}   //
        if (Trip1.3) {I2C_Buf[RecordLength] = Analogue_4; RecordLength++;}   //
        if (Trip1.4) {I2C_Buf[RecordLength] = Analogue_5; RecordLength++;}   //
        if (Trip1.5) {I2C_Buf[RecordLength] = Timer1_Value_H; RecordLength++;//
                      I2C_Buf[RecordLength] = Timer1_Value_L; RecordLength++;}//

        I2C_Buf[RecordLength] = 0x32;                              // The message terminator
        RecordLength++;                                            // must also be added
        I2C_Buf[0] = RecordLength;                                 // The message length is added

        I2C_address = _SMS_Message_Data+LogPosition + 5;           // We start saving the data right after
the
        I2C_bytes   = RecordLength;                                // header.
```

```
Save_String();                                          //
LogPosition += RecordLength;                            // The new logging address has moved…
LogFree -= RecordLength;                                // There is less logging space left…
                                                        //

if (ReportTheInput)                                     // Check to see if log must be sent
        {                                               //
        Send_The_Log();                                 // Yep - so send it.
        }                                               //
}                                                       //
```

3) Read Messages :

The time is read and processed (Get_Time() : OPM.6). From this procedure, we get the time in terms of day minutes, as well as weekday. This information is used to decided whether the real time reporting or logging should be active or not. In the next section, the time is compared to the stored start and stop times to determine whether an alarm is to be signalled.

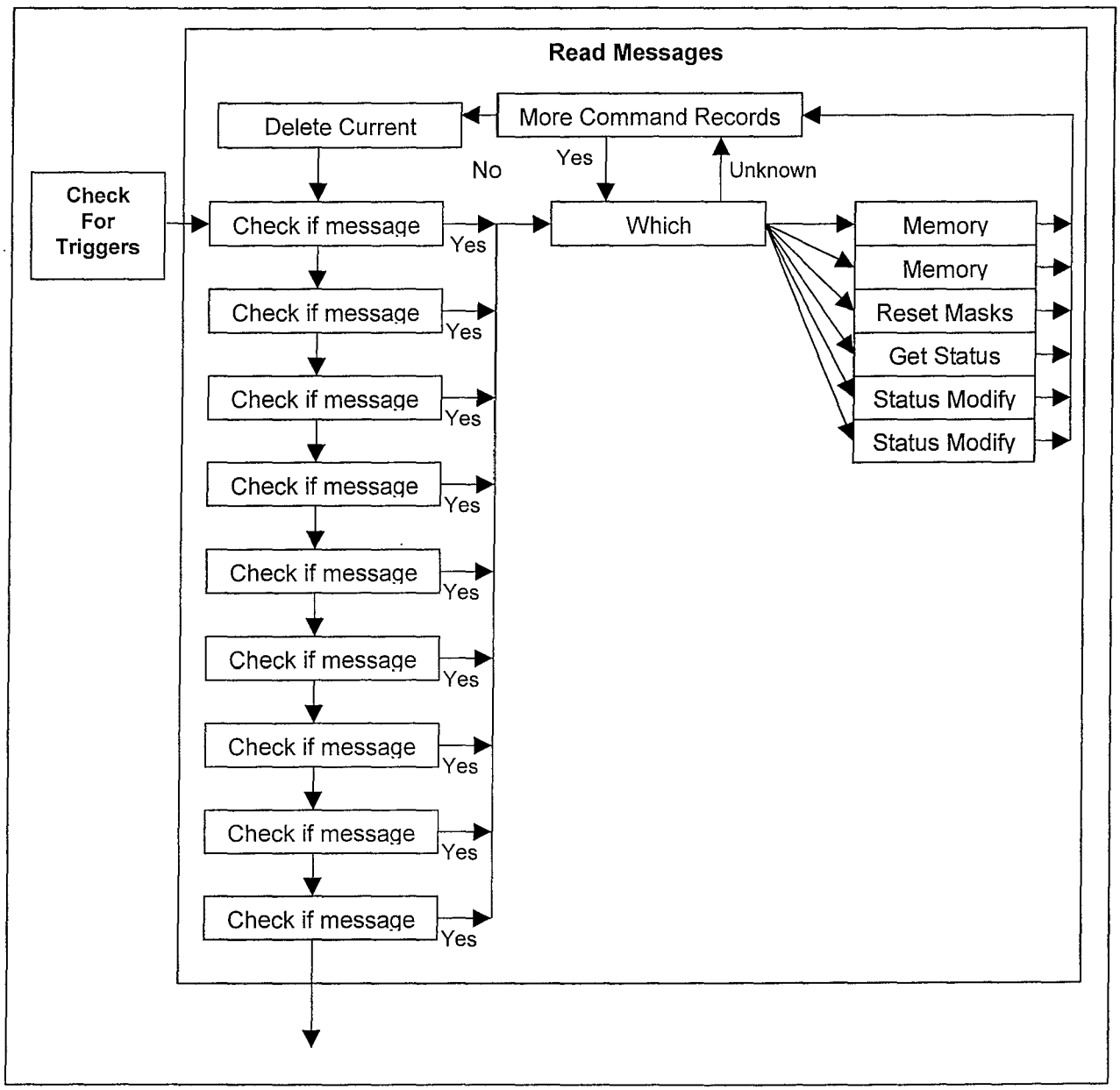


Figure 17. Reading the messages on the SIM card.

```
Result = _Response_None;                                    // Assume no response
while ((Result != _Response_Message)&&(MessageNumber<='9'))  // Read all messages from 1 to 9
        {                                                   //
        Command_Parameters[0] = MessageNumber;              // Set up message number
        Result = Issue_Command(__Read_SMS_Message);         // Send it and wait for result
        if (Result != _Response_Message) MessageNumber++;   // If empty, go to next message
        }

                                                            // The program comes here if there was
                                                            // a non-zero message read
Result = Resolve_Message_Header();                          // Get Header Information and validate

while (Result<253)                                          // While there are command records in
        {                                                   // the message to process...

        Result = Get_Next_Command();                        // Get the next command record...
        if ((RecordLength > LogFree) &&                     //
            ((RecordCommand==4)||(RecordCommand==1)))       // Check if there is enough space left
                {                                           // to store the response.
                Send_The_Log();                             // If there isn't, report the log first
                }                                           // and then start a new log.
                                                            //
        switch(RecordCommand)                               // The command record will now be
processed.
                {
/* Memory Write */              case 1 :                    // **** It was a memory write.
                                {                           // The data should be in the RTC memory
                                Load_String();              // Get the data from RTC memory
                                I2C_address = I2C_address_temp; // Restore the address and number
                                I2C_bytes   = I2C_bytes_temp;   // of bytes to save
                                Save_String();              // ...and save it in the non-volatile memory
                                break;                      //
                                }


/* Memory Read */case 2 :      {                           // **** It was a Memory Read
                                I2C_address_High = I2C_Buf[3];  // Get the high address byte
                                I2C_address_Low  = I2C_Buf[4];  // Get the low address byte
                                I2C_bytes        = I2C_Buf[5];  // Get the number of bytes
                                I2C_bytes_temp   = I2C_bytes;   // Save the address and the
                                I2C_address_temp = I2C_address  // number of bytes for later

                                Get_Time();                 // Fresh time

                                I2C_Buf[0] = I2C_bytes_temp+10 ; // Command record length
                                I2C_Buf[1] = 0x08;          // Device 1
                                I2C_Buf[2] = 0x02;          // This was a memory write
                                I2C_Buf[3] = RTC_DateTime_H;    // RTC goodies
                                I2C_Buf[4] = RTC_DateTime_M;    // RTC goodies
                                I2C_Buf[5] = RTC_DateTime_L;    // RTC goodies
                                I2C_Buf[6] = I2C_address_High;  // Set the high address byte
                                I2C_Buf[7] = I2C_address_Low;   // Set the low address byte
                                I2C_Buf[8] = I2C_bytes;         // Set the number of bytes

                                                            //
                                I2C_address = _SMS_Message_Data+LogPosition + 5;
                                I2C_bytes   = 9;            // Save the first section of record
                                Save_String();              // It was just the command header so far
                                                            // but the data will follow now

                                I2C_address = I2C_address_temp  // Get the I2C data from the
                                I2C_bytes = I2C_bytes_temp;     // RTC memory
                                Load_String();              //

                                I2C_address = _SMS_Message_Data+LogPosition + 5 + 9;
                                I2C_bytes = I2C_bytes_temp;     // Save it in the non-volatile memory
                                Save_String();              //

                                I2C_Buf[0] = 0x32;          // Terminator character
                                I2C_bytes = 1;              // The address will be correct
                                Save_String();              // because it increases on every write

                                RecordLength = I2C_bytes_temp+10; // This was the total length
                                LogPosition += RecordLength;    // Adjust the logging position
                                LogFree -= RecordLength;        // Adjust the free space
                                break;                      //
                                }
```

```
/* Reset Mask */ case 3 :          {                                   // **** It was a reset
                                   RM1 = I2C_Buf[3];                   // Get Mask 1
                                   RM2 = I2C_Buf[4];                   // Get Mask 1 XOR'ed
                                   RM3 = I2C_Buf[5];                   // Get Mask 2
                                   RM4 = I2C_Buf[6];                   // Get Mask 2 XOR'ed
                                   if (RM1.0& !RM2.0) Trip1 = 0;       // Reset Analogue & frequency counter
events
                                   if (RM1.1& !RM2.1) Trip2 = 0;       // Reset input events
                                   if (RM1.2& !RM2.2) GTEL_State = _GTEL_State_ResetWismo; // Reset Wismo
                                   if ((RM3==0xAA)&(RM4==0x55)) while(1); // Wait for a watch dog timeout
                                   break;
                                   }
/* Status Request */      case 4 :
                                   {                                   // **** It was a status request

                                   RM1 = I2C_Buf[3];                   // Get request mask 1
                                   RM2 = I2C_Buf[4];                   // Get request mask 2
                                   RM3 = I2C_Buf[5];                   // Get request mask 3

                                   Get_Time();                         // Fresh Time

                                   I2C_Buf[0] = RecordLength+9;        // Total length of Message
                                   I2C_Buf[1] = 0x08;                  // Device 1
                                   I2C_Buf[2] = 0x04;                  // Status request
                                   I2C_Buf[3] = RTC_DateTime_H;        // RTC goodies
                                   I2C_Buf[4] = RTC_DateTime_M;        // RTC goodies
                                   I2C_Buf[5] = RTC_DateTime_L;        // RTC goodies
                                                                       //
                                   I2C_Buf[6] = RM1;                   // The masks have to
                                   I2C_Buf[7] = RM2;                   // be returned so the base
                                   I2C_Buf[8] = RM3;                   // could use the data
                                                                       //
                                   I2C_address = _SMS_Message_Data+LogPosition + 5;
                                   I2C_bytes   = 9;                    // Save the header so long

                                   Save_String();                      //

                                   LogPosition += 9;                   // Adjust the new log position
                                   LogFree -= 9;                       // Adjust the free space
                                   RecordLength = 0;                   // Total Record length will be saved
                                                                       // in first byte later on.
                                                                       // Analogue Inputs
                                   if (RM1.7) {I2C_Buf[RecordLength] = Analog_Value[0]; RecordLength++;}
                                   if (RM1.6) {I2C_Buf[RecordLength] = Analog_High_Alarm[0]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_High[0]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Low_Alarm[0]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_Low[0]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Config[0]; RecordLength++;}
                                   if (RM1.5) {I2C_Buf[RecordLength] = Analog_Value[1]; RecordLength++;}
                                   if (RM1.4) {I2C_Buf[RecordLength] = Analog_High_Alarm[1]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_High[1]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Low_Alarm[1]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_Low[1]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Config[1]; RecordLength++;}
                                   if (RM1.3) {I2C_Buf[RecordLength] = Analog_Value[2]; RecordLength++;}
                                   if (RM1.2) {I2C_Buf[RecordLength] = Analog_High_Alarm[2]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_High[2]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Low_Alarm[2]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_Low[2]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Config[2]; RecordLength++;}
                                   if (RM1.1) {I2C_Buf[RecordLength] = Analog_Value[3]; RecordLength++;}
                                   if (RM1.0) {I2C_Buf[RecordLength] = Analog_High_Alarm[3]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_High[3]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Low_Alarm[3]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_Low[3]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Config[3]; RecordLength++;}
                                   if (RM2.7) {I2C_Buf[RecordLength] = Analog_Value[4]; RecordLength++;}
                                   if (RM2.6) {I2C_Buf[RecordLength] = Analog_High_Alarm[4]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_High[4]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Low_Alarm[4]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Debounce_Low[4]; RecordLength++;
                                               I2C_Buf[RecordLength] = Analog_Config[4]; RecordLength++;}

                                                                       // Digital Inputs
                                   if (RM2.5) {I2C_Buf[RecordLength] = Trip2; RecordLength++;}
                                   if (RM2.4) {I2C_Buf[RecordLength] = Digital_Config1; RecordLength++;}
                                   if (RM2.3) {I2C_Buf[RecordLength] = Digital_Config2; RecordLength++;}
                                   if (RM2.2) {I2C_Buf[RecordLength] = Digital_Debounce[0]; RecordLength++;}
                                   if (RM2.1) {I2C_Buf[RecordLength] = Digital_Debounce[1]; RecordLength++;}
                                   if (RM2.0) {I2C_Buf[RecordLength] = Digital_Debounce[2]; RecordLength++;}
```

```
                                        // Frequency/Counter Input
                if (RM3.7) {I2C_Buf[RecordLength] = Timer1_Value_L; RecordLength++;
                           I2C_Buf[RecordLength] = Timer1_Value_H; RecordLength++;}
                if (RM3.6) {I2C_Buf[RecordLength] = Counter_High_Alarm_High; RecordLength++;
                           I2C_Buf[RecordLength] = Counter_High_Alarm_Low; RecordLength++;
                           I2C_Buf[RecordLength] = Counter_Debounce_High; RecordLength++;
                           I2C_Buf[RecordLength] = Counter_Low_Alarm_High; RecordLength++;
                           I2C_Buf[RecordLength] = Counter_Low_Alarm_Low; RecordLength++;
                           I2C_Buf[RecordLength] = Counter_Debounce_Low; RecordLength++;
                           I2C_Buf[RecordLength] = Counter_Config; RecordLength++;}

