SASTRA UNIVERSITY

(A University under section 3 of UGC Act, 1956)

Thanjavur, Tamilnadu, India.

A Project on

REAL-TIME ACQUISITION, PROCESSING AND DISPLAY OF MIL-1553 TELEMETRY DATA

Project report submitted to

Shanmugha Arts, Science, Technology & Research Academy

SASTRA UNIVERISTY

In partial fulfillment of the requirements for the

Award of the Degree of B.Tech (Computer Science & Engineering)

In

SCHOOL OF COMPUTING

Under the Guidance of

Shri. ANEESH K. THAMPI,
Scientist/Engineer 'SC'
SITD/CSNG – IISU, ISRO

Dr. A. UMAMAKESWARI
Professor,
Department of CSE, SOC,
SASTRA University

Submitted by

CHITHROOBENI, S (011203053)
SUBESSWARE, SK (011203238)
VENKATA NARASIMHAN, A (011203257)

Department of COMPUTER SCIENCE

School of Computing

Shanmugha Arts, Science, Technology & Research Academy

SASTRA UNIVERSITY

JULY 2011 – MAY 2012

BONAFIDE CERTIFICATE

SCHOOL OF COMPUTING

SASTRA UNIVERSITY

This is to certify that the project work entitled **"Real-time acquisition, processing and display of MIL-1553 telemetry data",** is the bonafide work done by **CHITHROOBENI. S (011203053), SUBESSWARE. SK (011203238) and VENKATA NARASIMHAN. A (011203257),** Students of VIII Semester B.Tech Degree in COMPUTER SCIENCE during the academic year 2011-2012 in Partial Fulfillment of the requirement for the award of Degree of Bachelor of Technology in COMPUTER SCIENCE at SASTRA University.

______________________________ ______________________________

Project Guide **Dean, SOC**

Submitted for the University Exam held on ______________

______________________________ ______________________________

Internal Examiner **External Examiner**

Acknowledgements

We would like to thank the management, our honorable **Vice-Chancellor, Prof. R. Sethuraman, Dean – Planning & Development, Dr. S. Vaidhyasubramaniam,** and our esteemed **Registrar, Dr. S.N. Srivastava**, for giving us this opportunity to develop our knowledge and sharpen our technical skills.

We express our gratitude to **Dr. Mahendra Murali,** and **Dr. T.Sivamurugan** who provided us with the opportunity to work in a prestigious organization like ISRO (Indian Space Research Organization), and contribute a little to the space program of the nation.

We would like to convey our heartfelt thanks to our guide, **Mr. Aneesh K. Thampi** and **Mrs. Bindu Thomas** for spending their precious time and guiding us in the best possible manner, extending their invaluable support and help all through the project.

We are also grateful to the **Dean - School of Computing, Dr. P. Swaminathan,** and the **Associate Dean, Dr. Umamakeswari. A,** our internal guide, for lending us their priceless support in the completion of the project.

Last but not the least; we are thankful to the Almighty, our parents, and friends for the encouragement and moral support which they have been giving us in abundance.

Table of contents

List of figures

List of tables

1 Introduction

1. Introduction to the company

2. Introduction to the project
    1. Rockets and telemetry data
    2. INS and RESINS
    3. MIL STD1553 Bus
        1. Characteristics
        2. Hardware elements
        3. Protocol
        4. Message format
    4. Packages
        1. NIM
        2. NGCP
    5. Modes of transfer
        1. Monitor mode
        2. Flight mode
    6. Real time operating systems

3. Problem statement
    1. Vision
    2. Issue statement
    3. Method

1. INTRODUCTION TO THE COMPANY

The Indian Space Research Organization (ISRO) is the primary body of space research under the control of the Government of India. Realizing the immense potential of space technology for national development, Dr. Vikram Sarabhai, the father of India's space program, envisioned that this powerful technology could play a meaningful role in national development, and solving the problems of common man. As a result of coordinated efforts initiated earlier, ISRO was established in its modern form in 1969, with the objective to develop space technology and its application to various national tasks.

As stated by Dr. Vikram Sarabhai, *"There are some who question the relevance of space activities in a developing nation. To us, there is no ambiguity of purpose. We do not have the fantasy of competing with the economically advanced nations in the exploration of the moon or the planets or manned space-flight. But we are convinced that if we are to play a meaningful role nationally, and in the community of nations, we must be second to none in the application of advanced technologies to the real problems of man and society"*.

ISRO has successfully operationalised two major satellite systems, Indian National Satellite (INSAT) for communication services, and Indian Remote Sensing (IRS) satellites for management of natural resources; Polar Satellite Launch Vehicle (PSLV) for launching IRS type satellites, and Geostationary Satellite Launch Vehicle (GSLV) for launching INSAT type satellites. In 2008, ISRO successfully launched its first lunar probe, Chandrayaan-1, with future plans for further lunar exploration, and manned space missions among others.

ISRO Inertial Systems Unit (IISU) at Thiruvananthapuram is the centre of excellence in the area of inertial systems for launch vehicles and spacecrafts. IISU carries out research and development in the area of inertial sensors & systems and allied satellite elements. It has facilities for precision fabrication, assembly, clean room and integration and testing. This unit has the total capability to design, engineer, develop, qualify and deliver inertial systems for the entire Indian Space programme.

2. INTRODUCTION TO THE PROJECT

2.1 Rockets and telemetry data

In spaceflight, a launch vehicle or carrier rocket is a rocket used to carry a payload from the earth's surface into outer space. Usually the payload is an artificial satellite placed into orbit. Some spaceflights are sub-orbital while others enable spacecraft, along with the payload, to escape earth orbit entirely.

There are multiple processing units situated at different parts of the launch vehicle. They are used for critical operations like navigation, stage processing, sequencing, guidance and control of the vehicle. They are connected to a common bus. ISRO launch vehicles use Military Standard 1553 bus for this inter-processor communication. During flight time, 1553 bus data transactions are monitored and this data is transmitted to the ground stations through wireless link. It is called telemetry data and it contains functional information about the rocket. During launch, this data is analysed in ground stations to assess the mission performance of the rocket in real time.

2.2 INS and RESINS

An inertial navigation system (INS) is a navigation aid that uses a computer, motion sensors (accelerometers) and rotation sensors (gyroscopes) to continuously calculate via dead reckoning the position, orientation, and velocity (direction and speed of movement) of a moving object without the need for external references. It is used on vehicles such as ships, aircraft, submarines, guided missiles, and spacecraft. In ISRO REdundant Strapdown Inertial Navigation System (RESINS) is used.

Though the underlying principle behind INS and RESINS are the same, the two differ by one aspect. In normal Inertial systems the sensors are attached to a board and are manually isolated from the rotational motion of the vehicle. This is used for very sensitive calculations but is costly. Now in RESINS – Strapdown navigation systems the sensors are "strapped down" or attached rigidly to the rocket. The components are attached to a bus that runs throughout the length of the rocket. This facilitates small, lightweight and cheap inertial systems. **But this increases the complexity of the computers that are used to monitor the trajectory.**

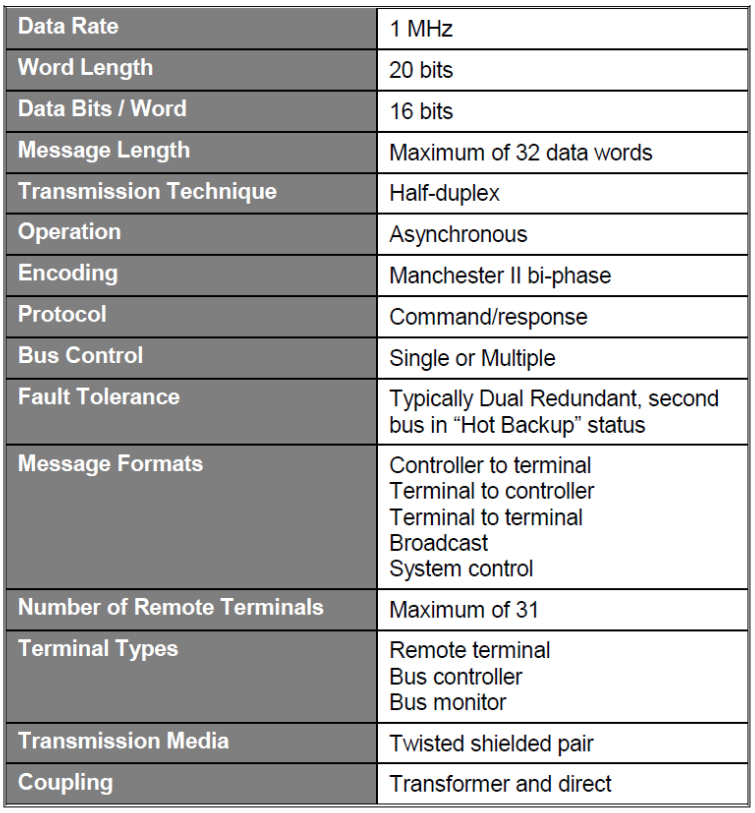
Here a redundant system is used. Redundancy is used for reducing error. The bus that connects all the on board components is duplicated, giving one prime, and one redundant bus. All the INS hardware on board is also duplicated as prime and redundant to achieve a higher degree of reliability and fault minimisation. This is the bus standard used is MIL-STD-1553. *The details of the bus are discussed next.*

2.3 MIL STD 1553 Bus

MIL-STD-1553 is a military standard published by the United States Department of Defence that defines the mechanical, electrical, and functional characteristics of a serial data bus. It was originally designed for use with military avionics, but has also become commonly used in spacecraft on-board data handling (OBDH) subsystems, both military and civil. It features a dual redundant balanced line physical layer, a (differential) network interface, time division multiplexing, half-duplex command/response protocol, and up to 31 remote terminals (devices). While the standard has been applied to satellites as well as payloads within the space shuttle, its military and avionics applications are the most numerous and far ranging.

2.3.1 Characteristics

A summary of the basic characteristics of MIL-STD-1553 is as follows.



Data Rate	1 MHz
Word Length	20 bits
Data Bits / Word	16 bits
Message Length	Maximum of 32 data words
Transmission Technique	Half-duplex
Operation	Asynchronous
Encoding	Manchester II bi-phase
Protocol	Command/response
Bus Control	Single or Multiple
Fault Tolerance	Typically Dual Redundant, second bus in “Hot Backup” status
Message Formats	Controller to terminal Terminal to controller Terminal to terminal Broadcast System control
Number of Remote Terminals	Maximum of 31
Terminal Types	Remote terminal Bus controller Bus monitor
Transmission Media	Twisted shielded pair
Coupling	Transformer and direct

2.3.2 Hardware elements

Transmission Media

The transmission media is basically the data bus. It is made of a twisted shielded pair transmission cable. It consists if the main bus and stubs.

The components that are connected to the bus can be of the following roles

1. **Bus Controller (BC)**

   The bus controller is responsible for directing the flow of data on the data bus. It is the only one allowed to issue commands onto the data bus. It initiates all the message communication over the bus. There is only one BC at any time on a MIL-STD-1553 bus. The Bus Controller,

   - Operates according to a command list stored in its local memory
   - Commands the various Remote Terminals to send or receive messages
   - Services any requests that it receives from the Remote Terminals
   - Detects and recovers from errors
   - Keeps a history of errors

2. **Bus Monitor**
   The bus monitor is a terminal that listens (monitors) to the exchange of information on the data bus. It may collect all the data from the bus or may collect selected data.
3. **Remote Terminals (RT)**
   Remote terminals are all terminals not operating as the bus controller or as a bus monitor. It comprises the electronics necessary to transfer data between the data bus and the subsystem. It consists of a transceiver, an encoder/decoder, a protocol controller, a buffer or memory, and a subsystem interface. The subsystem interface may consist of the buffers and logics necessary to interface to the computer's address, data, and control buses
   Functions of the remote terminal –
   a) Data formatting
   b) Receiving and decoding commands from the bus controller and responding accordingly
   c) Buffering a message worth of data, detecting transmission errors and performing validation tests upon the data
   d) Reporting the status of the message transfer only respond to commands received from the bus controller

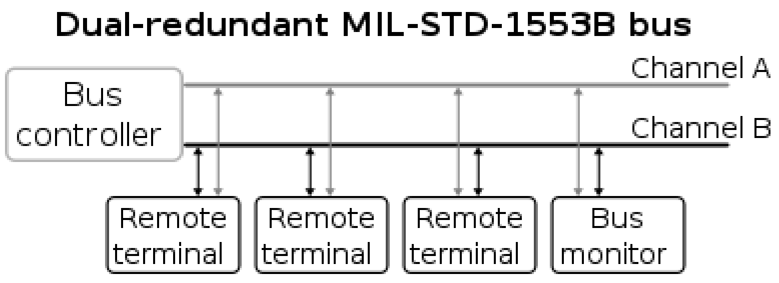


Fig 2.3.2.1 Bus model

2.3.3 Protocols

Protocols are rules under which the information transfer occurs. The control, data flow, status reporting, and management of the bus are provided by three word types.