                                        // Outputs
                if (RM3.5) {I2C_Buf[RecordLength] = Outputs_State; RecordLength++;}
                if (RM3.4) {I2C_Buf[RecordLength] = Output1_Pattern_Reload; RecordLength++;
                           I2C_Buf[RecordLength] = Output1_Delay; RecordLength++;
                           I2C_Buf[RecordLength] = Output1_Repeats; RecordLength++;}
                if (RM3.3) {I2C_Buf[RecordLength] = Output2_Pattern_Reload; RecordLength++;
                           I2C_Buf[RecordLength] = Output2_Delay; RecordLength++;
                           I2C_Buf[RecordLength] = Output2_Repeats; RecordLength++;}
                if (RM3.2) {I2C_Buf[RecordLength] = Output3_Pattern_Reload; RecordLength++;
                           I2C_Buf[RecordLength] = Output3_Delay; RecordLength++;
                           I2C_Buf[RecordLength] = Output3_Repeats; RecordLength++;}
                if (RM3.1) {I2C_Buf[RecordLength] = Output4_Pattern_Reload; RecordLength++;
                           I2C_Buf[RecordLength] = Output4_Delay; RecordLength++;
                           I2C_Buf[RecordLength] = Output4_Repeats; RecordLength++;}

                I2C_Buf[RecordLength] = 0x32; RecordLength++;       // Termination character

                I2C_address = _SMS_Data_Start+LogPosition;          // Save data at new
address
                I2C_bytes   = RecordLength;                         // Number of data bytes
                Save_String();                                      // Save it
                LogPosition += RecordLength;                        // Next log position
                LogFree -= RecordLength;                            // Adjust free space
                                                                    //
                break;
                }

/* Status Modify */     case 5 :
                {                                                   // **** Status Modify
                RM1 = I2C_Buf[3];                                   // Get first mask…
                RecordLength = 4;                                   // The length start at 4
                if (RM1.7) Analog_High_Alarm[0]    = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.6) Analog_Debounce_High[0] = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.5) Analog_Low_Alarm[0]     = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.4) Analog_Debounce_Low[0]  = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.3) {                                // The limits & debounce changed
                           RM2 = I2C_Buf[RecordLength];     // Now change configuration
                           RM3 = I2C_Buf[RecordLength+1]; //
                           RM1 = Analog_Config[0];          //
                                                            //
                           if (RM2.7) RM1.2 = 1;            // Analogue High Alarm Enable
                           if (RM2.6) RM1.6 = 1;            // Analogue Low Alarm Enable
                           if (RM2.5) RM1.5 = 1;            // Analogue Average
                           if (RM2.4) RM1.4 = 1;            // Analogue Trip logging
                           if (RM2.3) RM1.3 = 1;            // Analogue RTC Logging
                           if (RM2.2) RM1.2 = 1;            // Analogue RTC Reporting
                           if (RM3.7) RM1.2 = 0;            // Analogue High Alarm Disable
                           if (RM3.6) RM1.6 = 0;            // Analogue Low Alarm Disable
                           if (RM3.5) RM1.5 = 0;            // Analogue No Average
                           if (RM3.4) RM1.4 = 0;            // No Analogue Trip logging
                           if (RM3.3) RM1.3 = 0;            // No Analogue RTC Logging
                           if (RM3.2) RM1.2 = 0;            // No Analogue RTC Reporting
                           Analog_Config[0] = RM1;          // Set configuretion
                           }                                //
                RecordLength += 2;                          //
                RM1 = I2C_Buf[RecordLength]; RecordLength ++; // Get next analogue mask
                if (RM1.7) Analog_High_Alarm[1]    = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.6) Analog_Debounce_High[1] = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.5) Analog_Low_Alarm[1]     = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.4) Analog_Debounce_Low[1]  = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.3) {                                // The limits & debounce changed
                           RM2 = I2C_Buf[RecordLength];     // Now change configuration
                           RM3 = I2C_Buf[RecordLength+1]; //
                           RM1 = Analog_Config[1];          //
                                                            //
                           if (RM2.7) RM1.2 = 1;            // Analogue High Alarm Enable
                           if (RM2.6) RM1.6 = 1;            // Analogue Low Alarm Enable
                           if (RM2.5) RM1.5 = 1;            // Analogue Average
```

```
            if (RM2.4) RM1.4 = 1;           // Analogue Trip logging
            if (RM2.3) RM1.3 = 1;           // Analogue RTC Logging
            if (RM2.2) RM1.2 = 1;           // Analogue RTC Reporting
            if (RM3.7) RM1.2 = 0;           // Analogue High Alarm Disable
            if (RM3.6) RM1.6 = 0;           // Analogue Low Alarm Disable
            if (RM3.5) RM1.5 = 0;           // Analogue No Average
            if (RM3.4) RM1.4 = 0;           // No Analogue Trip logging
            if (RM3.3) RM1.3 = 0;           // No Analogue RTC Logging
            if (RM3.2) RM1.2 = 0;           // No Analogue RTC Reporting
            Analog_Config[1] = RM1;         // Set configuretion
                                                    //
            }                               //
RecordLength += 2;                          //
RM1 = I2C_Buf[RecordLength]; RecordLength ++; // Get next analogue mask
if (RM1.7) Analog_High_Alarm[2]     = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.6) Analog_Debounce_High[2]  = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.5) Analog_Low_Alarm[2]      = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.4) Analog_Debounce_Low[2]   = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.3) {                                // The limits & debounce changed
            RM2 = I2C_Buf[RecordLength];    // Now change configuration
            RM3 = I2C_Buf[RecordLength+1]; //
            RM1 = Analog_Config[2];         //
                                            //
            if (RM2.7) RM1.2 = 1;           // Analogue High Alarm Enable
            if (RM2.6) RM1.6 = 1;           // Analogue Low Alarm Enable
            if (RM2.5) RM1.5 = 1;           // Analogue Average
            if (RM2.4) RM1.4 = 1;           // Analogue Trip logging
            if (RM2.3) RM1.3 = 1;           // Analogue RTC Logging
            if (RM2.2) RM1.2 = 1;           // Analogue RTC Reporting
            if (RM3.7) RM1.2 = 0;           // Analogue High Alarm Disable
            if (RM3.6) RM1.6 = 0;           // Analogue Low Alarm Disable
            if (RM3.5) RM1.5 = 0;           // Analogue No Average
            if (RM3.4) RM1.4 = 0;           // No Analogue Trip logging
            if (RM3.3) RM1.3 = 0;           // No Analogue RTC Logging
            if (RM3.2) RM1.2 = 0;           // No Analogue RTC Reporting
            Analog_Config[2] = RM1;         // Set configuretion
            }                               //
RecordLength += 2;                          //
RM1 = I2C_Buf[RecordLength]; RecordLength ++; // Get next analogue mask
if (RM1.7) Analog_High_Alarm[3]     = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.6) Analog_Debounce_High[3]  = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.5) Analog_Low_Alarm[3]      = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.4) Analog_Debounce_Low[3]   = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.3) {                                // The limits & debounce changed
            RM2 = I2C_Buf[RecordLength];    // Now change configuration
            RM3 = I2C_Buf[RecordLength+1]; //
            RM1 = Analog_Config[3];         //
                                            //
            if (RM2.7) RM1.2 = 1;           // Analogue High Alarm Enable
            if (RM2.6) RM1.6 = 1;           // Analogue Low Alarm Enable
            if (RM2.5) RM1.5 = 1;           // Analogue Average
            if (RM2.4) RM1.4 = 1;           // Analogue Trip logging
            if (RM2.3) RM1.3 = 1;           // Analogue RTC Logging
            if (RM2.2) RM1.2 = 1;           // Analogue RTC Reporting
            if (RM3.7) RM1.2 = 0;           // Analogue High Alarm Disable
            if (RM3.6) RM1.6 = 0;           // Analogue Low Alarm Disable
            if (RM3.5) RM1.5 = 0;           // Analogue No Average
            if (RM3.4) RM1.4 = 0;           // No Analogue Trip logging
            if (RM3.3) RM1.3 = 0;           // No Analogue RTC Logging
            if (RM3.2) RM1.2 = 0;           // No Analogue RTC Reporting
            Analog_Config[3] = RM1;         // Set configuretion
            }                               //
RecordLength += 2;                          //
RM1 = I2C_Buf[RecordLength]; RecordLength ++; // Get next analogue mask
if (RM1.7) Analog_High_Alarm[4]     = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.6) Analog_Debounce_High[4]  = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.5) Analog_Low_Alarm[4]      = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.4) Analog_Debounce_Low[4]   = I2C_Buf[RecordLength]; RecordLength++;
if (RM1.3) {                                // The limits & debounce changed
            RM2 = I2C_Buf[RecordLength];    // Now change configuration
            RM3 = I2C_Buf[RecordLength+1]; //
            RM1 = Analog_Config[4];         //
                                            //
            if (RM2.7) RM1.2 = 1;           // Analogue High Alarm Enable
            if (RM2.6) RM1.6 = 1;           // Analogue Low Alarm Enable
            if (RM2.5) RM1.5 = 1;           // Analogue Average
            if (RM2.4) RM1.4 = 1;           // Analogue Trip logging
            if (RM2.3) RM1.3 = 1;           // Analogue RTC Logging
            if (RM2.2) RM1.2 = 1;           // Analogue RTC Reporting
            if (RM3.7) RM1.2 = 0;           // Analogue High Alarm Disable
```

```
                        if (RM3.6) RM1.6 = 0;          // Analogue Low Alarm Disable
                        if (RM3.5) RM1.5 = 0;          // Analogue No Average
                        if (RM3.4) RM1.4 = 0;          // No Analogue Trip logging
                        if (RM3.3) RM1.3 = 0;          // No Analogue RTC Logging
                        if (RM3.2) RM1.2 = 0;          // No Analogue RTC Reporting
                        Analog_Config[4] = RM1;        // Set configuretion
                        }                              //
                RecordLength += 2;                     //
                                                       //
                RM1 = I2C_Buf[RecordLength]; RecordLength ++; // Get Digital inputs mask
                RM2 = I2C_Buf[RecordLength]; RecordLength ++; // Get Digital inputs mask
                RM3 = I2C_Buf[RecordLength]; RecordLength ++; // Get Digital inputs mask

                                                       // Change the debounce times
                if (RM1.7) Digital_Debounce[0] = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.6) Digital_Debounce[1] = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.5) Digital_Debounce[2] = I2C_Buf[RecordLength]; RecordLength++;
                if (RM1.4) Digital_Input_3_Trip_Reporting  = RM3.4; // Input 3 reporting
enable
                if (RM1.3) Digital_Input_2_Trip_Reporting  = RM3.3; // Input 2 reporting
enable
                if (RM1.2) Digital_Input_1_Trip_Reporting  = RM3.2; // Input 1 reporting
enable
                if (RM1.1) Digital_Input_RTC_Logging       = RM1.0; // Inputs RTC logging
enable

                if (RM2.7) Digital_Input_3_Polarity        = RM3.7; // Input 3 polarity
                if (RM2.6) Digital_Input_2_Polarity        = RM3.6; // Input 2 polarity
                if (RM2.5) Digital_Input_1_Polarity        = RM3.5; // Input 1 polarity
                if (RM2.4) Digital_Input_RTC_Reporting     = RM3.1; // Inputs RTC reporting
                if (RM2.3) Digital_Input_RTC_Logging       = RM3.0; // Inputs RTC logging

                Save_IO_Parameters(void);                          // Save this set-up as
the
                                                                   // future default
                Load_IO_Parameters(void);                          // Make sure we use the
                break;                                             // right set-up
                }


/* Status Modify */     case 6:
                {                                                  // **** Status Modify
                RecordLength = 16;

                RM1 = I2C_Buf[3];                                  // Mask for Output 1
                if (RM1.7) Output1_Pattern_Reload = I2C_Buf[4];    // Change pattern
                if (RM1.6) Output1_Repeats = I2C_Buf[5];           // Change repeats
                if (RM1.5) Output1_Delay   = I2C_Buf[6];           // Change bit delays
                Output1_Counter            = Output1_Delay;        // Start again…
                if (RM1.4) {if (RM1.3) Output_1_ON;                // Activate…
                           else        Output_1_OFF;}              // …or deactivate the
output

                RM1 = I2C_Buf[7];                                  // Mask for Output 2

                if (RM1.7) Output2_Pattern_Reload = I2C_Buf[8];    // Change pattern
                if (RM1.6) Output2_Repeats = I2C_Buf[9];           // Change repeats
                if (RM1.5) Output2_Delay   = I2C_Buf[10];          // Change bit delays
                Output2_Counter              = Output2_Delay;      // Start again…
                if (RM1.4) {if (RM1.3) Output_2_ON;                // Activate…
                           else        Output_2_OFF;}              // …or deactivate the
output

                RM1 = I2C_Buf[11];                                 // Mask for Output 3

                if (RM1.7) Output3_Pattern_Reload = I2C_Buf[12];   // Change pattern
                if (RM1.6) Output3_Repeats = I2C_Buf[13];          // Change repeats
                if (RM1.5) Output3_Delay   = I2C_Buf[14];          // Change bit delays
                Output1_Counter              = Output3_Delay;      // Start again…
                if (RM1.4) {if (RM1.3) Output_3_ON;                // Activate…
                           else        Output_3_OFF;}              // …or deactivate the
output

                RM1 = I2C_Buf[15];                                 // Mask for Output 4
                if (RM1.7) Output4_Pattern_Reload = I2C_Buf[16];   // Change pattern
                if (RM1.6) Output4_Repeats = I2C_Buf[17];          // Change repeats
                if (RM1.5) Output4_Delay   = I2C_Buf[18];          // Change bit delays
                Output4_Counter              = Output4_Delay;      // Start again…
                if (RM1.4) {if (RM1.3) Output_4_ON;                // Activate…
```