Word types

The data is sent through the bus in the form of 20 bit words. There are three word types defined - command, data, status. Each word is twenty bits in length. The first three bits are used as a synchronization field. The next sixteen bits are the information field and the difference between the three word types. The last bit is the parity bit. Parity is based on odd parity for the single word. Bit encoding for all words is based on bi-phase Manchester II format.

1. **Sync Fields** consists of the first 3 bits of all the words. It is not a valid Manchester waveform. It is used to allow the decoder to re-sync at the beginning of each word received and maintain overall **stability** of the transmissions.
   a) Command/Status Sync - positive voltage level for the first one and a half bit times, then transitions to a negative voltage level for the second one and a half bit times
   b) Data Sync – opposite
2. **Command Word** specifies the function that the remote terminal is to perform
   a) Sync(3 bits) – Command Sync
   b) Terminal Address(5 bits) – To which remote terminal command is intended

c) Transmit/Receive bit(1 bit) – Defines direction of information flow
   1 - Transmit data
   0 – Receive data
d) Sub address(5 bits) – Direct the data to different functions within the sub-system
e) Parity(1 bit) – Odd parity

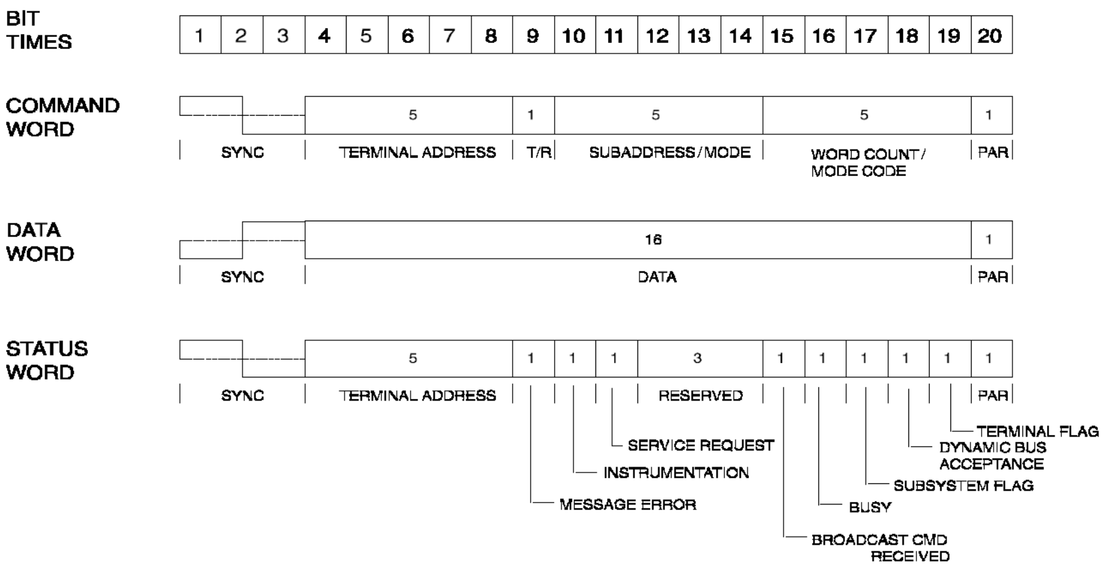


Fig 2.3.3.1 Word format

3. **Data Word** contains the actual information that is being transferred within a message
   a) Sync(3 bits) – Data Sync
   b) Data (16 bits) – Defined by designer. MSB of data transmitted first
   c) Parity(1 bit) – Odd parity
4. **Status Word** is used to convey to the bus controller whether a message was properly received or to convey the state of the remote terminal (i.e., service request, busy, etc.)

Message format

The messages sent through the bus are a set of words sent in a specific order, depending on the type of message. There are three types of messages that can be sent through the bus – BC to RT, RT to BC, and RT to RT. The message formats for all three types are as shown below.

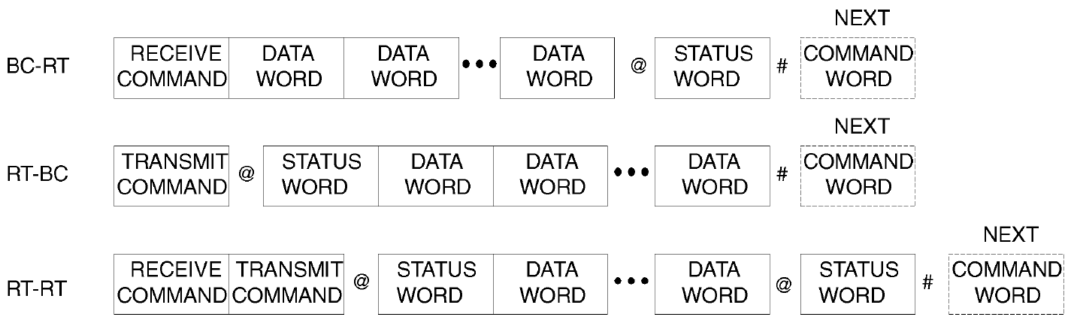


Fig 2.3.3.2 Message format

2.4 Packages

Package is the term used to represent a component connected to the MIL 1553 STD bus. The packages that we are most concerned about are the following

- **NIM**
  Navigation Interface Module (NIM) is connected to the MIL-1553 bus. It communicates with other packages through the bus. The NIM design is based on the 8031 micro-controller. It has a 16 bit address bus, and an 8 bit data bus. NIM acquires the digital and analog channel data from the Inertial Navigation System (INS) and sends the collective data to the bus controller.
  In Redundant INS, NIM package is replicated as NIM-P (Prime), and NIM-R (Redundant). NIM-P and NIM-R are connected to both the buses. NIM-P is connected through a paired connection, while NIM-R is connected using a cross-strapped connection.

- **NGCP**
  Navigation Guidance and Control Processors (NGCP) is also connected to the MIL – 1553 bus. This acts as the controller as soon as the rocket is launched.
  The roles they play in the telemetry are discussed in the next section.

2.5 Modes of transmission

There are two modes of operation of onboard processors – Monitor mode and Flight mode.

- **Monitor mode** - All onboard processors are in this mode when the rocket is on ground and ready for launch. In this the rocket is in direct contact with the ground station. Ground checkout system will act as the Bus Controller and packages responds to commands from checkout. Checkout system will be doing the surveillance of all onboard system's health and functional outputs though 1553 bus. Onboard flight parameter initialization is also done monitor mode.
- **Flight mode** - All onboard processors switched from the monitor mode to flight mode a few seconds before launch. In flight mode a sequence of predefined commands are send to every processors every 20 millisecond. All RT's gets synchronized to Bus Controller timing.

The two modes are discussed in detail below. The monitor mode and flight mode protocols implemented in all other packages are the same. The protocols are explained below using NIM.

2.5.1 Monitor mode

In the monitor mode operations, a ground station computer is the Bus Controller (BC), and NIM is the Remote Terminal (RT). Monitor Mode operations are used for system health checks and for initialization of parameters to take NIM to the flight mode. The ground station computer, which acts as an interface to the onboard system (RESIN) for the users, is called a checkout system.

The communication with the NIM in the monitor mode is through three commands - LOAD (Load to memory), READ (Read from memory), and GO (Execution from specifies memory). These commands are to be issued by the checkout.

2.5.2 Flight mode

In the flight mode, the package sends data with a periodicity of 20ms on command from the NGCP. The data sent every 20ms is encapsulated in the telemetry format, and is called telemetry data. In the flight mode operations, the NGCP is the Bus Controller (BC), and the NIM is the Remote Terminal (RT). Flight mode operations are used for system calibration and evaluation. The system is required to be in flight mode for navigation functions to be carried out. For NIM to be taken to flight mode, some of the memory locations need to be initialized with correct data.

In the flight mode, NIM expects three types of commands – Rx-1 word (BC-RT synchronization message), Tx-28 words, and Tx-10 words, in specific time windows and order in a 20ms cycle from the paired and cross-strapped BCs.

NIM's telemetry format

The data transmission from NIM to NGCP follows the telemetry format. As per the telemetry format, two messages are transmitted in each cycle. Message 1 consists of 28 words, and Message 2 consists of 10 words. Both the messages together constitute a minor. A set of 25 minors constitute a major. Out of the 28 words of Message 1, words 25, 26, and 27 are used for major cycle. These parameters repeat themselves only once in a complete major cycle (25 minors). Thus all the data from these 3 words of all 25 minors represent the major data. All the other words constitute the minor data and repeat themselves in every minor cycle. The following tables indicate what each word represents in the telemetry format, in case of NIM.

Word position	Data Transmitted
1	NIM Status Word
2	DTG X Digital Output
3	DTG X1 Digital Output
4	DTG Y Digital Output
5	DTG Y1 Digital Output
6	DTG Z Digital Output
7	DTG Z1 Digital Output
8	Accelerometer XA Digital Output
9	Accelerometer YA Digital Output
10	Accelerometer ZA1 Digital Output
11	Accelerometer ZA2 Digital Output
12	Accelerometer ZA3 Digital Output
13	Analog Channel Output of ZA1-median
14	Analog Channel Output of ZA2-median
15	Analog Channel Output of ZA3-median
16	Analog Rate X-median
17	Analog Rate X1-median
18	Analog Rate Y-median
19	Analog Rate Y1-median
20	Analog Rate Z-median
21	RGP Rate Z-median
22	RGP Rate Y-median
23	RGP Rate X-median
24	Minor Cycle ID (LSB only)
25	Major Cycle Data
26	Major Cycle Data
27	Major Cycle Data
28	Checksum (XOR of words from 1 to 27)

Table 2.5.1 Message 1 – 28 words

Word Position	Data Transmitted
1	Gyro Demodulator Output X-median
2	Gyro Demodulator Output X1-median
3	Gyro Demodulator Output Y-median
4	Gyro Demodulator Output Y1-median
5	Gyro Demodulator Output Z-median
6	Gyro Demodulator Output Z1-median
7	Analog Output of XA accelerometer-median
8	Analog Output of YA accelerometer-median
9	Analog Output Rate Z1-median
10	Checksum (XOR of words from 1 to 9)

Table 2.5.2 Message 2 – 10 words

Minor Cycle ID	Word 25	Word 26	Word 27
1	Wheel Speed X	Wheel Speed Y	Wheel Speed Z
2	RTC (MS word)	RTC (LS word)	Digital Channel Self Test 1
3	Digital Channel Self Test 2	X_ANA_M1	Y_ANA_M1
4	Z_ANA_M1	X1_ANA_M1	Y1_ANA_M1
5	Z1_ANA_M1	X_ANA_M2	Y_ANA_M2
6	Z_ANA_M2	X1_ANA_M2	Y1_ANA_M2
7	Z1_ANA_M2	RGP_X_M1	RGP_Y_M1
8	RGP_Z_M1	RGP_X_M2	RGP_Y_M2
9	RGP_Z_M2	Clu 1 temp	Clu 2 temp
10	VFC_X temp	VFC_Y temp	VFC_Z temp
11	ISUPM_X_5V	ISUPM_Y_5V	ISUPM_Z_5V
12	NIM_5V	Wheel Current X	Wheel Current Y
13	Wheel Current X	Pick off X	Pick off Y
14	Pick off Z	SPARE 1	Self Test 1-median
15	Self Test 2-median	Self Test 3-median	SYNC_CNT
16	TIMEOUT_CML	CORR_CML	SKEW_DIR (MSB) AUTOMODE_CML (LSB)
17	NIM_ER_CNT	PU	PL
18	XU	XL	Spare
19	Spare	Spare	Spare
20	Spare	Spare	Spare
21	Spare	Spare	Spare
22	Spare	Spare	Spare
23	Spare	Spare	Spare
24	Spare	Spare	Spare
25	Spare	Spare	Spare

Table 2.5.3 Major cycle data

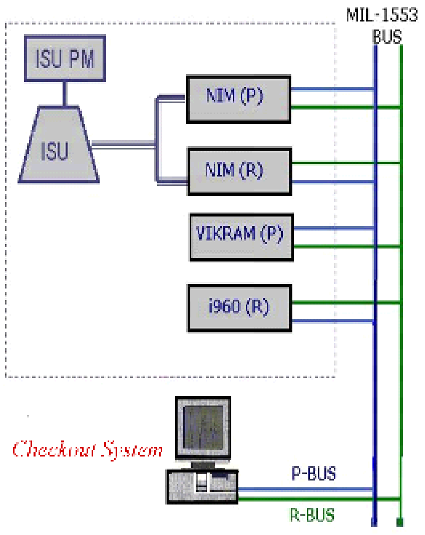


Fig 2.5.2.1 Packages connected –
VIKRAM(indigenous package developed by ISRO) and i960 are NGCPs.