```
                                        else        Output_4_OFF;}              // …or deactivate the
output

                                RM1 = I2C_Buf[19];                              // Mask for power
handling
                                if (RM1.7) PowerDwn_Trip_Reporting = RM1.3;     // Power down trigger
report
                                if (RM1.6) PowerDwn_Trip_Logging   = RM1.2;     // Power down trigger log
                                if (RM1.5) PowerDwn_RTC_Reporting  = RM1.1;     // Power status report
                                if (RM1.4) PowerDwn_RTC_Logging    = RM1.0;     // Power status log

                                RM1 = I2C_Buf[20];                              // Mask for counter
                                if (RM1.7) Counter_Mode          = RM1.3;       // Frequency/Counter mode
                                if (RM1.6) Counter_RTC_Logging   = RM1.2;       // RTC Logging
                                if (RM1.5) Counter_RTC_Reporting = RM1.1;       // RTC Reporting
                                if (RM1.4) Counter_Trip_Logging  = RM1.0;       // Trigger Logging

                                RM1 = I2C_Buf[21];                              //
                                if (RM1.5) Counter_High_Alarm_Enable  = RM1.4;  // High trigger reporting
                                if (RM1.1) Counter_Low_Alarm_Enable   = RM1.0;  // Low trigger reporting
                                if (RM1.7) { Counter_High_Alarm_Low = I2C_Buf[23]; // New high alarm
                                             Counter_High_Alarm_High = I2C_Buf[22];}//
                                if (RM1.6) Counter_Debounce_High   = I2C_Buf[24]; // New high alarm
debounce
                                if (RM1.3) { Counter_Low_Alarm_Low  = I2C_Buf[26]; // New low alarm
                                             Counter_Low_Alarm_High = I2C_Buf[25];} //
                                if (RM1.2)  Counter_Debounce_Low   = I2C_Buf[27]; // New low alarm debounce
                                                                                //
                                Save_IO_Parameters(void);                       // Save these parameters
                                Load_IO_Parameters(void);                       // Make sure we use the
                                                                                // right set-up

                                break;                                          //
                                }
                        }                                                       // Done executing the
                                                                                // current command

        }                                                                       // Done with the current
                                                                                // message

if ((RecordCommand==4)||(RecordCommand==1))                                     // Return only if needed
        {                                                                       // (in the case of a
        Send_The_Log();                                                         // memory read or a
status
        }                                                                       // request)