2.6 Real time operating system

A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application requests. A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task. Most time critical applications are implemented on real time operating systems.

Telemetry data ranks high on time criticality as it is data obtained in the order of a few milliseconds (a sequence of messages every 20ms). The data got corresponds to the state of the sensors at that instant of time. Data miss, or a delay in data acquisition may prove very fatal. Thus there is need for the data to be read, processed and analyzed in real time, owing to which a RTOS is needed for this application.

A RTOS has an advanced algorithm for scheduling. Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency; a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

3. PROBLEM STATEMENT

3.1 Vision

To monitor required bus data efficiently, and remotely, in real time.

3.2 Issue statement

An application that monitors required bus data and represents them using visual display methods such as tables and graphs in real time, through a Graphical User Interface.

3.3 Method

The QNX Operating System has been used for efficient data acquisition in real time. The QNX Momentics IDE has been used to develop the application. The GUI has been developed using the Photon Application Builder. The application has been developed in C, using a few system specific, and POSIX calls.

Software project plan

1. Purpose and Scope
    1. Purpose of project
    2. Scope
        1. Existing system
        2. Proposed system
        3. Benefits

2. Architecture
    1. QNX operating system
    2. System overview

1. PURPOSE AND SCOPE

1.1 Purpose of project

To monitor the MIL-1553 data bus remotely, acquire the required data in real time, and display the acquired data using visual aids such as tables, and graphs.

1.2 Scope of the project

1.2.1 Existing System

The Checkout System is used at present to monitor the bus data at real time. It is an Interface to the onboard system for users. The Checkout system implements interfaces to packages like the NIM, NGCP, etc. It also implements and automates test requirements apart from providing an easy to use interface for users.

1.2.2 Proposed System

The proposed system does a subset of operations that a checkout does, on limited hardware and software resource. The system is a server-client system.

- Server keeps monitoring the data from the MIL STD 1553 bus on real-time.
- Client requests the data needed and the server returns the requested data. The display is clear, concise and easy to understand, thus making it easy to analyze the obtained data. The user interacts with the system using a user friendly Graphical User Interface, and the data requested is presented to the user in the form of different visual displays like tables and graphs, with alerts for values out of range etc, that help in health surveillance and the overall functionality check.
- More than one client can attach itself remotely to the server.

1.2.3 Benefits

Checkout is a highly complex, highly powerful computing system which has its own hardware and software. That much of computing power is not needed for monitoring the bus. It is not portable too.
The proposed system is a software that

- runs on desktops
- needs no extra hardware or software
- can be used extensively from any desktop on the network

2. ARCHITECTURE

2.1 QNX Neutrino operating system

QNX real-time operating system is chosen for implementation of the software. The real-time capabilities, availability of API, compilers and debugging tools are the factors that led to the choice. Some details of QNX are given in the next section since they are of relevance to the discussion on checkout software architecture.

QNX is a real-time, multi-tasking operating system with micro-kernel architecture. The kernel is in charge of a group of co-operating processes that implement the different functions of the operating system. The kernel implements scheduling algorithm and message-passing among the existing processes. All other QNX services are handled by a number of standard QNX processes like

- Process Manager (Proc)
- File system Manager (Fsys)
- Device Manager (Dev)
- Network Manager (Net)

User-written processes can be run extending the functionality of the system. Inter-process communication mechanisms include

- Synchronous messages using Send() - Receive() - Reply() functions
- Pulse for event notification
- Signals for asynchronous communication

Message passing extends to all processes in the network. When a process is waiting for a part of the message protocol to end, the process is said to be blocked. (eg. SEND-blocked, REPLY-blocked, RECEIVE-blocked).Each node in a QNX network has a unique identity. Processes and resources can have unique network-wide identity. There is network-wide access of resources, providing transparency with respect to location.

Pre-emptive priority based scheduling is provided with different scheduling mechanisms for same-priority processes like

- FIFO scheduling
- Round-robin scheduling
- Adaptive scheduling

The real-time performance is achieved by low interrupt latency, low scheduling latency and priority-based pre-emption of interrupts and processes.

The QNX kernel contains only CPU scheduling, inter-process communication, interrupt redirection and timers. Everything else runs as a user process, including a special process known as proc which performs process creation, and memory management by operating in conjunction with the microkernel. This is made possible by two key mechanisms - subroutine-call type inter-process communication, and a boot loader which can load an image containing not only the kernel but any desired collection of user programs and shared libraries. There are no device drivers in the kernel.

QNX inter-process communication consists of sending a message from one process to another and waiting for a reply. This is a single operation, called MsgSend. The message is copied, by the kernel, from the address space of the sending process to that of the receiving process. If the receiving process is waiting for the message, control of the CPU is transferred at the same time, without a pass through the CPU scheduler. Thus, sending a message to another process and waiting for a reply does not result in "losing one's turn" for the CPU. This tight integration between message passing and CPU scheduling is one of the key mechanisms that makes QNX message passing broadly usable.

Due to the microkernel architecture QNX is also a distributed operating system. All I/O operations, file system operations, and network operations were meant to work through this mechanism, and the data transferred was copied during message passing. Later versions of QNX reduce the number of separate processes and integrate the network stack and other function blocks into single applications for performance reasons.

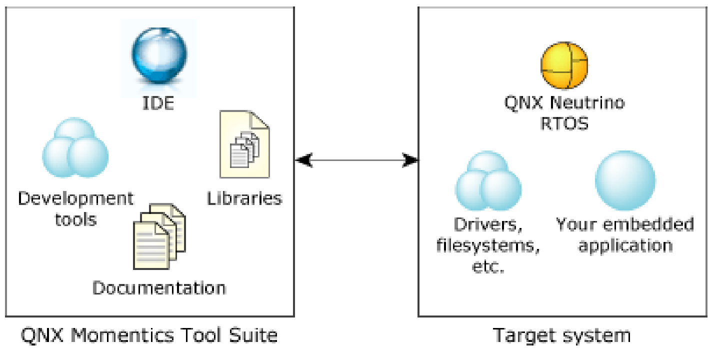
Message handling is prioritized by thread priority. Since I/O requests are performed using message passing, high priority threads receive I/O service before low priority threads, an essential feature in a hard real-time system.

The boot loader, although seldom discussed, is the other key component of the minimal microkernel system. Because user programs can be built into the boot image, the set of device drivers and support libraries needed for start-up need not be, and are not, in the kernel. Even such functions as program loading are not in the kernel, but instead are in shared user-space libraries loaded as part of the boot image. It is possible to put an entire boot image into ROM, which is used for diskless embedded systems. Neutrino supports symmetric multiprocessing and processor affinity, called bound multiprocessing (BMP) in QNX terminology. BMP is used to improve cache hitting and to ease the migration of non-SMP safe applications to multi-processor computers.

Neutrino supports strict priority-pre-emptive scheduling and adaptive partition scheduling (APS). APS guarantees minimum CPU percentages to selected groups of threads, even though others may have higher priority. The adaptive partition scheduler is still strictly priority-pre-emptive when the system is under loaded. It can also be configured to run a selected set of critical threads strictly real-time, even when the system is overloaded.

QNX Momentics

The development host runs the QNX Momentics Tool Suite; the target system runs the QNX Neutrino RTOS itself plus all the programs you develop:



The main parts of the QNX Momentics Tool Suite are as follows:

1. Integrated Development Environment - Helps to quickly set up the project, choose a programming language, choose a target processor, compile the code, connect to the target, transfer application to the target, run it, debug it, profile it, and fine-tune it.
2. Command-Line Tools – Used to develop applications when not using IDE.
3. Libraries - ANSI C, POSIX, Dinkum C++ (full and embedded), GNU C++ (x86 only), graphics, widgets, compression, etc.
4. Documentation - How-to guides, references, context-sensitive help, and technotes.

The development of an application with Momentics can be started even before the target hardware is known. The application code can be written once and compiled multiple times for multiple platforms, making it platform independent software.

Photon Application Builder (PhAB)

It is a visual design tool that generates the underlying C/C++ code to implement the application's user interface. PhAB modules can be accessed and created with own code. Databases of widgets can be set up and reused multiple times. Code functions can be attached to widget callbacks to implement the main functionality of the application.

Widget - PhAB provides a set of standard components called widgets that include sliders, lists, menus, buttons etc, that help ensure that the code need not be generated from the scratch. The widget set is built on an Object Oriented Framework loosely based on the X Toolkit Intrinsics Library(Xt). The widget combines data and operations required to implement a particular UI element (encapsulation). A widget encapsulates the knowledge of how to:

- draw itself
- respond to user events
- repair itself by redrawing when it is damaged

Containers are widgets that hold other widgets and manage their layout.
A widget hides the details of how it performs its responsibilities (information hiding). Thus the widget's internal implementation is separated from its public interface.

The public interface consists of all the attributes visible to other objects as well as the operations other objects may perform on the widget. The attributes in the widget's public interface are called *resources*.

2.2 System overview

The software is designed in such a way that it comprises of several logical units. Each logical unit is concerned with implementing a specific functionality and is independent of other units. The logical units communicate with one another in order to realize the total checkout functionality. The implementation details of a logical unit are entirely private to it. The logical units thus can be implemented and tested separately and this architecture has the advantages of easy extensibility, better maintainability and leads to easy implementation in an object-oriented language.

The different logical units are implemented as client and server programs. Typically a logical unit implementing device-driver functionality is a server program. The server exposes an interface through which its clients make use of its services. Thus, the software has client-server architecture. The adaptability and extensibility of the client-server architecture and the fact that it leads to a seamless extension of the underlying QNX architecture are the major factors that influenced this choice.

The clients and servers communicate through synchronous and asynchronous messages. The servers typically undertake initialization sequences and wait in a receive-blocked state for a request from clients. Once a request is received, the necessary action is taken and the results of the operation are sent back to the client, who will be waiting for the reply. The services that the server provides are well defined. This is the user-interface supported by it. There is a client program, called real time data monitor which acts as client to all the server programs and exposes a user-interface which is used by the user of the system. This client program controls the sequence of operations while the details of any particular operation are handled by the servers, offering specific services. There can be more than one client that connects themselves to the server. The connection is remote across the global namespace.

The co-operating threads are created by ‘pthread_create ()’ system call. Processes are identified using QNX naming convention. A process with global naming can be accessed through the network while processes with local names are visible only in that system. Messages to a process are sent using the process-identity (proc-id) number. The QNX operating system manages the proc-id when a process is created or deleted.

The different logical units are explained in more detail as use-cases in the following sections. The software has been designed with a Graphical User Interface built using QNX-Photon application builder.

In our system the server uses named server to create a channel of communication. The clients use this server to attach themselves to the server. There are separate data structures for each request and the corresponding reply from the server. The IPC messaging standard is used in the whole system. A pipe is used as buffer of communication between the threads of the same system. The server uses the object of Package class to keep track of the packages that need to be controlled monitored.

Software requirement specification

1. Functional requirements
    1. Use case
    2. Flow chart
    3. Sequence diagram
    4. Class diagram

2. Non functional requirements
    1. Performance requirements
    2. Resource requirements
        1. Hardware requirements
        2. Software requirements
    3. Quality and reliability requirements

1. FUNCTIONAL REQUIREMENTS

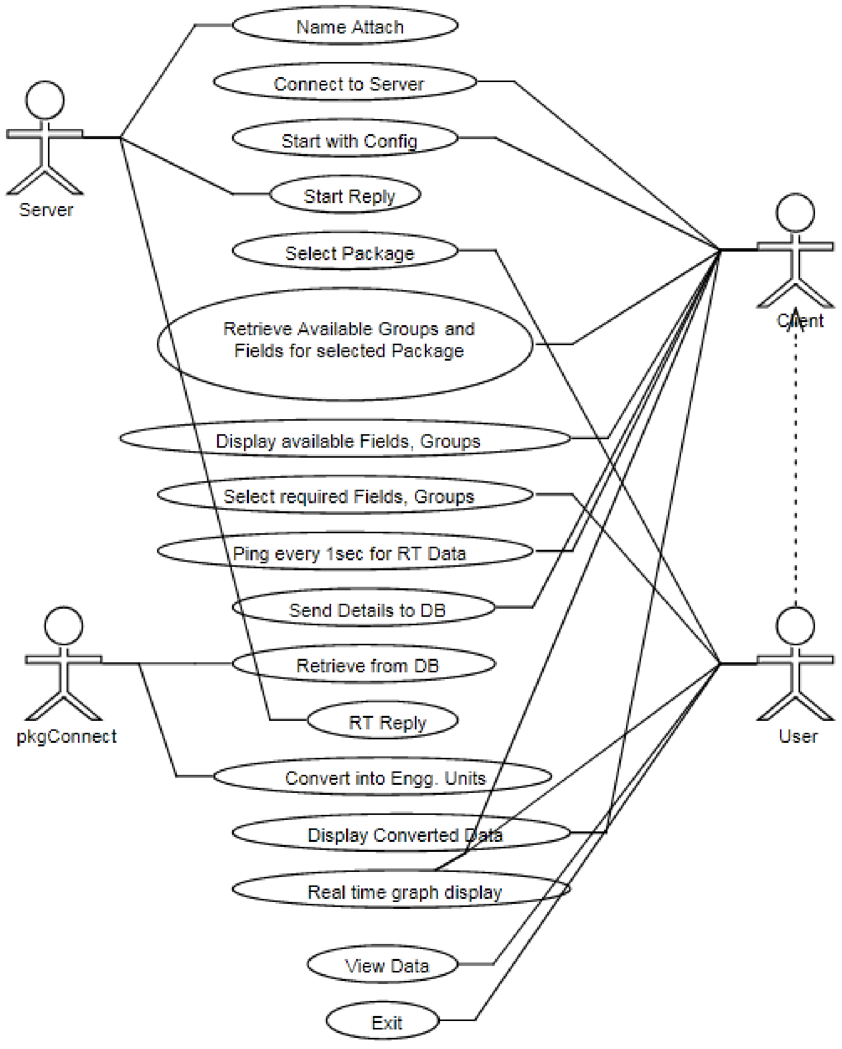
1.1 Use case diagrams

The main purpose of a use case diagram is to show what system functions are performed for which actors. Roles of the actors in the system can be depicted.

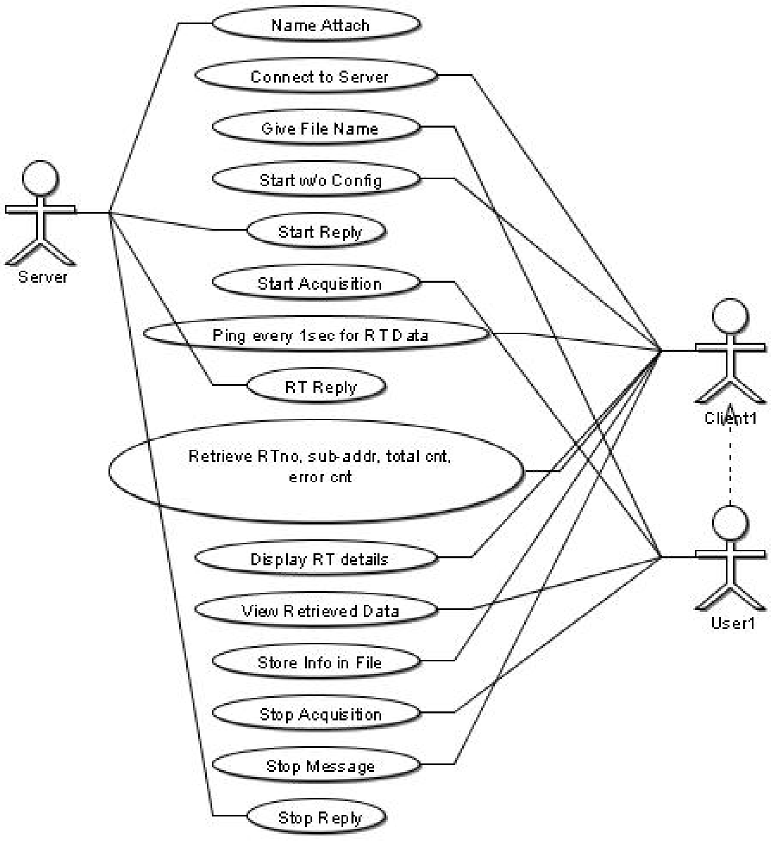
The actors used in the use case diagrams are

- Server
- Client - The use case defines the interaction of a single client. There can be more than one client co-existing.
- User – One for each client. User represents the human activity
- Pgkconnect – Represents database

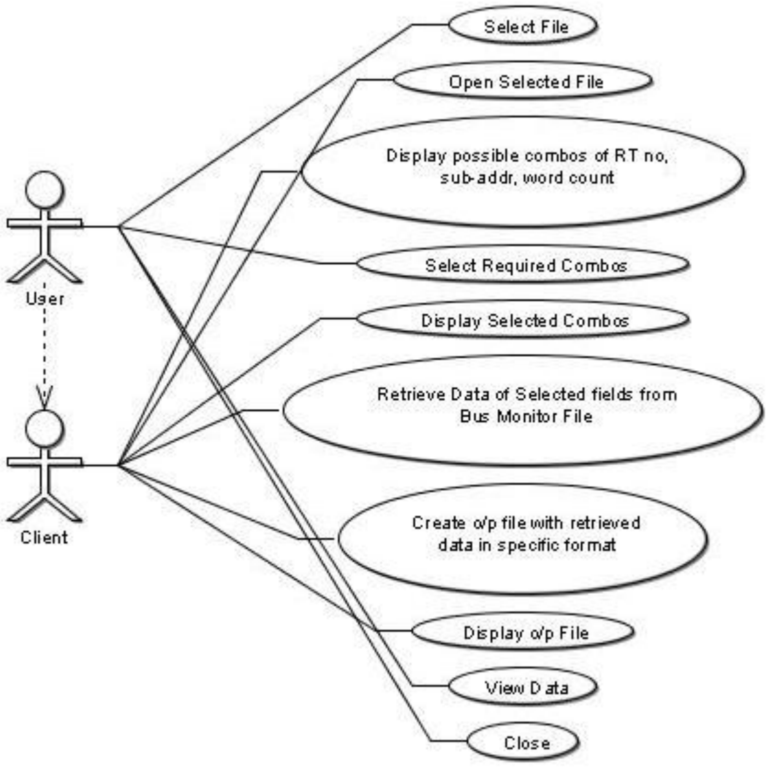
Three use case diagrams have been drawn to represent the three possible interactions in detail.



Real time data analysis



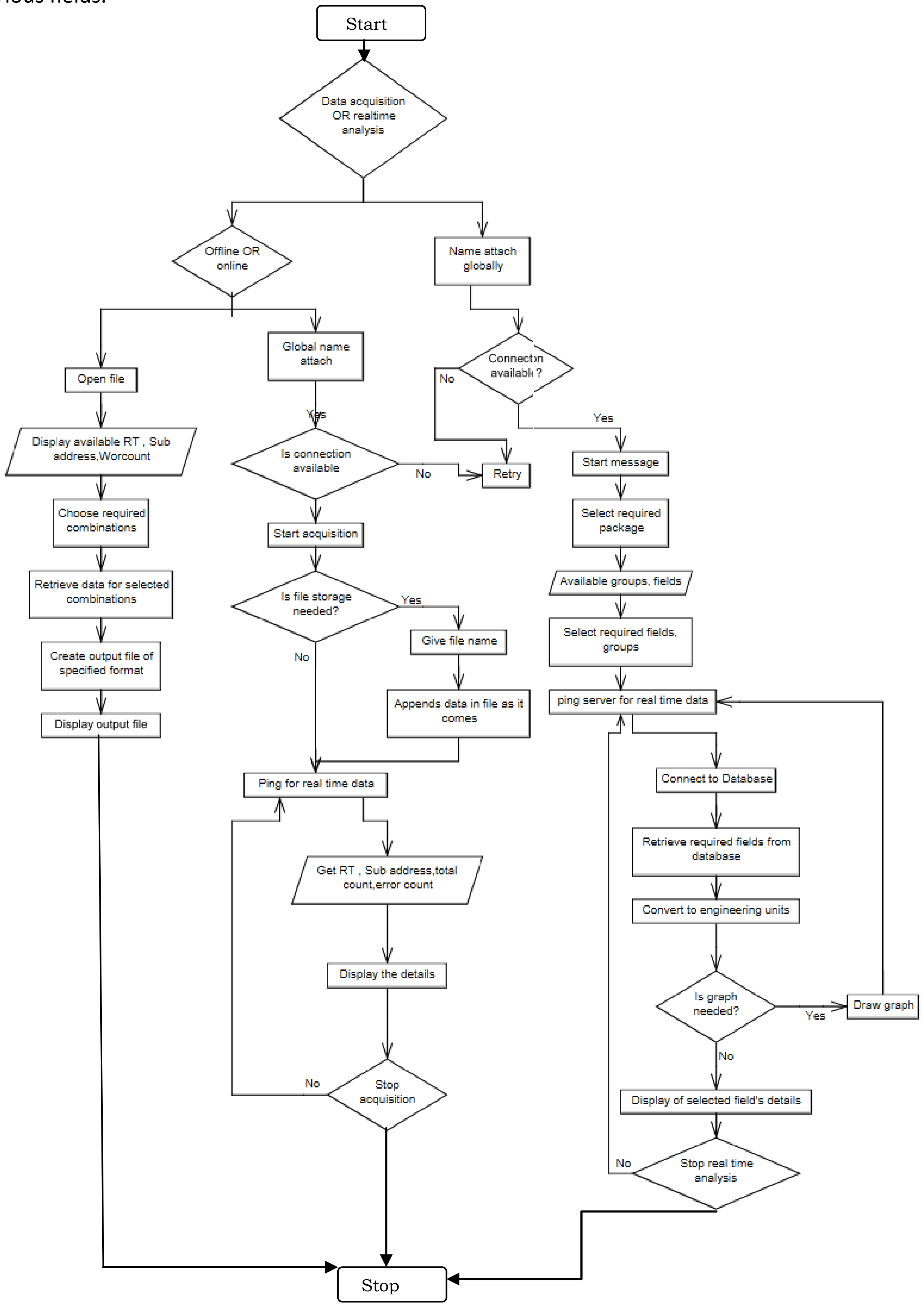
Real time bus monitoring


Select File
Open Selected File
Display possible combos of RT no, sub-addr, word count
Select Required Combos
Display Selected Combos
Retrieve Data of Selected fields from Bus Monitor File
Create o/p file with retrieved data in specific format
Display o/p File
View Data
Close
User
Client

Offline data analysis

1.2 Flow chart

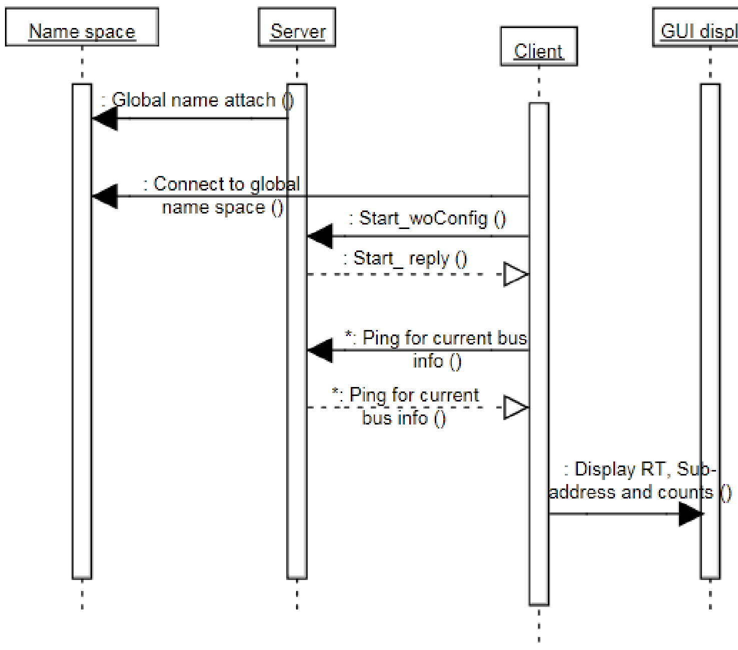
A **flowchart** is a common type of diagram, which represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. This diagrammatic representation can give a step-by-step solution to a given problem. Data is represented in these boxes, and arrows connecting them represent flow / direction of flow of data. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.