Command_Parameters[0] = MessageNumber;                                          // Done with all commands
Result = Issue_Command(__Delete_SMS_Message);                                   // Delete the current
                                                                                // message
```

11. Product development

11.1 Production Hardware

Bed-of-Nails

This jig is used during assembly to test and program the unit. It is also used to enter the unit into the database.

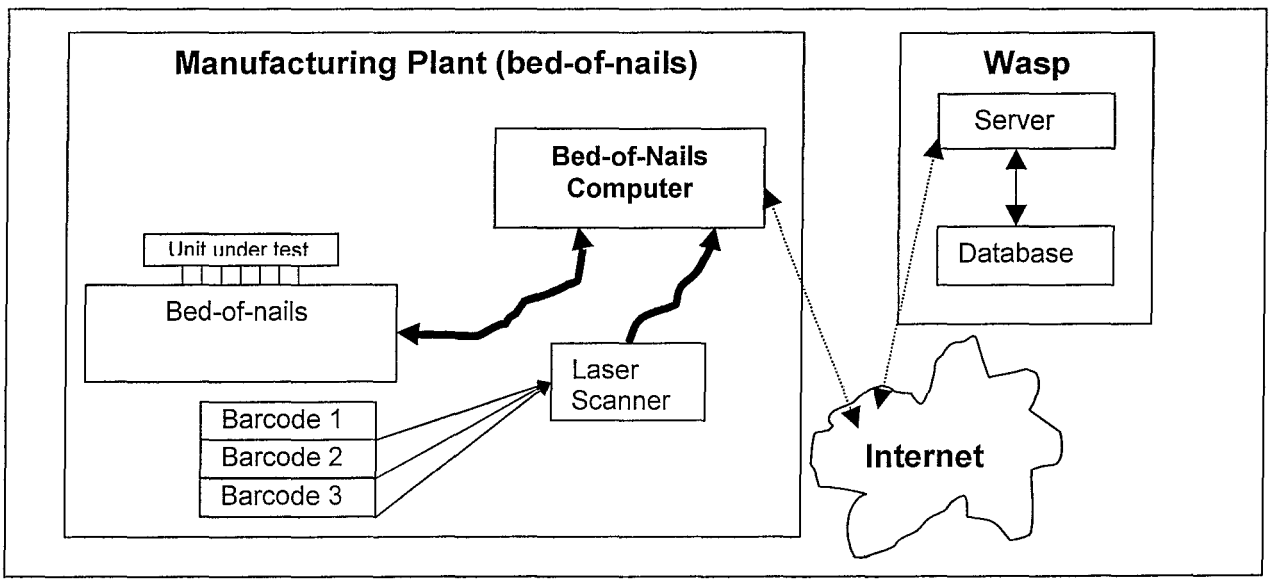


Figure 18. The manufacturing process.

The bed-of-nails is a pneumatically actuated jig with spring loaded nails that make contact with special pads on the circuit board. These contact points allow for the testing and programming of all the relevant components on the board. The bed-of-nails speeds up the manufacturing process by quickly pointing out continuity and component faults before the board is programmed. There are less operator processes that can go wrong or cause bottlenecks. A classic example is a free-standing programmer. To program parts before, or after assembly takes up much more time and almost always leads to bottlenecks.

By incorporating the programming functionality into the testing process, all these problems are eliminated. It is of course, important to adhere to the parts manufacturers programming specifications for production purposes. This is usually a combination of programming and verifying voltages and times, and the number of programming pulses per byte (or word).

The bed-of-nails is connected to a host computer via a serial cable. Bi-directional communications allows the host to control the bed-of-nails as well as to read back the results of the tests. The host computer also has a barcode scanner to scan in all the major components that make up the unit together with the unit serial number. This information is sent via the internet to a server that interacts with the Wasp database. At any time during the product's life, the production results can be analysed and trends examined. This may lead to the prevention of product failures by adapting the production processes.

Soak Test

After assembly the units will go through a burn-in, or soak testing period. Most failures will occur within the first few hours of operation, so the units are tested on such a specially designed test bench. The units are exercised through their full capacities in terms of interfaces as well as communication. To save time and costs, the soak test process is automated.

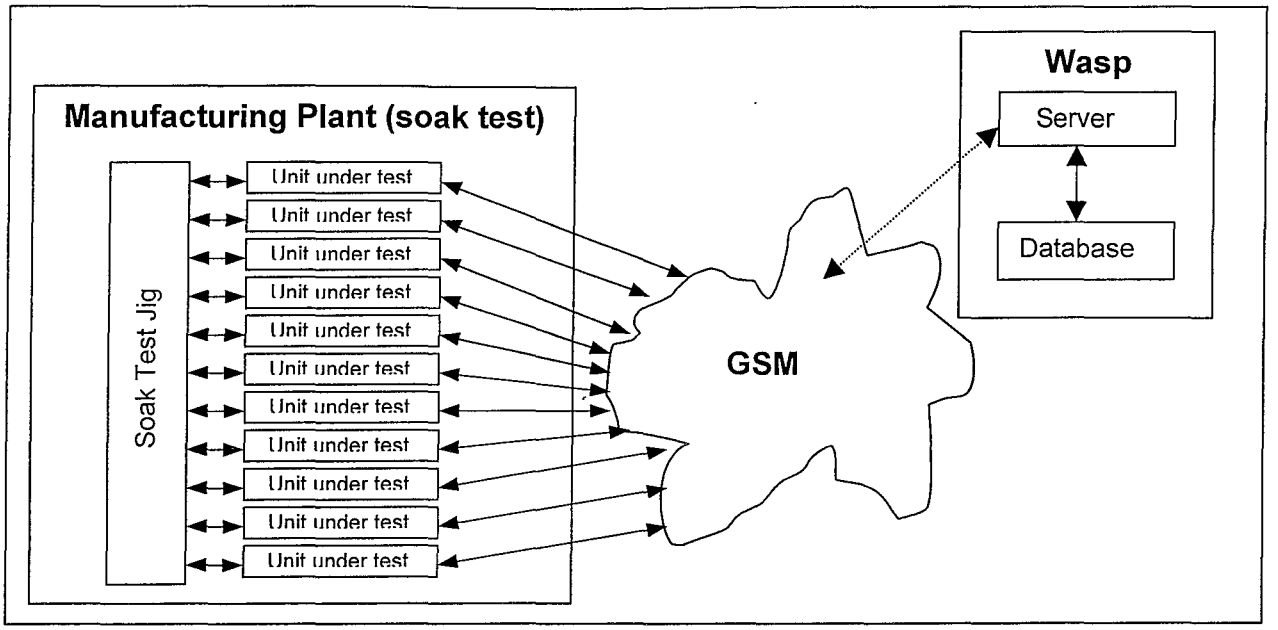


Figure 19. The soak test jig.

Units that don't pass a four hour soak test, will go to the repair/rework station.

Repair/Rework

The repair/rework station will have all the standard electronic test equipment together with :

1) A device to interface with the inputs and outputs of the unit.
2) A jig that can translate the TTL serial communication on the circuit board to RS-232 levels (breakout).
3) A computer to monitor the communications on the circuit board (two unused serial ports needed).
4) A device to monitor GSM communications.

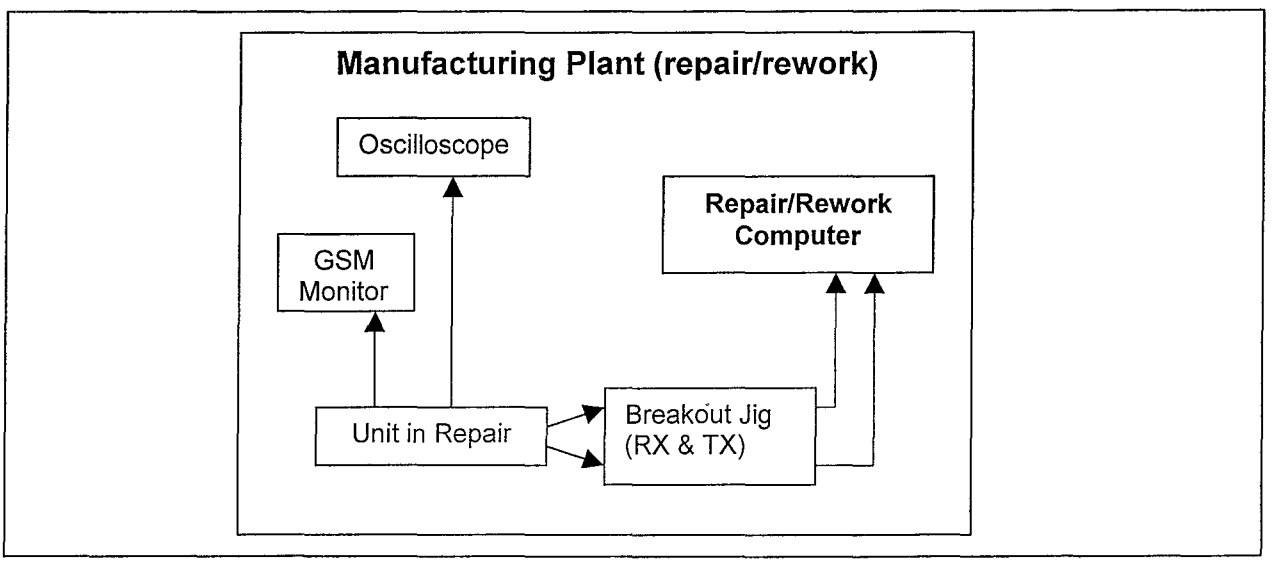


Figure 20. The rework process.

After repairing a unit, it will always go back to soak testing.

11.2 Production Software

Bed-of-Nails

The bed-of-nails host software controls the assembly process from where the newly populated circuit board is placed in the jig, to where the unit goes to soak testing. It also interacts with the database.

For clarity, the complete component placement process for a board with surface components on one side only, and a few through-hole components, is described in a flow diagram :

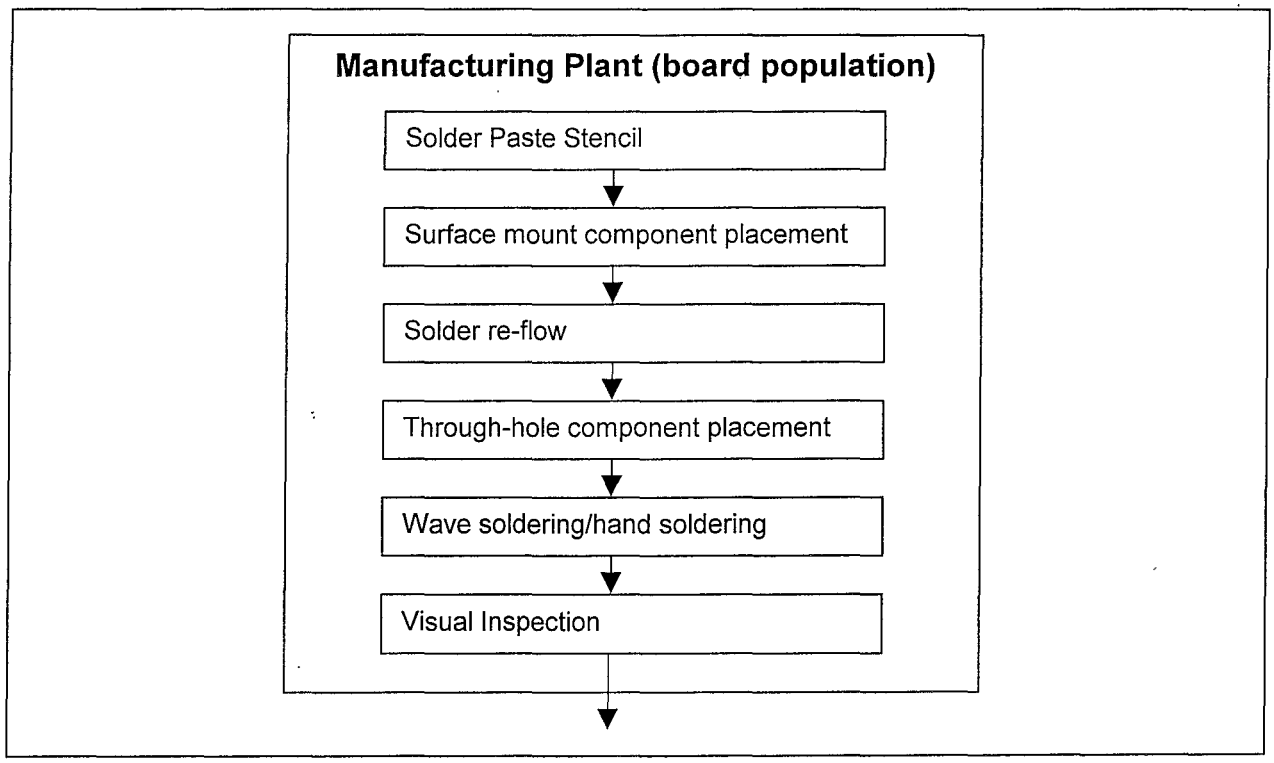


Figure 21. The component population process.

The 'software' for the bed-of-nails, is a combination of the software inside the jig as well as the software running on the host computer.

The complete assembly as controlled by the host computer software, is described in a flow diagram :

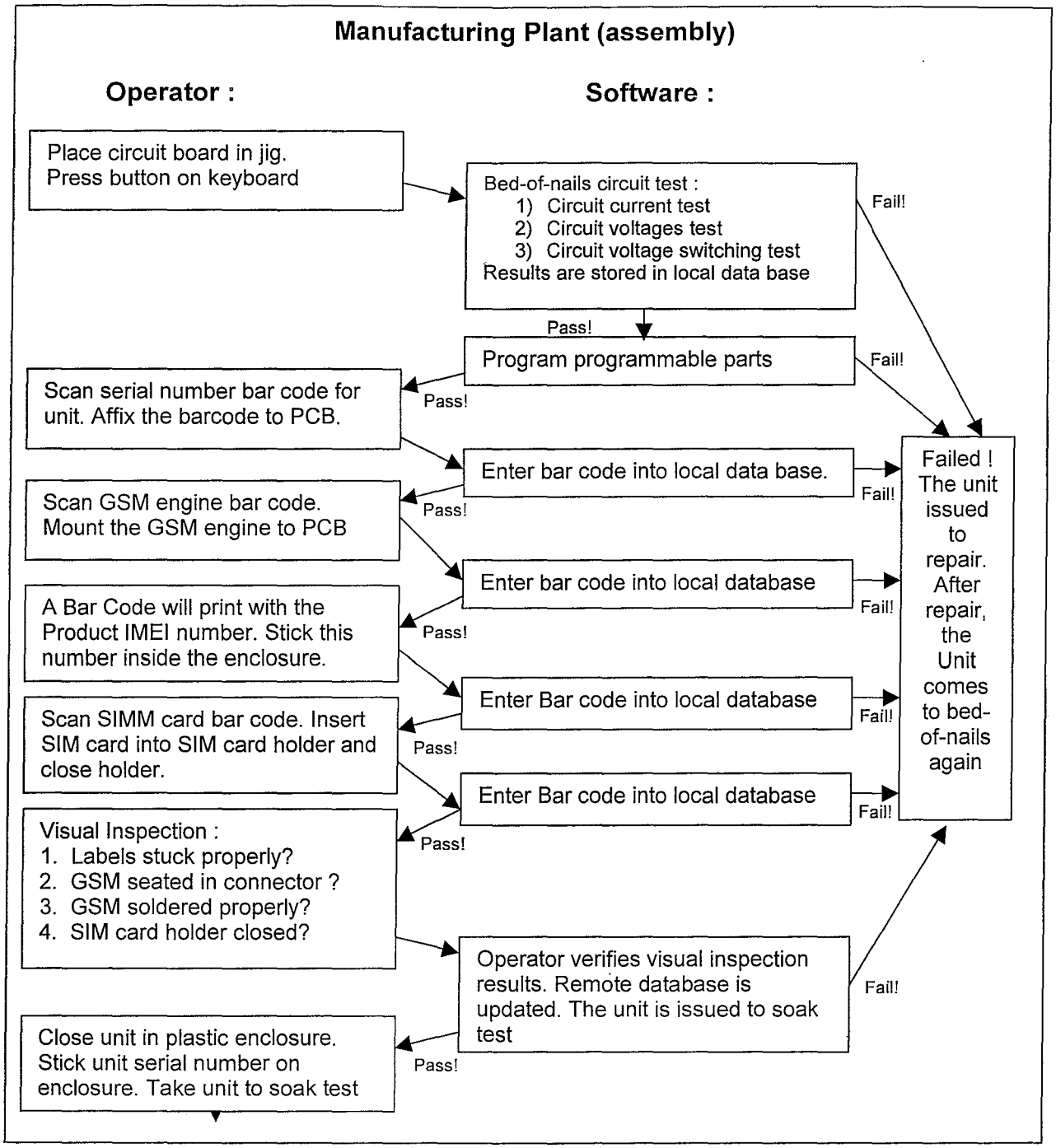


Figure 22. The assenbly process.

Soak Test

When a unit is placed in the database during assembly, it is assigned a default status of 'ready to soak'. When a pre-defined input is received from that unit, it signals the soak test server software to start soak testing the unit. If, at any time, the unit fails the soak test, all the software clients will be notified. When the unit passes, the software clients are also notified.

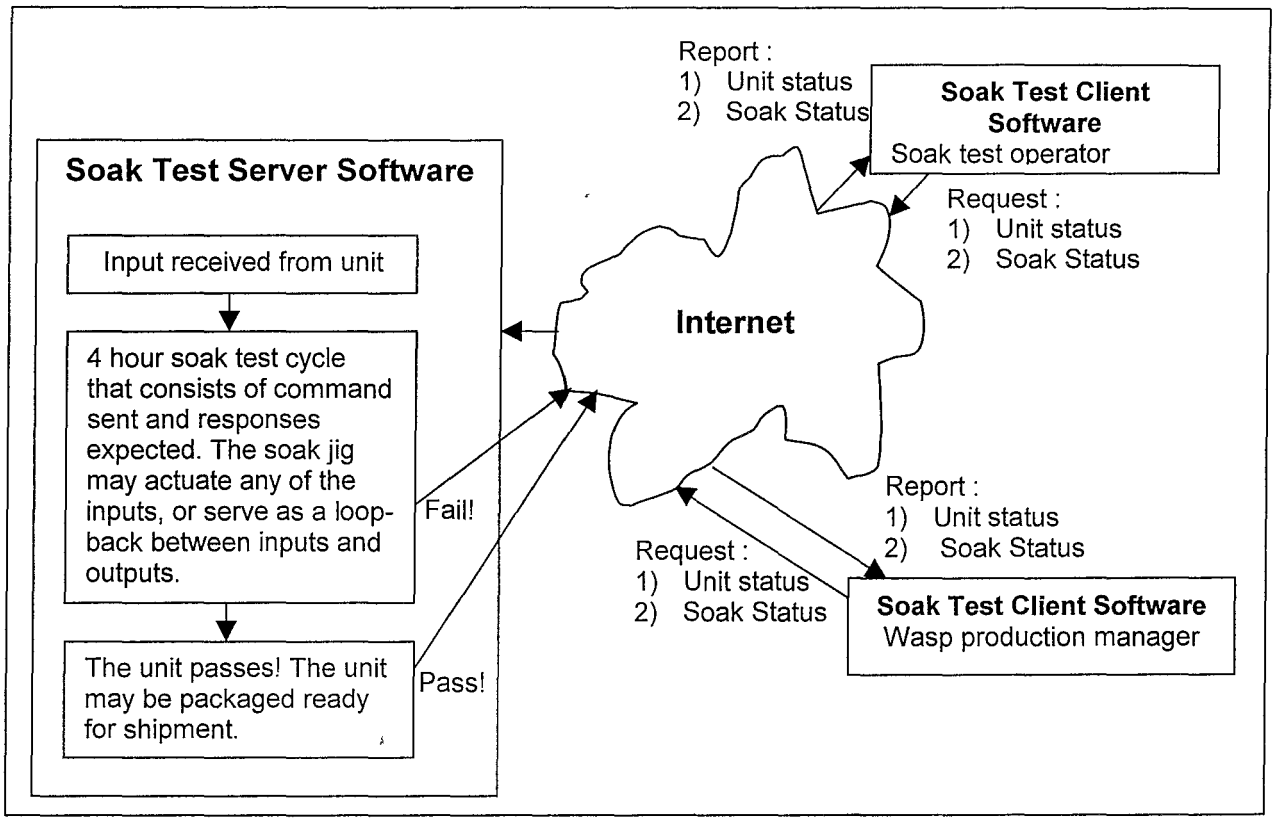


Figure 23. The soak test process.

The clients may request general soak test reports, or unit-specific reports from the server at any time.

Repair/Rework

This software simply monitors the communication on the circuit board. It replaces communication strings with looked-up English explanation strings. For certain combinations, it may also display certain repair suggestions like "Check connector A".

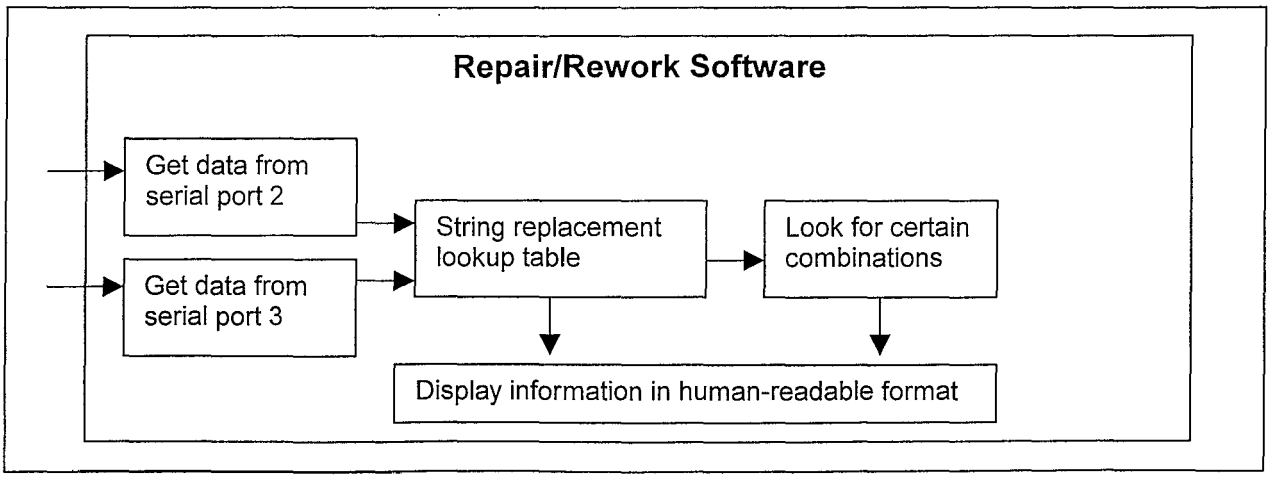


Figure 24. The rework/repair software.

12. Milestone 3 Evaluation

By this time, all the major jigs and production/repair software were finished. The bed-of-nails was tested in the production environment. The PCB was completed with the revised schematic and the bill of materials generated. Suppliers were found for all the components listed.

The protocol detail was defined. A prototype was built and programmed with the firmware using the bed-of-nails. The firmware and user software were tested together to verify the protocol.

Chapter 4 Design Revision

13. Sample Run

13.1 Testing for Manufacturability or Assembly Problems

The sample run is a small run of around 20 units to 100 units. Its purpose is to highlight any potential problems with the process.

13.2 Revision

Seeing as no such run has taken place so far, no comments can be made.

14. Production Quality Control (QC) Plan

14.1 Procedures for Quality Control

The quality control plan designed in the previous chapter is tested at this point. The boards are inspected for visible defects, and the operator processes monitored. Operator process inefficiencies can be identified and corrected.

Batch Procedure Development
The units are inspected to see if the quality can be expected to be consistent. Some input from the operators may be helpful at this point.

Component Manufacturer Specification
A few of the units are tested for full specification compliance and environmental tolerance.

15. Production Tools

15.1 Failure Mode Effect and Analysis (FMEA)

Hardware
The production hardware is checked to see if it will be able to handle large production runs.

Software
The production software is tested for full specified functionality and database access speed.

15.2 Revision

All of the potential failures must be discovered during the sample run.

15.3 Corrective Measures

The appropriate corrective measures and steps are taken. The documentation must also be updated and the operators retrained if necessary.

16. Production Manual

16.1 Final Copy

The final copy of this manual is signed off to the manufacturer. The procedures and methods stipulated in this document cannot be changed unless agreed to in writing.

17. Repair Manual

17.1 Final Copy

The final copy of this manual is signed off to the manufacturer. By this time the repair technicians must be able to repair minor defects.

18. Product Failure Mode Effect and Analysis (FMEA)

18.1 Hardware

The component under scrutiny goes into column 2. The failure mode is the state that the component is expected to fail in. The "Probability" is the perceived chance of such a failure taking place. "Effects" is what the customer will see in the behaviour of the product. The purpose of this analysis is to identify the weaker point in the design and to take preventative measures. This will result in a more robust and reliable unit. All instances where the potential to be a problem is higher than 4, the design around the particular component will be scrutinised to improve the reliability.

Hardware FMEA

This table relates the probability, failure mode, and effects of certain hardware failures in the product :

	Component	Function	Failure Mode	Probability	Effects	Potential to be a problem
1	PIC 16C77	Microcontroller	Reset Blown output Blown input	• Very low • Low • Low	The unit will not work at all An output may not work An input not read correctly	4 3 3
2	PCF8593	Real time clock	No response on I^2C bus Clock not running	• Very low • Very Low	The unit will not work at all The date/time function will not work	4 4
3	Super Cap	Back-up for RTC	Leaky Shorted	• Very Low • Very Low	The real time clock back-up won't work It will cause the regulators to fail with the result that the unit will stop functioning	4
4	LT1076	Switch mode regulator	Output zero	• Very low	The unit will not work at all	4
5	PE1902	Switch mode choke	Wire broken	• Medium	The unit will not work at all	3
6	470uF Cap	Switch mode smoothing cap	Shorted	• Very low	The unit will not work at all	4
7	Wismo	GSM Engine	Software hangs RF stages fail	• Medium • Low	The unit will reset itself after a short time The communication won't work, the outputs will carry on as usual	4 3
8	Wismo antenna	Wismo antenna	Broken off at engine	• Medium	The communication won't work, the outputs will carry on as usual	3
9	40 way connector	Wismo connector	• Bad solder joint	• Medium	The communication won't work, the outputs will carry on as usual	3
10	NJM2370	5V Regulator	No output	• Very low	The unit will not function at all	4

11	NJM2370	SIM Regulator	No output	• Very low	The communication won't work, the outputs will carry on as usual	4
12	Diode bridge	Input voltage rectification	Open circuit	• Very low	The unit will not function at all	4
13	Tranzorb	Input voltage spike protection	Open circuit	• Very low	The unit will function as usual. The problem will probably never show.	4
14	Screw down terminals	Input, outputs, and power input connections	Pin breaks PCB pad	• Very low	The associated function of the unit will not work.	4

Hardware Corrective Action

This table relates the possible actions taken:

	Component	Function	Failure Mode	Effects	Action to be Taken
1	PIC 16C77	Microcontroller	Reset Blown output Blown input	The unit will not work at all An output may not work An input not read correctly	No effective preventative action Series resistor to limit drive current. Internal zeners will clamp voltage spikes Effective input protection already present.
2	PCF8593	Real time clock	No response on I^2C bus Clock not running	The unit will not work at all The date/time function won't work	Series de-coupling resistor protect RTC from spikes on the rail. QC the crystal
3	Super Cap	Back-up for RTC	Leaky Shorted	The real time clock back-up won't work It will cause the regulators to fail with the result that the unit will stop functioning	No effective preventative action
4	LT1076	Switch mode regulator	Output zero	The unit will not work at all	Regulator is protected against overheating. It is also protected against voltage spikes (tranzorb) and reverse polarity. It is not possible to effectively protect the 6V rail to the outside of the unit. This responsibility lies with the installer.
5	PE1902	Switch mode choke	Wire broken	The unit will not work at all	The choke will be clamped down with a cable tie.
6	470uF Cap	Switch mode smoothing cap	Shorted	The unit will not work at all	No effective preventative action

7	Wismo	GSM Engine	Software hangs RF stages fail	The unit will reset itself after a short time The communication won't work, the outputs will carry on as usual	This issue is addressed in the firmware. The engine is reset if it acts unusually. Antennas will be issued with the units to prevent damage due to home-made antennas
8	Wismo antenna	Wismo antenna	• Broken off at engine	The communication won't work, the outputs will carry on as usual	A strain-relief cable tie on the inside of the enclosure will prevent the antenna from being pulled from the engine.
9	40 way connector	Wismo connector	• Bad solder joint	The communication won't work, the outputs will carry on as usual	Visual inspection after component placement
10	NJM2370	5V Regulator	No output	The unit will not function at all	No effective preventative action
11	NJM2370	SIM Regulator	No output	The communication won't work, the outputs will carry on as usual	No effective preventative action
12	Diode bridge	Input voltage rectification	Open circuit	The unit will not function at all	These diodes also act as fuses to the unit, so a diode failure in the event of a bigger failure is desirable.
13	Tranzorb	Input voltage spike protection	Open circuit	The unit will function as usual. The problem will probably never show.	A big (1.5Ke) tranzorb will be used.

18.2 Firmware

Firmware FMEA

This table relates the probability, failure mode, and effects of certain software modules in the unit :
The module under scrutiny goes into column 2. The failure mode is the state that the software is expected to fail in. The "Probability" is the perceived chance of such a failure taking place. "Effects" is what the customer will see in the behaviour of the product. The purpose of this analysis is to identify the weaker point in the design and to take preventative measures. This will result in a more robust and reliable unit. All instances where the potential to be a problem is higher than 4, the design around the particular module will be scrutinised to improve the reliability.

	Module	Function	Failure Mode	Probability	Effects	Potential to be a problem
1	UART Character received	Receives incoming UART data	No known failure conditions	• Very low	This code is proven	4
2	UART Character transmitted	Transmits data string to UART	Wismo CTS is held too long	• Very low	This code is proven	4
3	Time Base : 16ms	Debouncing/Sampling	No known failure conditions	• Very low	This code is proven	4

4	Time Base : 100ms	Output management	No known failure conditions	• Very low	This code is proven	4
5	Time Base : 1s	Timers	No known failure conditions	• Very low	This code is proven	4
6	Set-up micro	Load all set-up values into micro	No known failure conditions	• Very low	This code is proven	4
7	Download set-up values	Download set-up values into non-volatile memory with cable	No known failure conditions	• Very low	This code is used only when downloading data - not in normal use	4
8	Load_String()	Loads string from non-volatile memory	No known failure conditions	• Very low	This code is proven	4
9	Save_String()	Loads string from non-volatile memory	No known failure conditions	• Very low	This code is proven	4
10	Load_IO_Parameters()	Loads default set-up values for functionality from non-volatile memory	No known failure conditions	• Very low	This code is proven	4
11	Save_IO_Parameters()	Saves default set-up values for functionality in non-volatile memory	No known failure conditions	• Very low	This code is proven	4
12	Setup_RTC()	Sets up real time clock to function as required	No known failure conditions	• Very low	Real time clock functions won't work properly. Erroneous time/date stamps	4
13	Get_Time()	Gets time from real time clock and processes the data	No known failure conditions	• Very low	Erroneous time/date stamps	4
14	Issue_Command(Command_Number)	Sends command to GSM engine and waits for response	Wismo CTS is held too long	• Very low	Command fails to execute. The GSM engine may expect more data.	4
15	Resolve_Message_Header()	Checks message integrity	Garbage in message, but password and terminator are intact	• Very low	Message rejected	4
16	Get_Next_Command()	Gets next command in message	Garbage in record, but structure and terminator are intact	• Very low	Wrong set-up saved in non-volatile memory. The unit will behave very strangely.	4
17	Send_The_Log()	Sends the saved log	Transmission failure	•Medium high	The unit re-tries until the message is sent.	4
18	ASCII_2_HEX()	Converts ASCII characters to hex	No known failure conditions	• Very low	This code is proven	4

19	BIN2BCD(BCDBIN_In)	Converts values into BCD	No known failure conditions	• Very low	This code is proven	4
20	BCD2BIN(BCDBIN_In)	Converts BCD into values	No known failure conditions	• Very low	This code is proven	4

Firmware Corrective Action

This table relates the probability, failure mode, and effects of certain software modules in the unit :
The module under scrutiny goes into column 2. The failure mode is the state that the software is expected to fail in. The "Probability" is the perceived chance of such a failure taking place. "Effects" is what the customer will see in the behaviour of the product. The purpose of this analysis is to identify the weaker point in the design and to take preventative measures. This will result in a more robust and reliable unit. All instances where the potential to be a problem is higher than 4, the design around the particular module will be scrutinised to improve the reliability.

	Module	Function	Failure Mode	Probability	Effects	Action to be Taken
1	UART Character received	Receives incoming UART data	No known failure conditions	• Very low	This code is proven	No action
2	UART Character transmitted	Transmits data string to UART	Wismo CTS is held too long	• Very low	This code is proven	No action
3	Time Base : 16ms	Debouncing/Sampling	No known failure conditions	• Very low	This code is proven	No action
4	Time Base : 100ms	Output management	No known failure conditions	• Very low	This code is proven	No action
5	Time Base : 1s	Timers	No known failure conditions	• Very low	This code is proven	No action
6	Set-up micro	Load all set-up values into micro	No known failure conditions	• Very low	This code is proven	No action
7	Download set-up values	Download set-up values into non-volatile memory with cable	No known failure conditions	• Very low	This code is used only when downloading data - not in normal use	No action
8	Load_String()	Loads string from non-volatile memory	No known failure conditions	• Very low	This code is proven	No action
9	Save_String()	Loads string from non-volatile memory	No known failure conditions	• Very low	This code is proven	No action
10	Load_IO_Parameters()	Loads default set-up values for functionality from non-volatile memory	No known failure conditions	• Very low	This code is proven	No action

11	Save_IO_Parameters()	Saves default set-up values for functionality in non-volatile memory	No known failure conditions	• Very low	This code is proven	No action
12	Setup_RTC()	Sets up real time clock to function as required	No known failure conditions	• Very low	Real time clock functions won't work properly. Erroneous time/date stamps	Periodic updates are suggested
13	Get_Time()	Gets time from real time clock and processes the data	No known failure conditions	• Very low	Erroneous time/date stamps	Code tested - no action
14	Issue_Command(Command_Number)	Sends command to GSM engine and waits for response	Wismo CTS is held too long	• Very low	Command fails to execute. The GSM engine may expect more data.	The code automatically handles all error conditions - no action