Start
Data acquisition OR realtime analysis
Offline OR online
Name attach globally
Open file
Global name attach
Connection available ?
No
Yes
Display available RT , Sub address,Worcount
Yes
Is connection available
No
Retry
Start message
Choose required combinations
Start acquisition
Select required package
Retrieve data for selected combinations
Is file storage needed?
Yes
Available groups, fields
Give file name
No
Select required fields, groups
Create output file of specified format
Appends data in file as it comes
ping server for real time data
Display output file
Ping for real time data
Connect to Database
Get RT , Sub address,total count,error count
Retrieve required fields from database
Convert to engineering units
Display the details
Is graph needed?
Yes
Draw graph
No
No
Stop acquisition
Display of selected field's details
No
Stop real time analysis
Stop

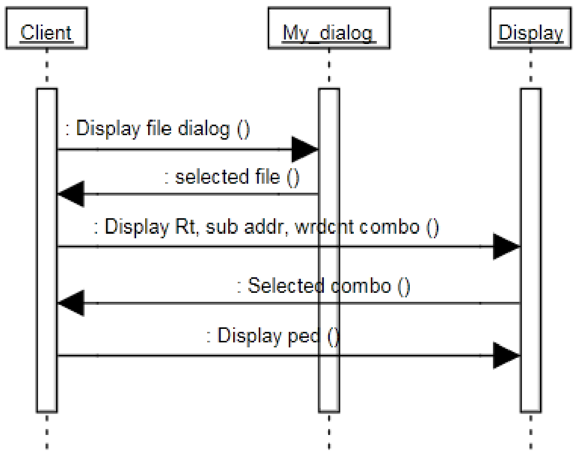
1.3 Sequence diagram

A **sequence diagram** in Unified Modeling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart.

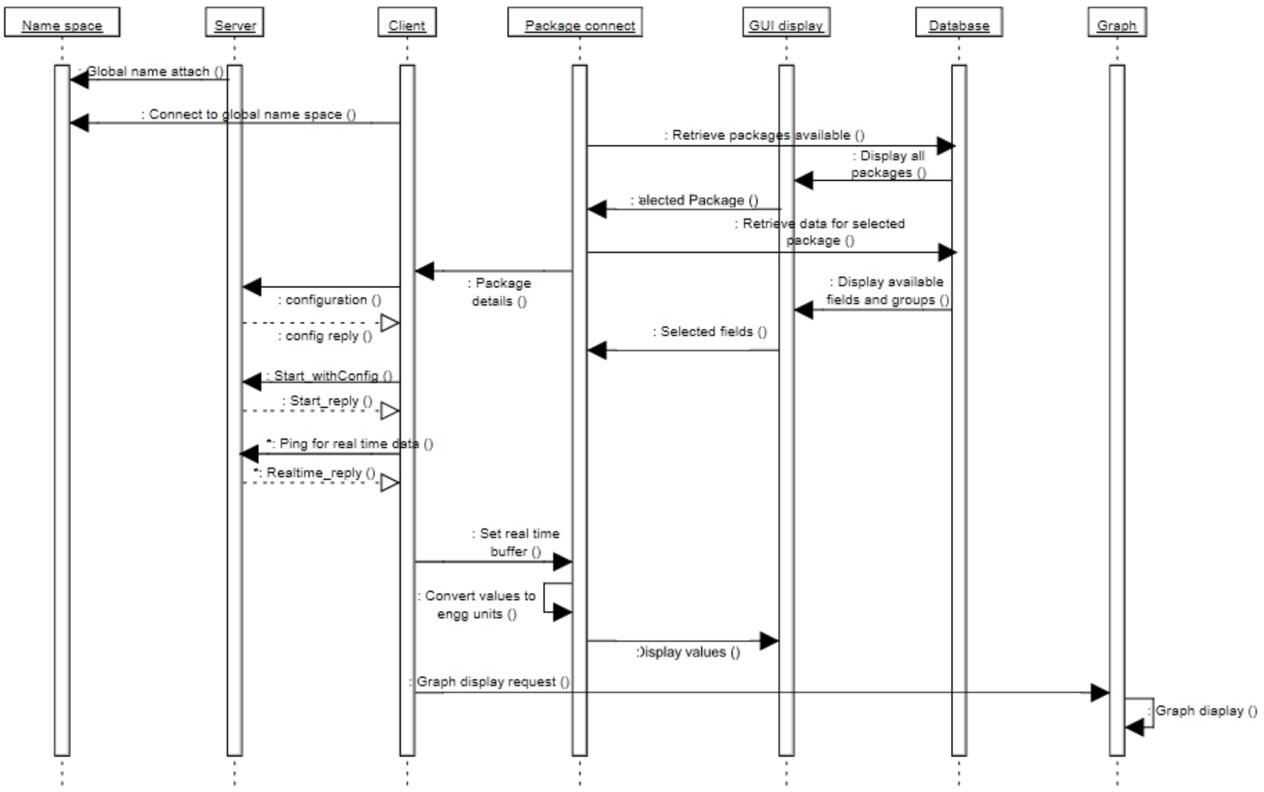
Sequence diagrams are sometimes called Event-trace diagrams, event scenarios, and timing diagrams.


Name space
Server
Client
GUI display
: Global name attach ()
: Connect to global name space ()
: Start_woConfig ()
: Start_ reply ()
*: Ping for current bus info ()
*: Ping for current bus info ()
: Display RT, Sub-address and counts ()

Real time bus monitoring



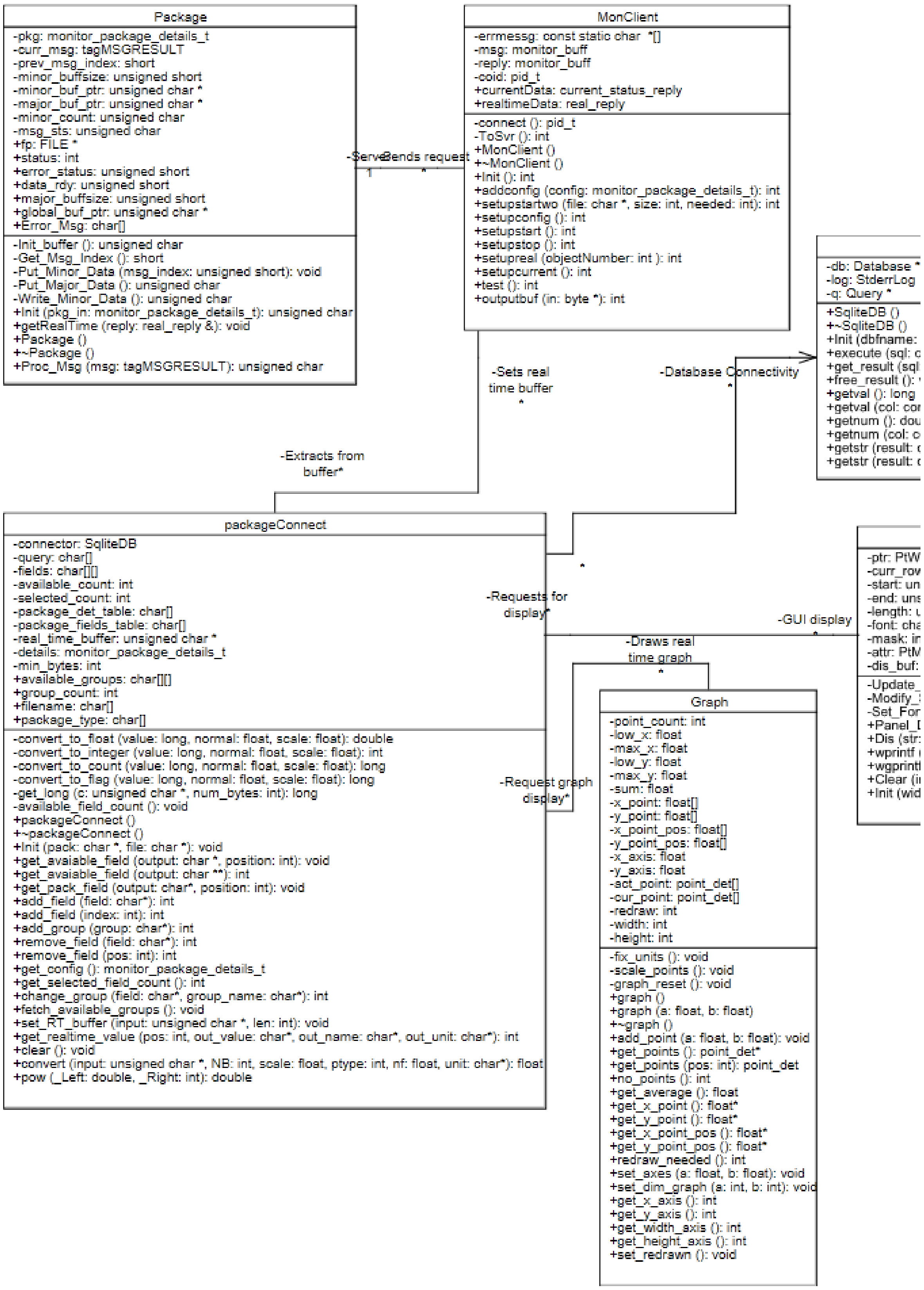
Offline data analysis


Name space
Server
Client
Package connect
GUI display
Database
Graph
: Global name attach ()
: Connect to global name space ()
: Retrieve packages available ()
: Display all packages ()
: Selected Package ()
: Retrieve data for selected package ()
: Package details ()
: configuration ()
: config reply ()
: Display available fields and groups ()
: Selected fields ()
: Start_withConfig ()
: Start_reply ()
*: Ping for real time data ()
*: Realtime_reply ()
: Set real time buffer ()
: Convert values to engg units ()
:Display values ()
: Graph display request ()
: Graph display ()

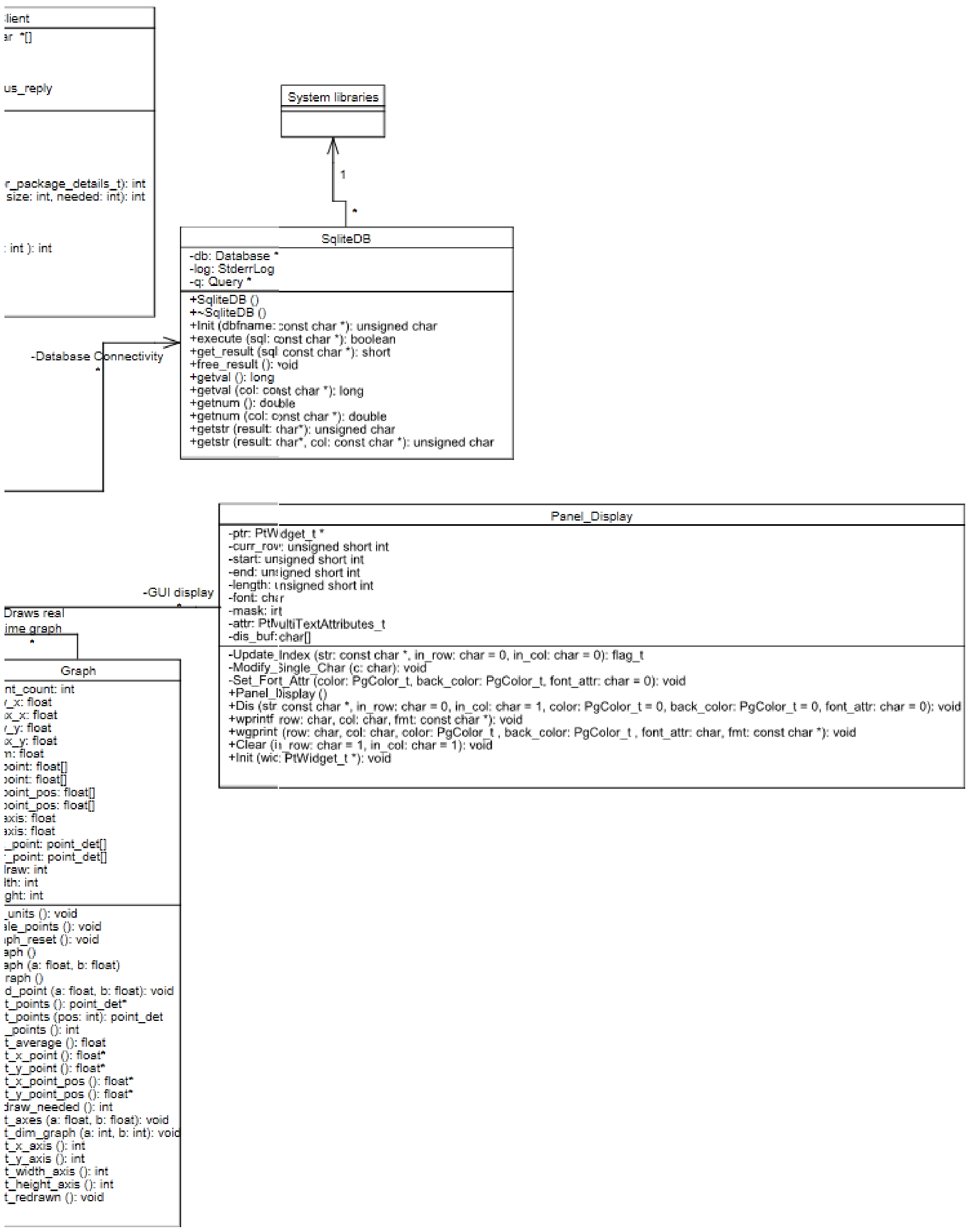
Real time data analysis

1.4 Class diagram

Class diagram describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among the classes.