15	Resolve_Message_Header()	Checks message integrity	Garbage in message.	• Very low	Message rejected	No action
16	Get_Next_Command()	Gets next command in message	Garbage in record, but structure and terminator are intact	• Very low	Wrong set-up saved in non-volatile memory. The unit will behave very strangely.	Rigorous testing of Server command generation code.
17	Send_The_Log()	Sends the saved log	Transmission failure	• Very low	The unit re-tries until the message is sent.	No action
18	ASCII_2_HEX()	Converts ASCII characters to hex	No known failure conditions	• Very low	This code is proven	No action
19	BIN2BCD(BCDBIN_In)	Converts values into BCD	No known failure conditions	• Very low	This code is proven	No action
20	BCD2BIN(BCDBIN_In)	Converts BCD into values	No known failure conditions	• Very low	This code is proven	No action

19. Packaging

19.1 Design

As these products will be handled mainly by installation personnel and not by the end-user himself, it is suggested that no graphics be printed on the box. A basic off-the-shelf corrugated cardboard box with two bubble pouches is suggested. One bubble pouch is for the unit, and the other for the GSM antenna. The box must be long enough to be able to take both the unit and the GSM antenna.

19.2 Die-Cutting Tools

No die-cutting tool would be required if an off-the-shelf product is used.

19.3 Corrective Measures

None suggested.

20. Enclosure

20.1 Tooling

The PCB was designed to be used in a currently-used plastic enclosure, so no tooling for plastic injection would be necessary.

20.2 Effect on Assembly

The plastic enclosure to be used consists of two snap-on clam-type shells and two side plates. This greatly enhances the assembly process, as no retaining screws have to be screwed in and no lids have to be secured.

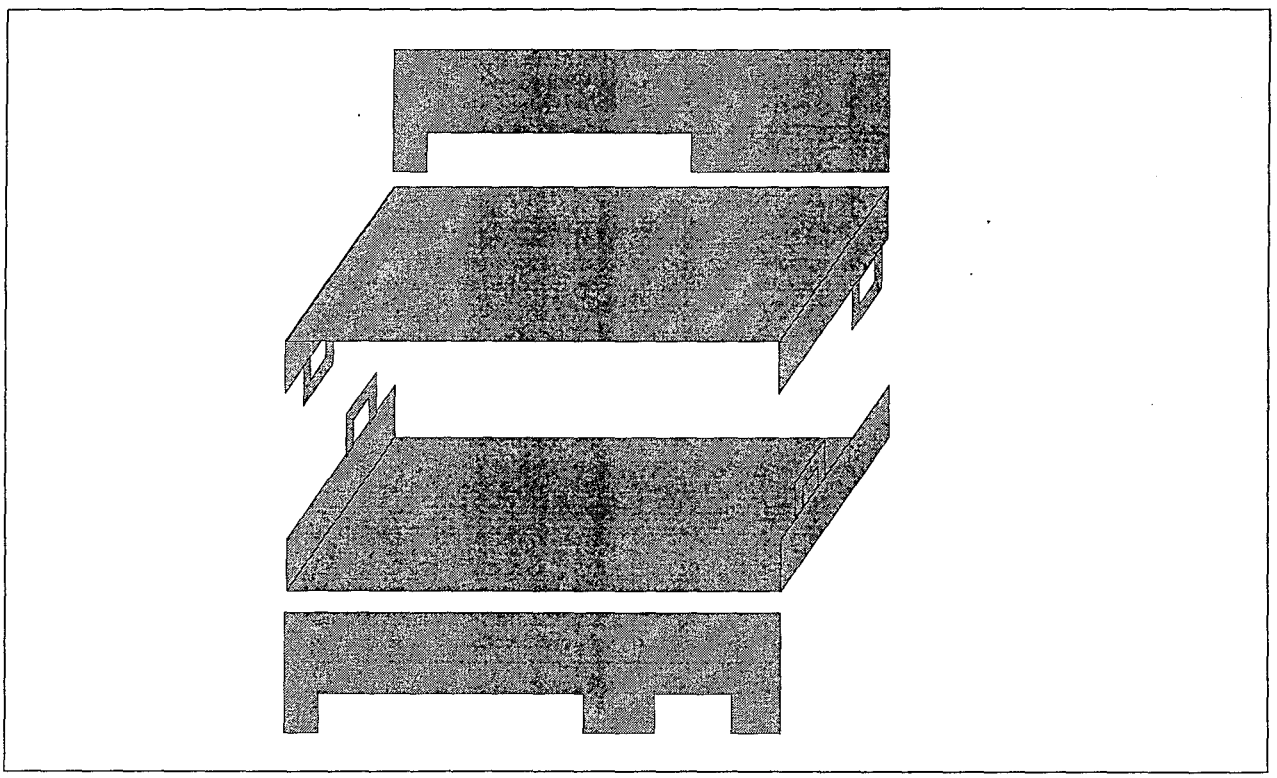


Figure 25. The snap-on clam-type enclosure.

20.3 Off-Tool Samples

The exact measurements of the enclosure are known

20.4 Corrective Measures

No measures necessary.

21. Customer Software

21.1 Failure Mode Effect and Analysis (FMEA)

No action is necessary.

22. Customer Manual

22.1 Information Package

User Manual

This manual describes the operational functionality and specifications in full to the customer.

Pocket Guide

This manual is a short reference to the most important aspects of the system. Ideally it should be the size of a business card and laminated, so the user can have it handy when he needs it.

23. Installation Manual

23.1 Information Package

Installation Manual

This manual can be the same manual as the Users' Manual as described previously.

Trouble Shooting Guide

A troubleshooting section in the Users' guide will enable the user to solve small problems without calling for support.

23.2 Corrective Measures

None suggested.

24. Bar Coded Labels

24.1 Design

Serial Number

This label will also carry all the relevant markings for specifications passed.

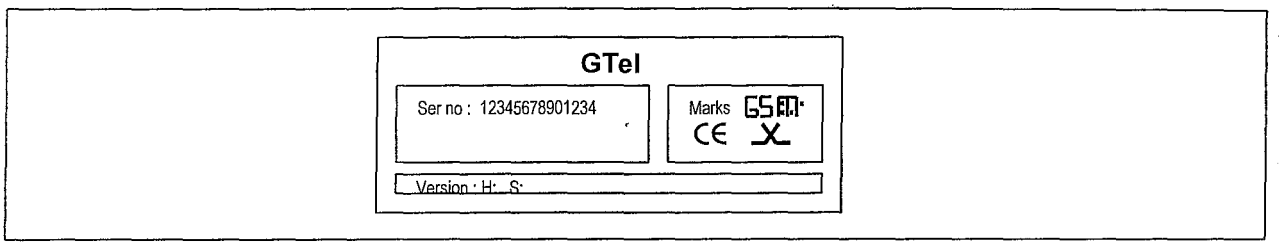


Figure 26. The bar-code label.

24.2 Check for Durability

Glue/Temperature/Colour fading/solvents/water

The label is fully tested in an environment similar to the one that is expected to encounter.

25. Face Plates

25.1 Design

Legend

Instead of a face plate, the legend will be pad-printed on the enclosure.

Tooling

The design for the legend will be handed over to the printers.

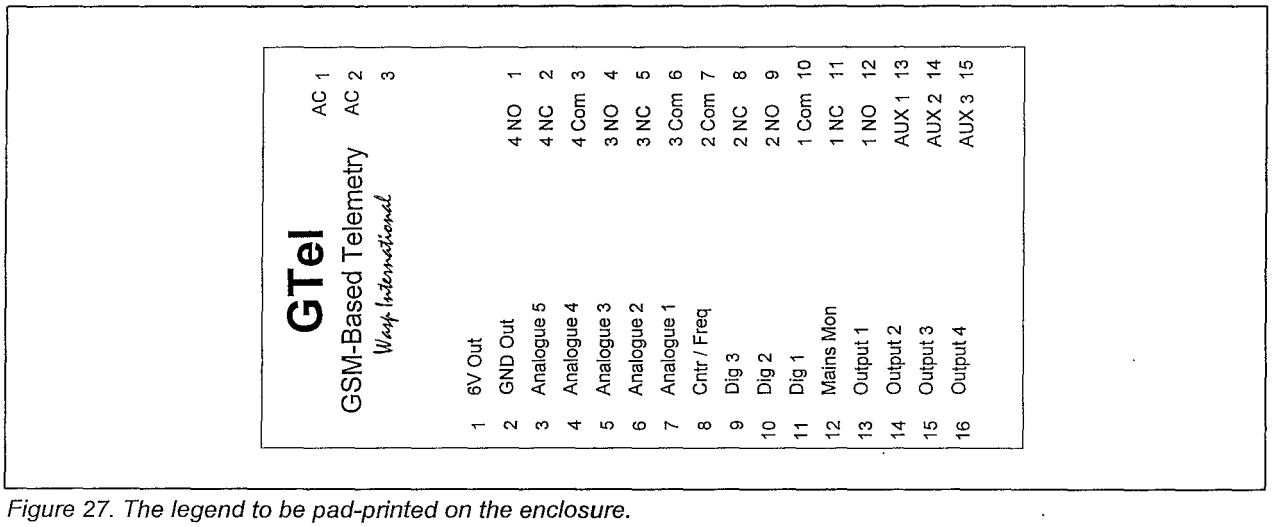


Figure 27. The legend to be pad-printed on the enclosure.

25.2 Check for Durability

Temperature

The paint screen was tested up to around 60°C without visible degrading.

Solvents

The paint can be removed with aggressive solvents only.

Water

The paint is water-proof.

25.3 Corrective Measures

None suggested.

26. Help Desk

26.1 Product Familiarising

Documentation

The help desk operators have to have in-depth insight into the system. They must have access to all the documentation, except the operational specifications, which contains all the trade secrets associated with the product.

Information Sessions

A few information sessions with the design engineers after the operators have studied the literature, will enable them to fully understand the system and have their questions answered. The system is also fully demonstrated, and the installation procedure explained.

Hands-on Training

After these information sessions, the operator will use a demonstration set-up to practise his new knowledge.

Frequently Asked Questions (FAQ)

All questions asked from this point by anyone, must be noted, and answered. These questions and answers will be taken up into a 'frequently asked questions' or 'FAQ' document. This document will be on-hand for the operators, and also be downloadable from the web-site.

26.2 Web Site Downloads

Public Information

Some documentation can be made available to the general public. This would include information leaflets, the FAQ, example applications, and other informative literature. The preferred format would be Adobe Acrobat ('.pdf' files).

Customer Information

This section will contain files like the FAQ, or updated versions of the client or point-to-point software for registered users.

27. Marketing

Even though the engineers are not involved with this part of the company, there should exist a cohesion between the design and marketing teams. There should be a relationship between what the product can

actually do, and what it is being sold as. Niche markets may be identified by the designers, that could be followed up by the marketing experts.

27.1 Strategy

Markets

One or two meetings can be held where the design team and the marketing team will discuss the product's full potential, and identify possible target markets. The prioritising of these target markets are up to the marketing team.

27.2 Design

Photography

The designers will provide a mock-up for photography purposes.

Web site

The flyers and other information leaflets designed by the marketing team will be posted on the web site.

28. Milestone 4 Evaluation

A small sample run was built and the manufacturing procedures verified. Procedures for quality control were drawn up. The production and repair manuals were completed. FMEA's were carried out on the production process, the product, and the user software and corrective measures were suggested. The production labels were finalised. The print tooling for the enclosure faceplate was completed and checked for accuracy.

The enclosure tooling was finished and the packaging material checked. A few photographs were taken of a mock-up for advertising and artwork purposes. The customer manual was completed.

Chapter 5 Pre-Production

29. Sample Run

29.1 Test for Manufacturability or Assembly Problems

Final Testing
Final tests made and adjustments to the production hardware or software are made.

30. Production Quality Control (QC) Plan

30.1 Procedures for Quality Control Implemented

Batch testing and Random sample testing is implemented. From this point the policy of batch rejection is enforced.

31. Production Manual

31.1 Final Copy

The final copy of this document is signed by both parties (design house and manufacturing house).

32. Repair Manual

32.1 Final Copy

The final copy of this document is signed by both parties (design house and manufacturing house).

33. Installer Training

33.1 Training Sessions

Training sessions are held for all the training personnel and some of the installers. The user software is used to demonstrate the functionality. All the interfacing procedures and electrical specifications are explained.

33.2 Train by Example

A full installation is done in a step-by-step procedure.

33.3 Q&A Session

After the installation session, a final question and answer session is held.

34. Bureau Operator Training

34.1 Training Sessions

34. Bureau Operator Training

34.1 Training Sessions

Training sessions similar to the ones for the installers are held. More time is spent with the user software and typical problems that may be encountered.

34.2 Q&A Session

A final question and answer session is held.

35. Cosmetic Checks

35.1 Production Marks

A few units are inspected to see if there are any production marks like burns or cracks in the enclosure.

35.2 Labels and Legend

All the labels are inspected to ensure they are affixed to the correct surfaces.

35.3 Packaging

A fully-assembled and packaged unit is inspected to ensure that the packaging is sufficient and as attractive as was planned.

36. Milestone 5 Evaluation

A small off-tool sample run was built and a few environmental tests performed. The units were tested in failure modes. The production quality control plan is implemented. All the outstanding documentation was finalised and training completed.

Chapter 6 Testing and Validation

37. Protocol Robustness and System Performance

The terminal was tested by sending 50 SMS messages to the unit, and by forcing it to send 50 messages back. Because the message path for both directions was the same, the times logged are representative of the system.

37.1 Testing Methodology

The terminal was tested by exercising two of the digital inputs, and by reading the states on two of the outputs. The times from input activation to message arrival, and from message transmission to output activation were taken and stored in a log file. The outputs were set up to produce one 2 second pulse, and the inputs were triggered by a similar pulse. The inputs and outputs were exercised by one host computer.