Package
-pkg: monitor_package_details_t
-curr_msg: tagMSGRESULT
-prev_msg_index: short
-minor_buffsize: unsigned short
-minor_buf_ptr: unsigned char *
-major_buf_ptr: unsigned char *
-minor_count: unsigned char
-msg_sts: unsigned char
+fp: FILE *
+status: int
+error_status: unsigned short
+data_rdy: unsigned short
+major_buffsize: unsigned short
+global_buf_ptr: unsigned char *
+Error_Msg: char[]
-Init_buffer (): unsigned char
-Get_Msg_Index (): short
-Put_Minor_Data (msg_index: unsigned short): void
-Put_Major_Data (): unsigned char
-Write_Minor_Data (): unsigned char
+Init (pkg_in: monitor_package_details_t): unsigned char
+getRealTime (reply: real_reply &): void
+Package ()
+~Package ()
+Proc_Msg (msg: tagMSGRESULT): unsigned char
-Serve
1
-Sends request
*
MonClient
-errmessg: const static char *[]
-msg: monitor_buff
-reply: monitor_buff
-coid: pid_t
+currentData: current_status_reply
+realtimeData: real_reply
-connect (): pid_t
-ToSvr (): int
+MonClient ()
+~MonClient ()
+Init (): int
+addconfig (config: monitor_package_details_t): int
+setupstartwo (file: char *, size: int, needed: int): int
+setupconfig (): int
+setupstart (): int
+setupstop (): int
+setupreal (objectNumber: int): int
+setupcurrent (): int
+test (): int
+outputbuf (in: byte *): int
-db: Database *
-log: StderrLog
-q: Query *
+SqliteDB ()
+~SqliteDB ()
+Init (dbfname:
+execute (sql: c
+get_result (sql
+free_result ():
+getval (): long
+getval (col: cor
+getnum (): dou
+getnum (col: c
+getstr (result:
+getstr (result:
-Sets real
time buffer
*
-Database Connectivity
*
-Extracts from
buffer*
packageConnect
-connector: SqliteDB
-query: char[]
-fields: char[][]
-available_count: int
-selected_count: int
-package_det_table: char[]
-package_fields_table: char[]
-real_time_buffer: unsigned char *
-details: monitor_package_details_t
-min_bytes: int
+available_groups: char[][]
+group_count: int
+filename: char[]
+package_type: char[]
-convert_to_float (value: long, normal: float, scale: float): double
-convert_to_integer (value: long, normal: float, scale: float): int
-convert_to_count (value: long, normal: float, scale: float): long
-convert_to_flag (value: long, normal: float, scale: float): long
-get_long (c: unsigned char *, num_bytes: int): long
-available_field_count (): void
+packageConnect ()
+~packageConnect ()
+Init (pack: char *, file: char *): void
+get_avaiable_field (output: char *, position: int): void
+get_avaiable_field (output: char **): int
+get_pack_field (output: char*, position: int): void
+add_field (field: char*): int
+add_field (index: int): int
+add_group (group: char*): int
+remove_field (field: char*): int
+remove_field (pos: int): int
+get_config (): monitor_package_details_t
+get_selected_field_count (): int
+change_group (field: char*, group_name: char*): int
+fetch_available_groups (): void
+set_RT_buffer (input: unsigned char *, len: int): void
+get_realtime_value (pos: int, out_value: char*, out_name: char*, out_unit: char*): int
+clear (): void
+convert (input: unsigned char *, NB: int, scale: float, ptype: int, nf: float, unit: char*): float
+pow (_Left: double, _Right: int): double
*
-Requests for
display*
-GUI display
*
-Draws real
time graph
*
-ptr: PtW
-curr_row
-start: un
-end: uns
-length: u
-font: cha
-mask: in
-attr: PtM
-dis_buf:
-Update_
-Modify_
-Set_For
+Panel_
+Dis (str:
+wprintf
+wgprintf
+Clear (i
+Init (wid
-Request graph
display*
Graph
-point_count: int
-low_x: float
-max_x: float
-low_y: float
-max_y: float
-sum: float
-x_point: float[]
-y_point: float[]
-x_point_pos: float[]
-y_point_pos: float[]
-x_axis: float
-y_axis: float
-act_point: point_det[]
-cur_point: point_det[]
-redraw: int
-width: int
-height: int
-fix_units (): void
-scale_points (): void
-graph_reset (): void
+graph ()
+graph (a: float, b: float)
+~graph ()
+add_point (a: float, b: float): void
+get_points (): point_det*
+get_points (pos: int): point_det
+no_points (): int
+get_average (): float
+get_x_point (): float*
+get_y_point (): float*
+get_x_point_pos (): float*
+get_y_point_pos (): float*
+redraw_needed (): int
+set_axes (a: float, b: float): void
+set_dim_graph (a: int, b: int): void
+get_x_axis (): int
+get_y_axis (): int
+get_width_axis (): int
+get_height_axis (): int
+set_redrawn (): void


lient
ar *[]
us_reply
r_package_details_t): int
size: int, needed: int): int
: int): int
System libraries
1
*
SqliteDB
-db: Database *
-log: StderrLog
-q: Query *
+SqliteDB ()
+~SqliteDB ()
+Init (dbfname: const char *): unsigned char
+execute (sql: const char *): boolean
+get_result (sql const char *): short
+free_result (): void
+getval (): long
+getval (col: const char *): long
+getnum (): double
+getnum (col: const char *): double
+getstr (result: char*): unsigned char
+getstr (result: char*, col: const char *): unsigned char
-Database Connectivity
*
Panel_Display
-ptr: PtWidget_t *
-curr_row: unsigned short int
-start: unsigned short int
-end: unsigned short int
-length: unsigned short int
-font: char
-mask: int
-attr: PtMultiTextAttributes_t
-dis_buf: char[]
-Update_Index (str: const char *, in_row: char = 0, in_col: char = 0): flag_t
-Modify_Single_Char (c: char): void
-Set_Font_Attr (color: PgColor_t, back_color: PgColor_t, font_attr: char = 0): void
+Panel_Display ()
+Dis (str const char *, in_row: char = 0, in_col: char = 1, color: PgColor_t = 0, back_color: PgColor_t = 0, font_attr: char = 0): void
+wprintf (row: char, col: char, fmt: const char *): void
+wgprint (row: char, col: char, color: PgColor_t , back_color: PgColor_t , font_attr: char, fmt: const char *): void
+Clear (in_row: char = 1, in_col: char = 1): void
+Init (wid: PtWidget_t *): void
-GUI display
*
Draws real
time graph
*
Graph
nt_count: int
_x: float
x_x: float
_y: float
x_y: float
n: float
point: float[]
point: float[]
point_pos: float[]
point_pos: float[]
axis: float
axis: float
_point: point_det[]
_point: point_det[]
raw: int
th: int
ght: int
_units (): void
le_points (): void
ph_reset (): void
aph ()
aph (a: float, b: float)
raph ()
d_point (a: float, b: float): void
t_points (): point_det*
t_points (pos: int): point_det
_points (): int
t_average (): float
t_x_point (): float*
t_y_point (): float*
t_x_point_pos (): float*
t_y_point_pos (): float*
draw_needed (): int
t_axes (a: float, b: float): void
t_dim_graph (a: int, b: int): void
t_x_axis (): int
t_y_axis (): int
t_width_axis (): int
t_height_axis (): int
t_redrawn (): void

2. NON FUNCTIONAL REQUIREMENTS

2.1 Performance requirements

- There should be no data lost when read from bus
- The minor and major cycle data must be updated as and when data is read from bus
- Storage must be optimal – global bus file must be stored optimally
- Display must be update in real time – graph must also be plotted in real time
- Alert for out of range values
- There should be general ease of use for the client
- Provision for creation of user defined collection of parameters to be monitored
- Important of all - Server must never core dump. It should run infinitely and also ready to accept clients at any point. All the calculations must happen in real time. No time must be wasted during the processing.

2.2 Resource requirements

2.2.1 Hardware requirements

Server:

- Minimal required hardware to run the QNX Neutrino OS.
- 1553 Bus interface card.
- Input devices – mouse and key board.
- Mode of connection to a LAN.

Client:

- Minimal required hardware to run the QNX Neutrino OS.
- Input devices – mouse and key board.
- Output devices – monitor and printer if needed.
- Mode of connection to the LAN, to which sever is connected.

2.2.2 Software requirements

Server:

- Real time operating system - QNX Neutrino
- Drivers for the 1553 Bus interface card

Client:

- Real time operating system -QNX Neutrino
- SQLite compatibility
- A basic text editor – preferably ped

The development of the project needs the following software

- QNX Momentics IDE
- Photon Application builder - It is a visual design tool that generates the underlying C/C++ code to implement the application’s user interface
- SQLite

2.3 Quality and reliability requirements

- There minor and major cycles should be correct. There should not be any mistake in their generation
- Optimal storage must be done strictly adhering to the accepted protocol
- The output file of offline analysis must be compatible with all software that use the format
- At any point, the data shown to the user must be the latest – updated at real time
- The engineering conversions must be accurate

Design

1. Major functional components
2. Server
3. Master client software
4. Graph plotting software

1. MAJOR FUNCTIONAL COMPONENTS

The logical entities to which the user interacts are the following

1. Master Client Software: Real-time Data Monitor - with configuration (or) without configuration
2. Master Client Software: Offline Data Processing
3. Graph plotting software

Real-time Data Monitoring with (or) without configuration: The user interacts with "with configuration" use-case in selecting configurations that he wants to monitor in run-time and also the parameters he wants to check at run time. A configuration refers to a set of parameters that include – package to monitor (along with details of that package), specific RT number – sub address that has to be monitored in that package etc.

The user interacts with "without configuration use-case" if he wants to monitor the bus in run-time and to know the details of messages that are sent across the bus. Both use-cases internally interacts with server, which in turn does the following operations

- Data acquisition
- Data storage and major cycle generation

Offline-data processing software: The checkout user interacts with this use-case for the offline processing of the stored data files in the system viz. the global bus data.

Graph plotting software: The checkout user interacts with this use-case for the plotting the graph corresponding to real time parameters vs. RTC. The graph plotting software can be run from a FIFO that can be filled with values in two ways

- Master client software
- From output of "mcdash" program

The use-cases are exploded into more detail in the following sections. A textual description follows the pictorial representation for each use-case.

2. SERVER

The server is the one that has direct contact with the MIL-STD-1553 bus interface. It receives data from the bus every 20ms. Server does the following

- Reads the messages from the bus
- Checks for the validity of the messages and discard the invalid messages
- Stores the valid messages to be retrieved by the object of packages depending upon their configuration
- Responds to the client when it receives a message. The response must be according the request received
- Keeps generating minor cycles and major cycles for the compatible packages

Actors

The server exposes an interface as a list of commands that it supports. Any message with any such command is an actor to the use-case. The list of commands is as shown in the table.

Basic Flow

A server can manage more than one client. The flow discussed below is for single client. The various commands the server corresponds to are

No.	***Commands***
1	Adding configurations
2	Starting acquisition with configuration
3	Starting acquisition without configuration
4	Real time data request
5	Current bus stats request
6	Stop the acquisition

The main operations are

1. Server creates a named server and waits for connection. Once a client sends commands the server starts responding and processing.

2. **Real time data analysis**
    - Server initialises objects of the Package class, a user defined class, with the configurations got in the Adding configurations command. Each object corresponds to one configuration. The class is self sufficient. It can generate minor and major cycles for that package.

    - On receiving the start with configuration command it starts the data acquisition from the MIL – 1553 bus. Start command also specifies if a global file, that contains all the messages that are passed across the bus, is to be written. If needed the file is created with the desired name.

- It checks for the validity of the message. Invalid messages are discarded. Validity is checked based on the bits of status word and command word of the message. Valid messages are then written to global file if needed. The messages are written as stream of data after removing the unwanted details in order to save space. There is a specific set of rules that are followed in this process to facilitate easy re-construction of the messages from file when needed.

  Every valid message is then passed to all the objects of the package class. Those objects determine if the message is of their configuration. If so the corresponding object copies that message and analyses it. The 32 word data of the message is then put into the minor cycle. On reaching the number of minors for a major, the major cycle is written into the major cycle buffer of the object. The major cycle is also written into the file specified for the package by the user.

- In the mean time if a Real time data request command is got the server checks the package configuration for which it is requested. The major cycle buffer, at that point, of the corresponding object is sent to the client.

- The reading from the bus keeps continuing till the client sends a stop command. On stop the server clears all the memory used and goes back to waiting again for the next client to send a command.

3. **Real time monitoring with configuration**
   - On receiving the start without configuration command it starts the data acquisition from the MIL – 1553 bus. Start command also specifies if a global file, that contains all the messages that are passed across the bus, is to be written. If needed the file is created with the desired name.

   - It checks for the validity of the message. Invalid messages are discarded. Validity is checked based on the bits of status word and command word of the message. Valid messages are then written to global file if needed. The messages are written as stream of data after removing the unwanted details in order to save space. There is a specific set of rules that are followed in this process to facilitate easy re-construction of the messages from file when needed.

   - Updates the count of number of messages passed on the bus, number of valid messages and number of invalid messages.

   - The server also keeps track of the RT numbers involved in the messages sent, the sub address of the RTs and also the valid/invalid count for every RT number – sub address combination.

   - On receiving a Current bus stats request all these information are passed to the client.

The main functions are implemented using a set of threads with different priorities.

No.	Threads	Priority
1	Data acquisition	22
2	Data storage	11
3	Command handling Process - main	10

• **DATA ACQUISITION Read thread**: The read thread opens the data acquisition functions that will fetch the messages from the bus. It then checks for the validity of the messages. The validity is checked based on the block status field of the tagMSGRESULT structure. The valid messages are written into a pipe. Correspondingly the valid messages count, RT number sub address count are updated.

The thread runs at a very high priority of 22 so that there is no data loss. The data from the bus is received in bursts of messages every 20 ms and hence the chances of missing a few tags are very high. A high priority will ensure that the thread gets enough CPU time to acquire all the tags at real time.

• **DATA STORAGE Write thread:** The write thread reads the valid messages from the pipe. It then broadcasts the messages to all the package objects at hand. The package objects decide for themselves if the message is in accordance with the parameters they are configured to. The write thread then uses file operations to write the messages to a global bus data file if needed.

If configurations are selected the tag is sent to all the configuration objects (package class) to generate the minor data and then the major data.

• **COMMAND HANDLING thread:** This is the one that handles the messages of the system from the client. It returns the real-time major buffer data on request and also the current status of the bus for the without configuration request.

3. MASTER CLIENT SOFTWARE

This use-case refers to the main client program which schedules the different operations in the system, initiates data monitoring, bus monitoring and initiates generation of real-time major cycle data with display and logging support. It acts as client to the server mentioned above and requests services as and when required.

The client has been developed by using Photon graphics library of QNX.

Actors

The interface to the use-case is in the form of a QNX-Windows based GUI. The GUI contains entries corresponding to the different functions of the system. The user selects the configuration with the user interface and also the parameters that have to be monitored.

Basic flow

Three flows are possible.

1. **One for real time data analysis** –

- The user selects the packages he wants from a drop down list. He can specify the file name to store the data for the selected package. The client then opens the ***sqlite*** database connected to it. The data base has the configuration details like the RT number, sub address, word count, major cycle count etc. The row corresponding to the selected package is retrieved. Using that object of monitor_package_details_t structure is initialised. The user also specifies if there is a need for writing the major cycles of a package into a file. If so the file name is also sent.

  Once the user completes selecting packages the configurations are sent to the server via configuration command.

- There are two lists in the window for real time data analysis. One is populated with the fields available with the package. Another list is populated with the group available. User can select the fields needed or select groups needed. These are noted down for displaying the data later. These are fetched from the same data base. The database has all these details.

- The server sends the real time data, which contains the major cycle data of the packages selected, on real request command. Client requests real time data the client every second. The real time data got is then analysed. The database is used to find the byte position of each and every field in the major cycle and also the type of conversion needed to be done. The unit of that field is also found. Engineering conversions are done to get the value of the parameter.

The real time data got as a stream of bytes. The format has to be converted into engineering units before it can be used for any useful tasks. The algorithm used is as follows

i. Obtain the data to be converted.
ii. Connect to the database.

iii. Obtain the normalization factor, scale factor, and the number of bytes the data type occupies, for each and every selected parameter. Repeat the following for all of them
iv. If the normalization factor is 0, multiply the data with the scale factor to obtain the final value.
v. Else, if the normalization factor is 1, normalize the data using the formula, (Obtained data/maximum possible value of data)*1, and multiply this value with the scale factor to obtain the final value.
vi. Return final value. (The final value obtained is the value of the initial data in engineering units.)

- Only the selected field's data is displayed and is updated every second. Values in accepted range of the parameter are displayed in black. Values below range are displayed in blue and values beyond range are displayed in red.

- There is a provision for a real time graph. If the option is invoked by the client a RTC vs selected parameter graph is plotted continuously using the real time data got from the server.

2. Real time bus monitoring –

- The client requests for the bus stats every one second. The user specifies the file name to store the global bus data, if needed.

- The stats (at that time) are sent to the client every one second by the server. Stats is defined by a structure that has details of the bus – RT – Sub addresses combinations used, valid - invalid message that correspond to each of those configuration and size of the global file, if enabled.

- The window for displaying the bus data has columns for RT number, sub address, valid count and invalid count. These are updated every second with the data got, until user decides to stop the acquisition. The file size is also updated if global write is enabled.

3. **Offline data analysis** –

This option doesn't need the server to be running. It can be done fully on the client side.

- The client opens a file – of the type the global bus file written by the server.

- The messages are re constructed and the user is given a lot of options to choose – RT number. Sub address, word count combinations from the available set of combinations.

- The messages that pertain to the selected combinations are formatted in an ISRO standard format, written into a single text file and is displayed in a text editor. This text file can be used as input to a lot of other software for performing variety of analysis.

4. GRAPH PLOTTING SOFTWARE

The graph plotting software is used to plot the real time graph.

Actors

There are two possible actors - Master client software and output of “mcdash” program. Both the actors pump the data into a fifo. The fifo serves as the input for the graph software. The data is read from the fifo continuously, as and when data is pumped to it, and re scaled to be plotted.

Basic flow

The graph plotting algorithm is discussed in detail in algorithm section. Salient features of the graph plotting software are

- Real time plotting – data is plotted as and when got
- Automatic resizing – window resize is detect to make the graph occupy the whole window
- Dynamic scaling – scales for x and y axis are not fixed. It is calculated according to the input got. This ensures the graph is as legible as it can be. A scale change prompts the re drawing of the whole graph, including the already drawn part, to the new scale
- Average display – average of the inputs is displayed and updated for every input got.

Implementation

1. Concepts used
    1. Threads
    2. Message passing
    3. Pipes
2. Problems faced
3. Lessons learnt

1. CONCEPTS USED

1.1 Threads

A system is a collection of processes, where each process is responsible for providing a service of some nature. A process can have one or more threads. A thread can be thought of as the minimum "unit of execution," the unit of scheduling and execution in the microkernel. A process is a "container" for threads, defining the "address space" within which threads will execute. A process will always contain at least one thread.

Thread Attributes

Although threads within a process share everything within the process's address space, each thread still has some "private" data.

1. Tid – Unique identification of each thread within the process. Starts at 1
2. Priority – Helps to determine when it runs.
3. Name – Can assign a name to a thread.
4. Register Set – Thread's own instruction pointer (IP), stack pointer (SP), and other processor-specific register context.
5. Stack - Each thread executes on its own stack, stored within the address space of its process.
6. Thread Local Storage (TLS) – System defined data area to store "per thread information".
7. Cancellation Handlers - Call-back functions that are executed when the thread terminates.

Thread Lifecycle

Thread creation involves allocating and initializing the necessary resources within the process's address space (e.g. thread stack) and starting the execution of the thread at some function in the address space. Thread termination involves stopping the thread and reclaiming the thread's resources. In between the initialization and the termination, a thread can exist in many states such as DEAD, INTERRUPT, NANOSLEEP, JOIN, MUTEX, READY, RECIEVE etc.. The thread's lifecycle starts from the initialization of the thread and lasts till its termination.

Thread Join

The thread join property allows a thread to wait for the termination of another thread to terminate before it terminates itself. This is used for exit synchronization.

Thread Priority

The thread priority is used to indicate the "importance" of the thread with respect to the others. It is global priority across the entire system. Apart from only the priority scheduling algorithms also come into play to decide the order of execution of threads.

Thread Scheduling

Scheduling basically determines the order of execution of the threads. A scheduling decision is made whenever the execution state of any thread changes. Threads are scheduled globally across all processes in the system.A thread scheduler performs a context switch from one thread to another when the running thread moves to one of the following states.

1. Blocked: Thread must wait for some event to occur.
2. Pre-empted: A higher priority thread is placed on the ready queue.
3. Yielded: Thread voluntarily yields the processor and is placed at the end of the ready queue for that priority.

Scheduling Algorithms:

QNX Neutrino supports three scheduling algorithms – FIFO, round robin, and sporadic scheduling.

1. FIFO Scheduling
   A thread selected to run continues executing until it:
   - voluntarily relinquishes control (e.g. it blocks)
   - is pre-empted by a higher-priority thread
2. Round Robin Scheduling - A thread selected to run continues executing until it:
   - voluntarily relinquishes control
   - is pre-empted by a higher-priority thread
   - consumes its *time slice*
3. Sporadic Scheduling

Employs a “budget” for a thread's execution. If a higher-priority thread becomes READY, it immediately pre-empts all lower-priority threads. It allows a thread to service aperiodic events without jeopardizing the hard deadlines of other threads or processes in the system.

A running thread continues executing until it blocks or is pre-empted by a higher-priority thread. The thread may drop in priority, but with sporadic scheduling there is much more precise control over the thread's behavior.

1.2 Message passing

Under QNX Neutrino, the modules communicate among themselves via message passing. The application is considered a client requesting data from a server. The server processes the data and sends back the necessary processed data and information to the client. The communication between a server and client takes place through a channel.

Channels and Connections

A thread that wishes to receive messages first creates a channel. Another thread that wishes to send a message to that thread must first make a connection by “attaching” to that channel.

Channels are required by the message kernel calls and are used by servers to receive messages on. Connections are created by client threads to “connect” to the channels made available by servers. Once connections are established, clients can send messages over them. If a number of threads in a process all attach to the same channel, then the connections all map to the same kernel object for efficiency. Channels and connections are named within a process by a small integer identifier. Client connections map directly into file descriptors.

Server and Client Threads

1. **The client thread:**

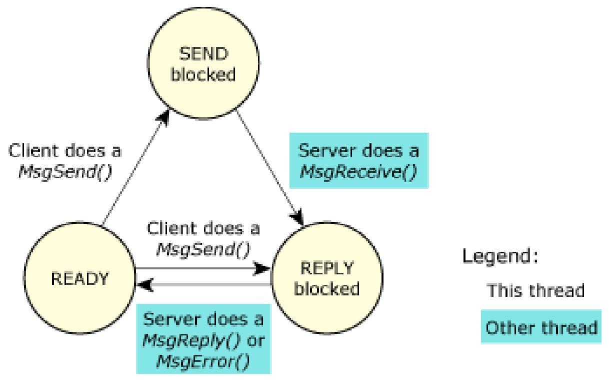


Fig 1.2.1 Changes of state for a client thread in a send-receive-reply transaction.

- If the client thread calls *MsgSend()*, and the server thread hasn't yet called *MsgReceive()*, then the client thread becomes SEND blocked. Once the server thread calls *MsgReceive()*, the kernel changes the client thread's state to be REPLY blocked, which means that server thread has received the message and now must reply. When the server thread calls *MsgReply()*, the client thread becomes READY.
- If the client thread calls *MsgSend()*, and the server thread is already blocked on the *MsgReceive()*, then the client thread immediately becomes REPLY blocked, skipping the SEND-blocked state completely.
- If the server thread fails, exits, or disappears, the client thread becomes READY, with *MsgSend()* indicating an error.