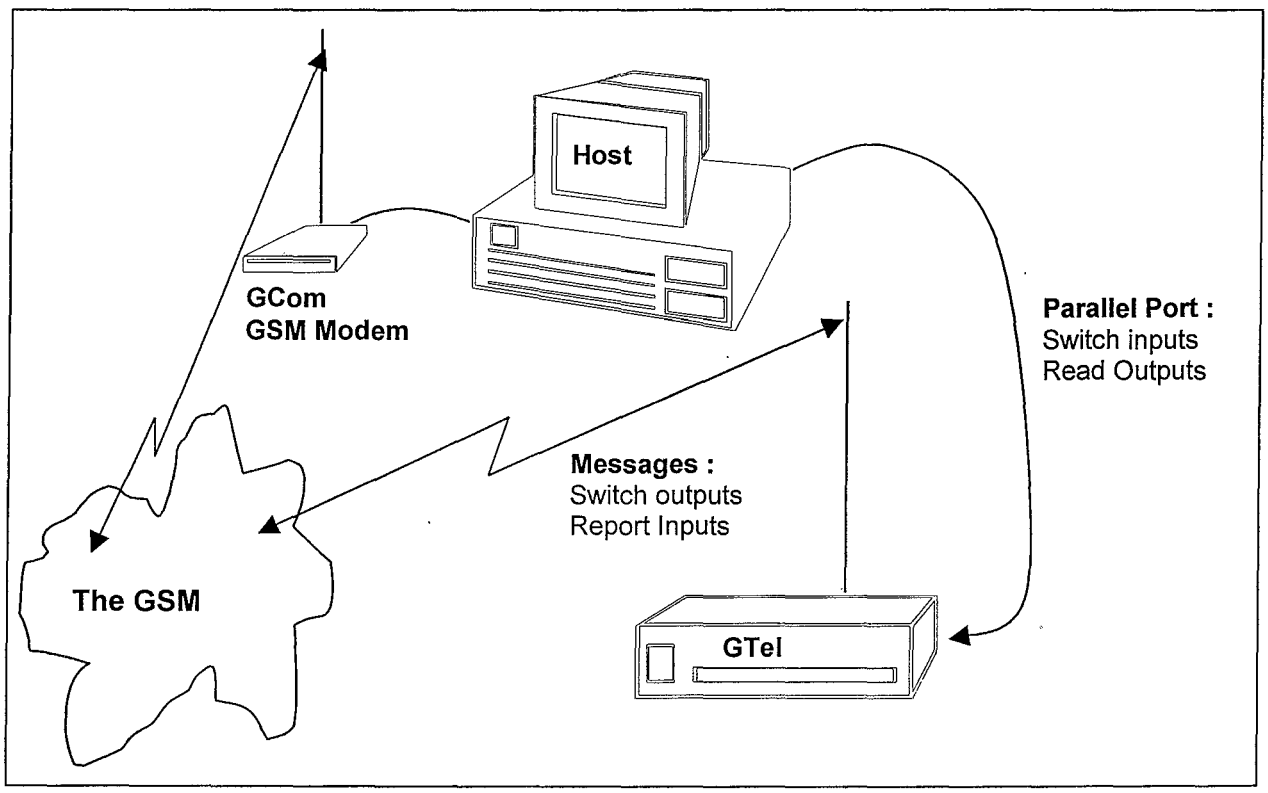


Figure 28. The test configutation.

The test consisted of 25 cycles of the following :

1) Send a message to GTel (the terminal) that will pulse output 1 high for 2 seconds.
2) Wait for the terminal output to go high. It is read via the parallel port of the host computer.
3) Measure the time taken from 1 to 2, and log it in a file on the host computer.
4) Send a message to GTel (the terminal) that will pulse output 2 high for 2 seconds.
5) Wait for the terminal output to go high. It is read via the parallel port of the host computer.
6) Measure the time taken from 4 to 5, and log it in a file on the host computer.
7) Force an input 1 trigger by pulsing the parallel port output high for 2 seconds.
8) Wait for the terminal to report the input event.
9) Measure the time taken from 7 to 8, and log it in a file on the host computer.
10) Force an input 2 trigger by pulsing the parallel port output high for 2 seconds.
11) Wait for the terminal to report the input event.
12) Measure the time taken from 10 to 11, and log it in a file on the host computer.

A one minute interval was taken between messages and triggers, making each cycle 4 minutes long.

37.2 Results

The results for the 100 message delivery times are as follows :

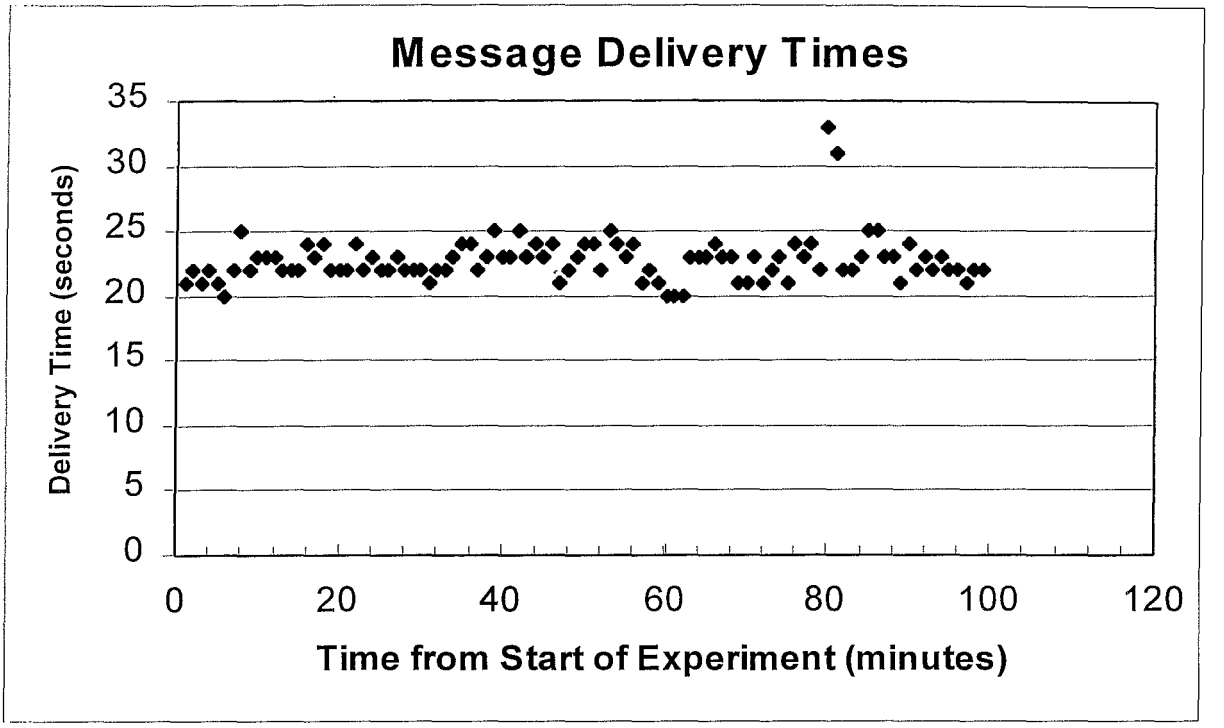


Figure 29. The delivery times of SMS messages over a 100 minute period.

It should be noted that these results are effected by the state of the program in the terminal at the time of message delivery or input trigger. In some data points this may skew the data into patterns. The delivery times in the period 55 minutes to 62 minutes show a downward trend, but this may be due to the terminal software.

In two cases (at 80 minutes), the delivery times are much higher than the others. This may be due to incoming message clashing with an outgoing one in the terminal or the modem.

All the messages were tested by the computer for data integrity, and were found to be without errors. The results of the test are presented in the form of a log file in Appendix B.

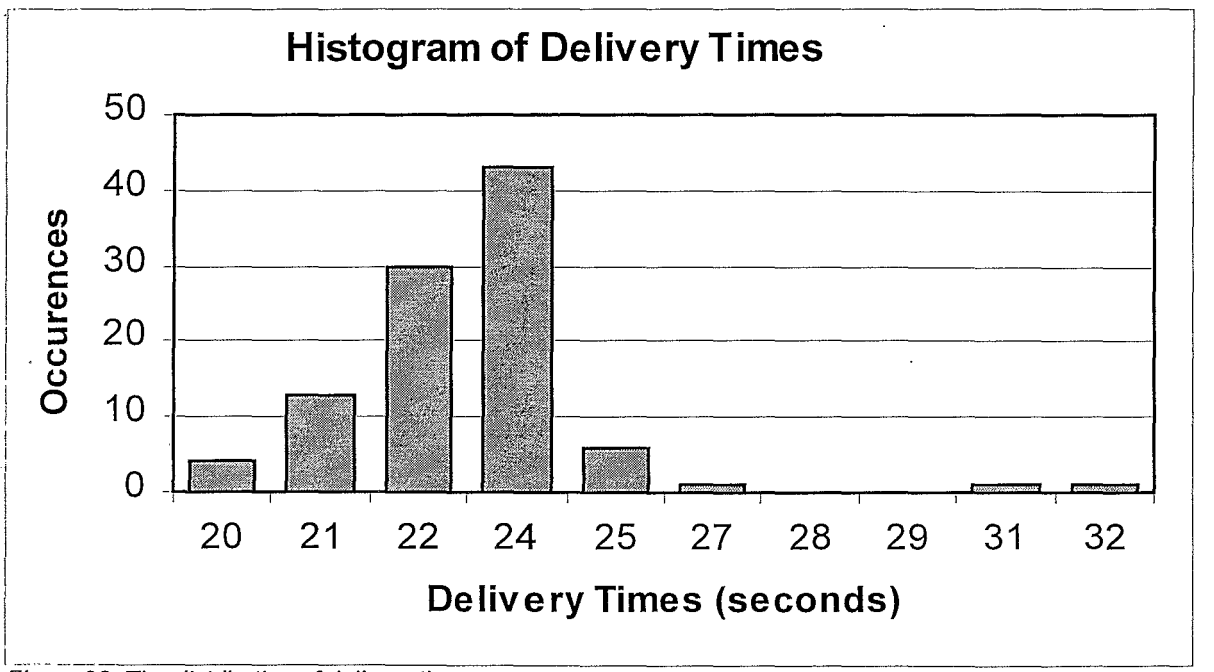


Figure 30. The distribution of delivery times.

38. Interface Accuracy

38.1 Testing Methodology

The interface was connected to test equipment to verify the accuracy of the parameters. The tested parameters were :

1) 3 Digital inputs
2) 1 Power sense input
3) 5 Analogue inputs (8 bit)
4) 1 Frequency counter (16 bit)
5) 4 Digital outputs

38.2 Results

Digital Inputs

The inputs were tested input logic levels, and were found to respond correctly to 0V/5V levels.

Power Sense

The power sensing input was tested with a debounce period of 20 seconds. The unit reported a power failure correctly.

Analogue Inputs

The five analogue inputs read full scale at 5V with a resolution of 19.5mV, but these inputs are protected by a voltage divider network with 470Ω series resistors, and 220kΩ pull-down resistors. This results in an error of around 1 LSB on a full scale reading. This is quite acceptable, but this error may be reduced by increasing the pull-down resistors to 1MΩ.

Frequency Counter Input

The frequency counter input showed an error of between 5Hz and 25Hz low. This error only became apparent over frequencies of 30kHz. This may be due to to interrupt latency, and the internal 1 second clock that is not a perfect divisor of the crystal used in the oscillator (11.0592MHz). Some adjustments are made for clock errors in the software, and this may be improved to yield better frequency counting results.

Outputs

The outputs were tested for output voltage with a 10mA load. The results were satisfactory with a 47mV voltage drop at this current for sinking and sourcing topologies because of the 4.7Ω series protection resistor.

39. Functionality

39.1 Testing Methodology and Results

The functionality was tested by exercising the terminal with the host software written for this purpose. Although the software did not test the functionality automatically, the author could test all the functions to verify that it works according to the design specifications.

Chapter 7 Conclusion

40. The Management and Implementation of a Global System for Mobile communications (GSM) based Data Terminal

40.1 Competing in Information Technology Product Markets

To compete in markets where innovation, time-to-market and volume sale are at crucial to the survival of a designing company, such companies have to take a hard look at the way in which they approach new design challenges. One no longer has the luxury of over-designing a product, or long design periods to achieve the goals. Novel ways must be sought to improve :

1) The design cycle.
2) The flexibility of the end product.
3) Efficient and complete testing during production.
4) The mass manufacturability of the product.
5) Effective component sourcing and usage.
6) The reliability of the product.
7) The cost of the product (including the development costs)

40.2 Management Strategy

The proposed management strategy was used with great success in the design cycle of the data terminal. The core aspects of the system were completed in under four months, and the terminal itself was ready for testing after three months.

By allowing the project development to follow the logical flow of the suggested strategy, many concurrent tasks could be completed without undue interference. It was easier to keep focus, as the tasks to be completed were all well-defined.

By completing the project quickly, a design house will be better positioned to offer more competitively priced products and to penetrate markets in less time.

The author is of the opinion that the management strategy offered will improve the design cycle of medium to large scale projects. Costs can be kept low without compromising the quality of the finished product; in fact the quality and reliability will improve with this structured approach.

40.3 The Data Terminal

The terminal was tested by exercising two of the digital inputs, and by reading the states on two of the outputs. The times from input activation to message arrival, and from message transmission to output activation were taken and stored in a log file. The results proved the design to be successful, and the product to be stable under high data activity. The terminal protocol is extremely flexible, and the user may choose to not receive or transmit redundant data. This could lead to significant cost savings when only one or two parameters are important to the user.

This flexibility led to very complex firmware, and an important point to consider is whether that code space couldn't be put to better use. By using fixed-length data records instead of the current variable length data records in the message payload, a significant amount of program memory can be regained.

With this extra program memory, a few very attractive options become available :

The GSM engine may be replaced or augmented by a satellite communicator

If the GSM network is unavailable, the unit would automatically start using the satellite communication channel.

The implementation of MODbus

MODbus is the de-facto standard for SCADA systems all over the world. By having such an interface, the terminal could be used in any of these SCADA systems. There are many third party developers who develop new and novel products around SCADA systems. For every successful project where they can use the terminal, there exists the potential for future sales.

The implementation of a proprietary protocol

Provisions were made on the terminal PCB to be able to accept daughter boards inside the enclosure. If a simple serial protocol could be developed that would protect third party developers from the complexity of sustaining a stable communications channel, they could a develop hundreds of new products like GSM-based security systems, vending systems, access control and energy control systems.

40.4 Conclusion

By using new technology and a structured project management system, the author was able to implement a very flexible, yet low cost data terminal in a short time.

Bibliography

ANALYSYS. 1998. E-Sat. [Available on Internet:] http://www.analysys.com/products/satellite/profiles/esat.htm [Date of use: 1 March 1999].

DOLPHIN TELECOM. 1999. TETRA : The strength behind Dolphin. [Available on Internet:] http://www.dolphin-telecom.co.uk/content/tetrpage.htm [Date of use: 1 March 1999].

ETSI (European Telecommunications Standards Institute). 1998. GTS GSM 07.05 V5.5.0. Sophia Antipolis : ETSI Secretariat. 68p.

ETSI (European Telecommunications Standards Institute). 1996. GTS GSM 03.40 V5.3.0. Sophia Antipolis : ETSI Secretariat. 109p.

ETSI (European Telecommunications Standards Institute). 1994. ETS 300 536 ed.4. Sophia Antipolis : ETSI Secretariat. 97p.

LEO ONE. 1998. Overview. [Available on Internet:] http://www.leoone.com/overview.html [Date of use: 1 March 1999].

MILLAR, H. 1996. Rockets for the Rest of Us. ***WIRED***, 4.09. [Available on Internet:] http://www.wired.com/wired/4.09/es.space.html [Date of use: 1 March 1999].

MOBILE SMS. 1999. Guide to SMS support on different networks. [Available on Internet:] http://mobilesms.com/guide.htm [Date of use: 1 March 1999].

National Academy of Sciences (United States of America). 1997. The Evolution of Untethered Communications. [Available on Internet:] http://nap.edu/readingroom/books/evolution/1.html [Date of use: 1 March 1999].

ORBCOMM. 1999. System description. [Available on Internet:] http://www.orbcomm.com/about/sysdesc.html [Date of use: 1 March 1999].

SCOURIAS, J. 1997. Overview of the Global System for Mobile Communications. [Available on Internet:] http://ccnga.uwaterloo.ca/~jscouria/GSM/gsmreport.html [Date of use: 1 March 1999].

THE AMERICAM RADIO RELAY LEAGUE. 1998. Band Threat: Who are the "Little LEO" Players? [Available on Internet:] http://www.arrl.org/fcc/wrc97/players.html [Date of use: 1 March 1999].

WAVECOM. 1999. AT Commands Interface. Paris: WAVECOM. 70p.

WAVECOM. 1999. Software Release Note for Wavecom GSM Module. Paris: WAVECOM. 11p.

WAVECOM. 1999. WISMO1a Requested Environment. Paris: WAVECOM. 6p.

WHITBY, WF III. 1999. Foreign Trunked Radio Systems. [Available on Internet:] http://members.aol.com/wwhitby2/for_trs.html [Date of use: 1 March 1999].

Appendix A - Electrical Specifications

1. Electrical Characteristics of the Data Terminal

1.1 Absolute Maximum Ratings

Ambient temperature under power ..... -20 to +55°C
Storage temperature ..... -25°C to +70°C
Voltage on any input line with respect to GROUND ..... -0.3V to (30V)
Voltage on VDD with respect to VSS ..... -0.3 to +7.5V
Maximum clamp current out of output pin ..... 250 mA
Maximum clamp current into output pin 250mA
Maximum output current sunk by any I/O pin ..... 25 mA
Maximum output current sourced by any I/O pin ..... 25 mA

1.2 DC Characteristics

	Parameter		Minimum	Typical	Maximum	Units
1	Supply Voltage	(AC) (DC)	9 9	-	25 32	V
2	Supply Current	@12VDC @24VDC	40 28	-	160 80	mA
3	Digital Input Voltage (Low)		0	-	0.8	V
4	Digital Input Voltage (High		2	-	30	V
5	Analogue Input Voltage (0)		0	-	0.020	V
6	Analogue Input Voltage (255)		5	-	5.02	V
7	Frequency Counter Input		0	-	30	V
7	Frequency Counter Input		1	-	65300	Hz
8	Output Voltage High (@ 10mA)		4	-	4.3	V
9	Output Voltage Low (@ -10mA)		0	-	0.7	V

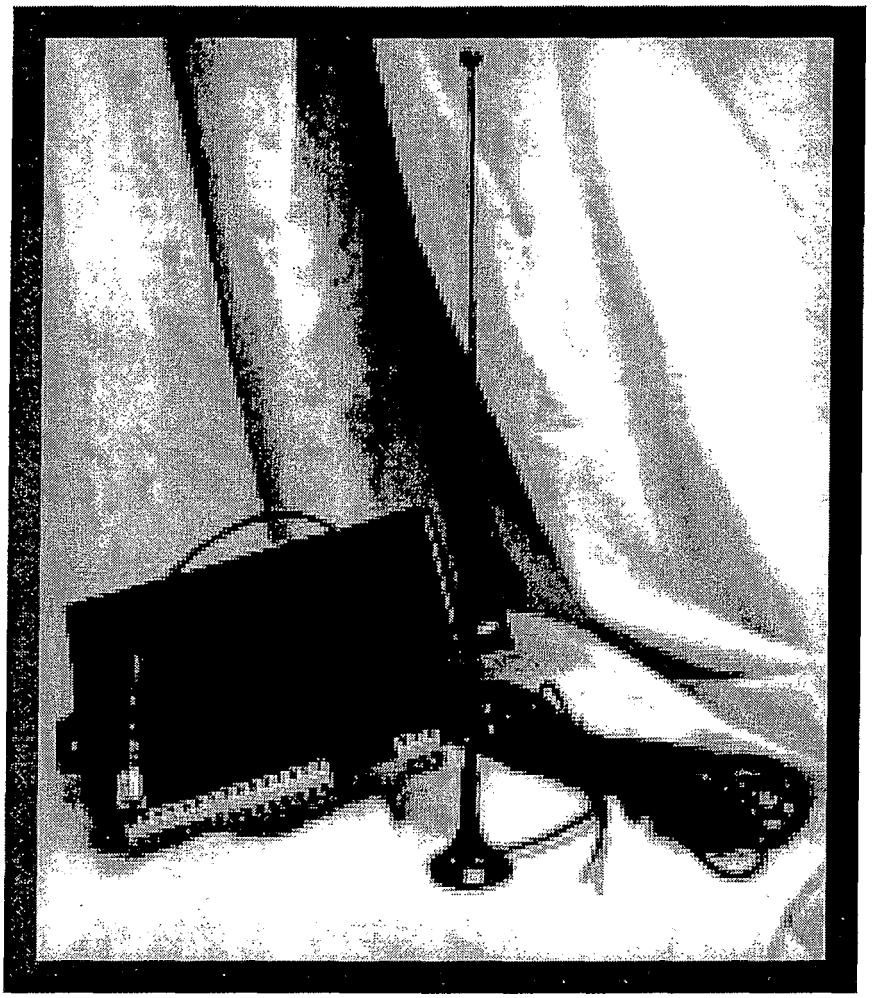
WOOD, L. 1999. Lloyd's satellite constellations. [Available on Internet:] http://www.ee.surrey.ac.za/Personal/L.Wood/constellations/overview.html [Date of use: 1 March 1999].

Appendix B Test Results - Log File

9:30 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 9:30:21 AM
9:31 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 9:31:22 AM
9:32 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 9:32:21 AM
9:33 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 9:33:22 AM
9:34 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 9:34:21 AM
9:35 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 9:35:20 AM
9:36 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 9:36:22 AM
9:37 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 9:37:23 AM
9:38 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 9:38:22 AM
9:39 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 9:39:23 AM
9:40 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 9:40:24 AM
9:41 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 9:41:24 AM
9:42 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 9:42:22 AM
9:43 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 9:43:22 AM
9:44 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 9:44:22 AM
9:45 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 9:45:25 AM
9:46 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 9:46:22 AM
9:47 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 9:47:23 AM
9:48 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 9:48:22 AM
9:49 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 9:49:23 AM
9:50 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 9:50:22 AM
9:51 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 9:51:25 AM
9:52 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 9:52:22 AM
9:53 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 9:53:23 AM
9:54 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 9:54:23 AM
9:55 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 9:55:21 AM
9:56 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 9:56:23 AM
9:57 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 9:57:22 AM
9:58 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 9:58:21 AM
9:59 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 9:59:24 AM
10:00 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:00:22 AM
10:01 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:01:22 AM
10:02 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:02:21 AM
10:03 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:03:24 AM
10:04 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:04:24 AM
10:05 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:05:23 AM
10:06 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:06:22 AM
10:07 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:07:24 AM
10:08 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:08:23 AM
10:09 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:09:22 AM
10:10 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:10:22 AM
10:11 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:11:25 AM
10:12 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:12:21 AM
10:13 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:13:23 AM
10:14 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:14:23 AM
10:15 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:15:24 AM
10:16 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:16:22 AM
10:17 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:17:25 AM
10:18 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:18:22 AM
10:19 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:19:22 AM
10:20 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:20:22 AM
10:21 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:21:22 AM
10:22 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:22:22 AM
10:23 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:23:23 AM
10:24 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:24:25 AM
10:25 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:25:24 AM
10:26 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:26:22 AM
10:27 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:27:22 AM
10:28 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:28:23 AM

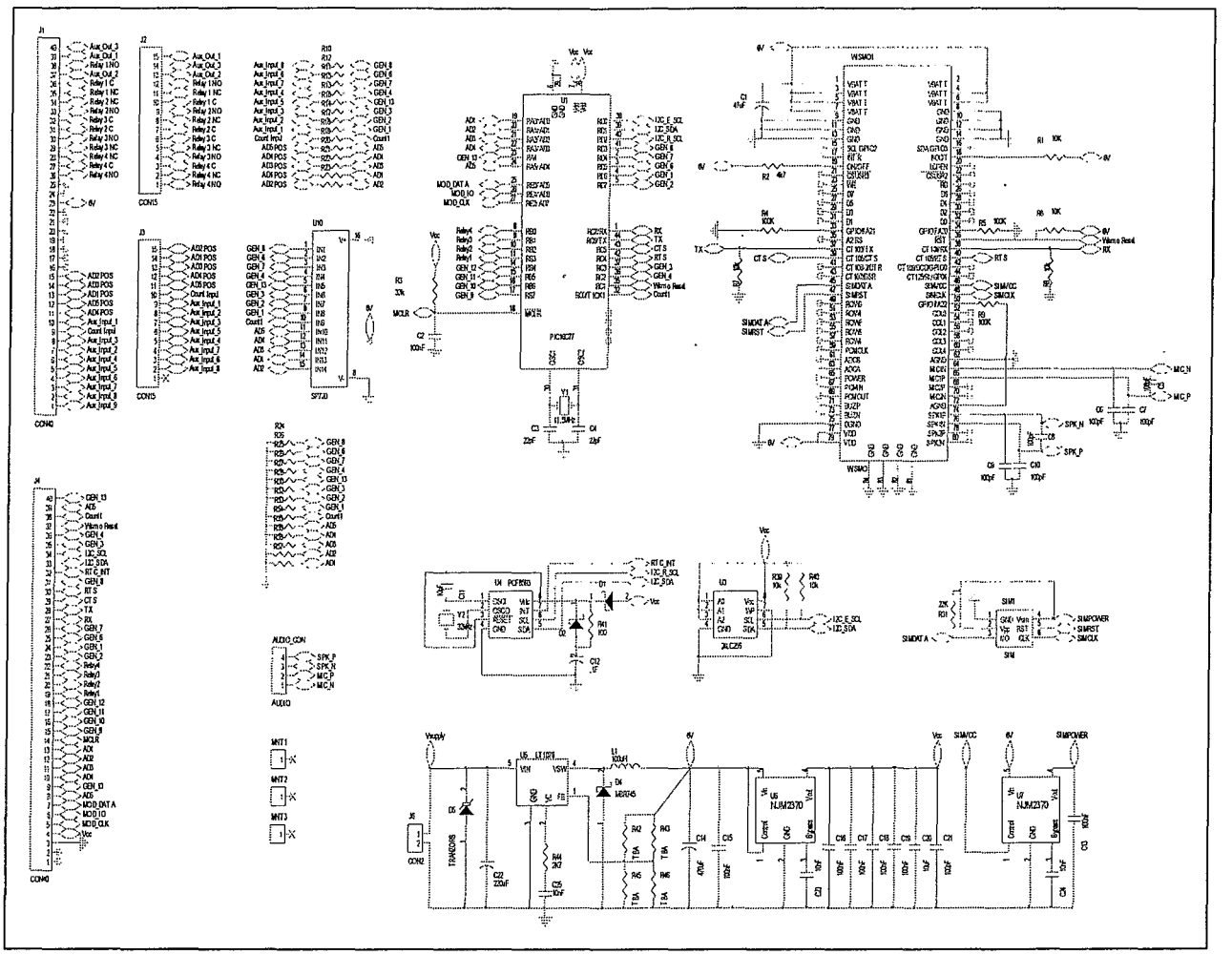
10:29 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:29:20 AM
10:30 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:30:20 AM
10:31 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:31:20 AM
10:32 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:32:23 AM
10:33 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:33:26 AM
10:34 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:34:23 AM
10:35 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:35:23 AM
10:36 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:36:24 AM
10:37 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:37:24 AM
10:38 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:38:21 AM
10:39 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:39:21 AM
10:40 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:40:22 AM
10:41 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:41:21 AM
10:42 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:42:24 AM
10:43 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:43:24 AM
10:44 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:44:21 AM
10:45 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:45:25 AM
10:46 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:46:25 AM
10:47 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:47:22 AM
10:48 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:48:21 AM
10:49 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:49:28 AM
10:50 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:50:30 AM
10:51 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:51:22 AM
10:52 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:52:22 AM
10:53 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:53:23 AM
10:54 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:54:24 AM
10:55 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:55:25 AM
10:56 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 10:56:23 AM
10:57 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 10:57:23 AM
10:58 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 10:58:22 AM
10:59 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 10:59:21 AM
11:00 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 11:00:23 AM
11:01 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 11:01:23 AM
11:02 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 11:02:23 AM
11:03 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 11:03:23 AM
11:04 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 11:04:23 AM
11:05 AM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 11:05:22 AM
11:06 AM Input 1 Triggered with pulse. Waiting for input 1 event message... ...message received at : 11:06:21 AM
11:07 AM Input 2 Triggered with pulse. Waiting for input 2 event message... ...message received at : 11:07:22 AM
11:08 AM Output 1 pulse command sent. Waiting... ...Pulse on output 1 detected at : 11:08:24 AM
12:08 PM Output 2 pulse command sent. Waiting... ...Pulse on output 2 detected at : 12:08:00 PM

Appendix C Photograph of Finished Product



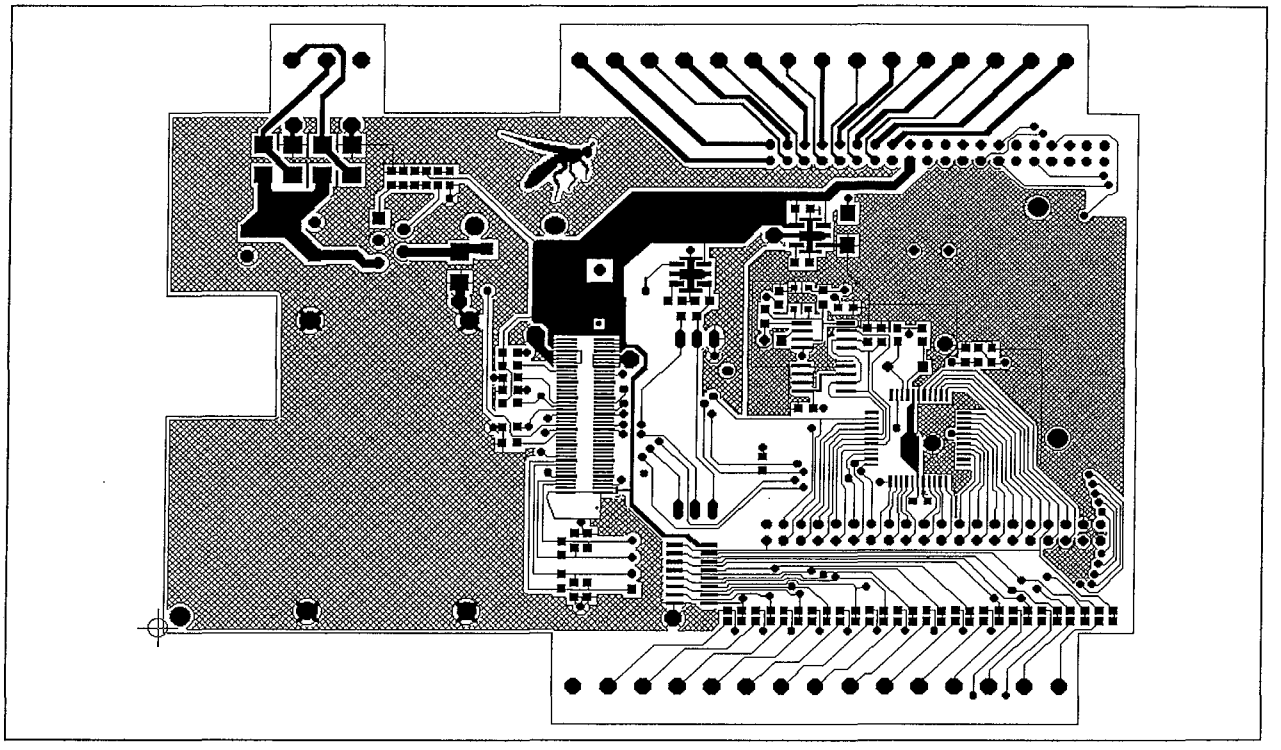
The completed data terminal with the 900MHz GSM antenna.

Appendix D Circuit Design - Schematic



Appendix E Printed Circuit Board Design

1. Top Layer



2. Bottom Layer