2. **The server thread:**

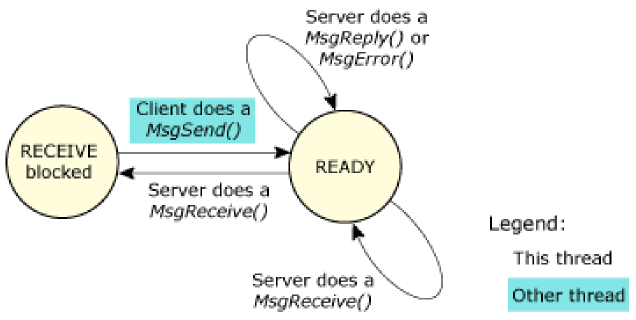

Server does a MsgReply() or MsgError()
Client does a MsgSend()
RECEIVE blocked
READY
Server does a MsgReceive()
Server does a MsgReceive()
Legend:
This thread
Other thread

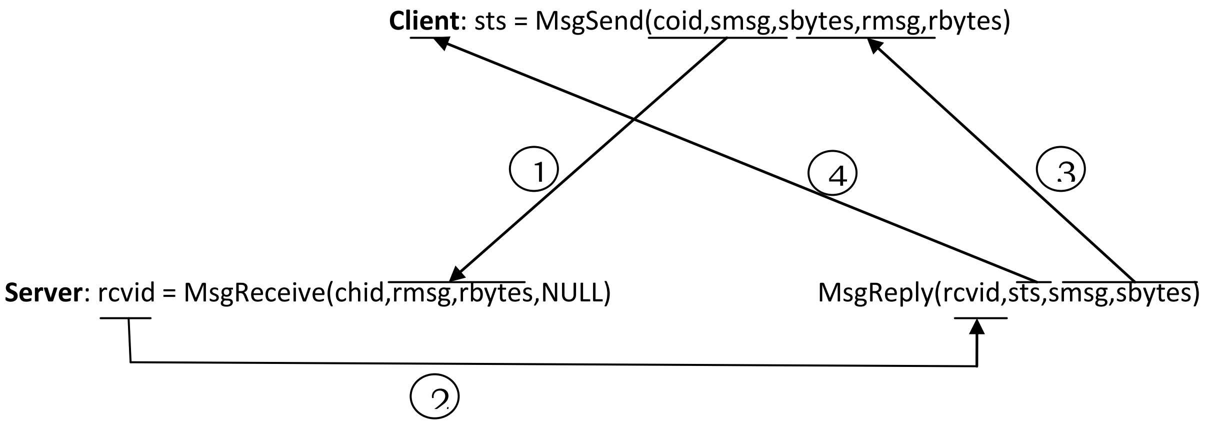
Fig 1.2.2 Changes of state for a server thread in a send-receive-reply transaction.

- If the server thread calls *MsgReceive()*, and no other thread has sent to it, then the server thread becomes RECEIVE blocked. When another thread sends to it, the server thread becomes READY.

- If the server thread calls *MsgReceive()*, and another thread has already sent to it, then *MsgReceive()* returns immediately with the message. In this case, the server thread doesn't block.
- If the server thread calls *MsgReply()*, it doesn't become blocked.

This inherent blocking synchronizes the execution of the sending thread. This happens without requiring explicit work by the kernel.

Message Data Flow


Client: sts = MsgSend(coid,smsg,sbytes,rmsg,rbytes)
1
4
3
Server: rcvid = MsgReceive(chid,rmsg,rbytes,NULL)
MsgReply(rcvid,sts,smsg,sbytes)
2

1.3 Pipes

Pipes form a subset of the file I/O. As a simple form of inter-process communication, pipes are mostly used to connect the output of one process to the input of another to form a series of filters. The *pipe()* function is used to create a pipe. On creation, it places a file descriptor each for the read end and write end of the pipe. Data can be written through the write file descriptor, and read using the read file descriptor. The value that is written first is read first, following the FIFO (First In First Out) scheme. A pipe may thus also be considered an unnamed or an anonymous FIFO.

The pipe buffer is allocated by the pipe resource manager. The total buffer size of the pipe must be at least the atomic write size. The buffer size indicates the number of unread bytes of information the pipe can hold effectively before the write clients start to block or fail.

The pipe utility is terminated only if an error occurs during its start. In such a case, a diagnostic message is printed to the standard output stream and it exists with a non-zero status.

2. PROBLEMS FACED

(i) Initially, a dual buffer scheme was used. The scheme is implemented as follows.

As the name suggests, two buffers are used. The required data read from the bus is written into the first buffer. Once the buffer is full, the buffer is said to be ready for reading. The buffer is then read, and the values read from the buffer are displayed to the user using visual display aids. As the first buffer is read, any more data read from the bus is written into the second buffer till it becomes ready for reading. Once all the data from the first buffer is read, it is cleared and made ready for writing again. Thus the buffers keep alternating between read and write.

This scheme did not prove very efficient, as there is a time delay between writing into the buffer and reading from it, considering a buffer is made ready for reading only after it is full. This time difference proves costly as we are dealing with real time data, changing every few milliseconds. Moreover, the last bit of data written in the buffer is never read as the buffer does not get full, and only filling the buffer makes it ready for reading.

To overcome this problem, pipes were used. Data is written into the pipe from one end, and read from the other, using two file pointers, one for reading and one for writing. The two actions can thus be dome independent of each other, avoiding the time lag incurred using the Dual Buffer scheme.

(ii) The data types of the data that are read are different from the traditional data types used in the C language. A lot of data types have been created by ISRO for use on the indigenous processors. The problem was to store those data types using the traditional c data types without loss in precision or altering the data.

A conversion algorithm has been devised and used to convert the data into engineering units, meaningful to the user.

3. LESSONS LEARNT

- Work environment management
- Coping up with pressure and deadlines
- Time management
- Building an entire application and solving the problems faced without the help of the Internet (Internet usage is highly restricted inside ISRO’s facility)
- The importance of good object oriented programming
- Understanding new systems and hardware from the basics and working on them
- Coding to company standards

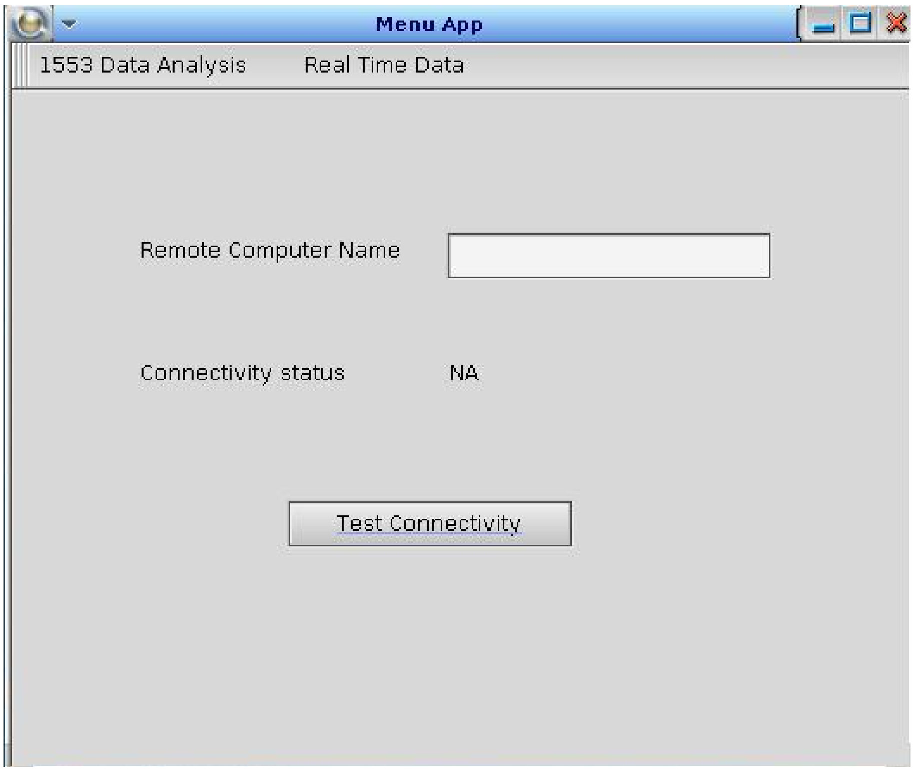
Screen shots

1. Welcome screen

2. 1553 Data analysis
    1. Real time bus monitoring
    2. Offline analysis

3. Real time data analysis
    1. Package selection
    2. Group create
    3. Real time data display
    4. Real time graph

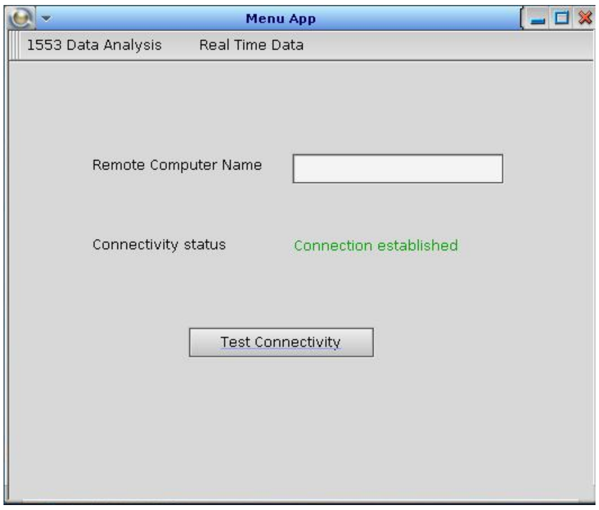
1. WELCOME SCREEN



Remote Computer Name

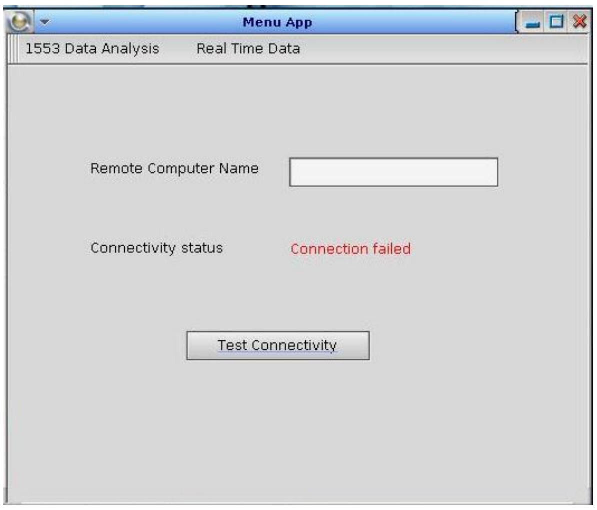
Connectivity status NA

Fig 1.1 Welcome screen



Remote Computer Name

Connectivity status Connection established



Remote Computer Name

Connectivity status Connection failed

Fig 1.2 Connection available
Server is visible and accepting connections

Fig 1.3 Connection failed
Server is not available (or) nor accepting connections

2. 1553 DATA ANALYSIS

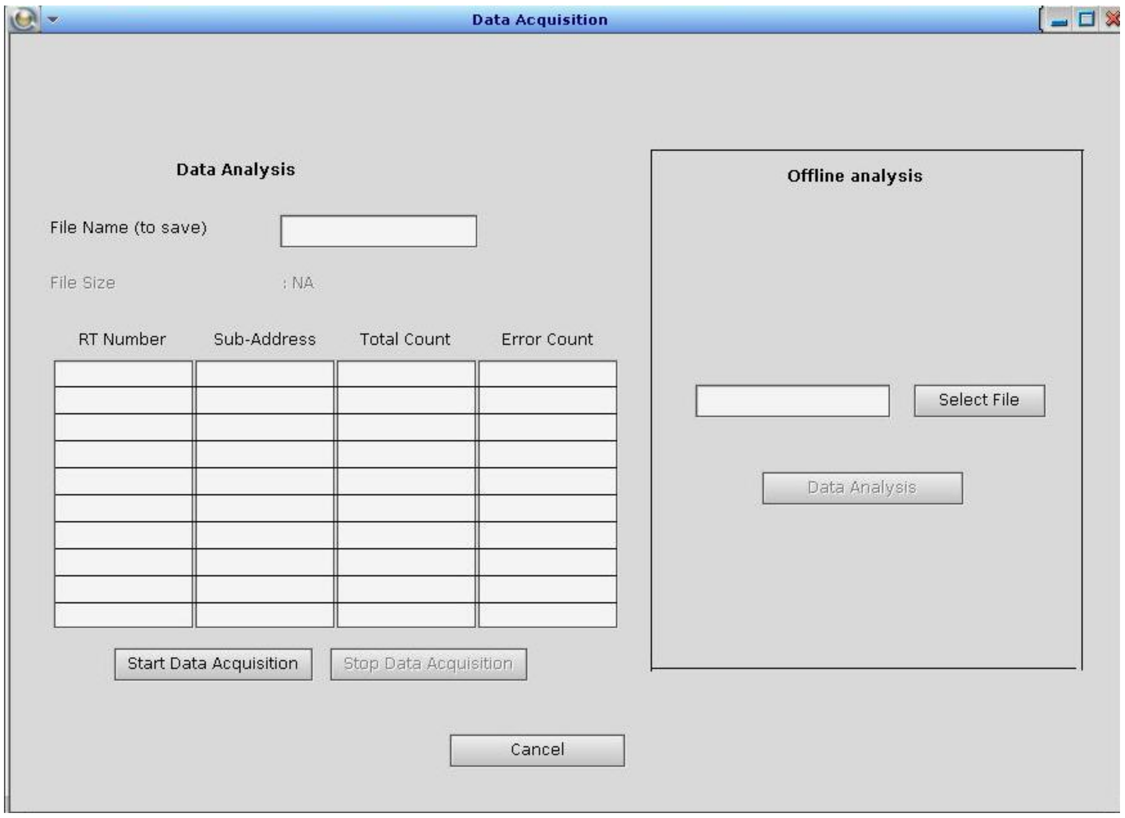
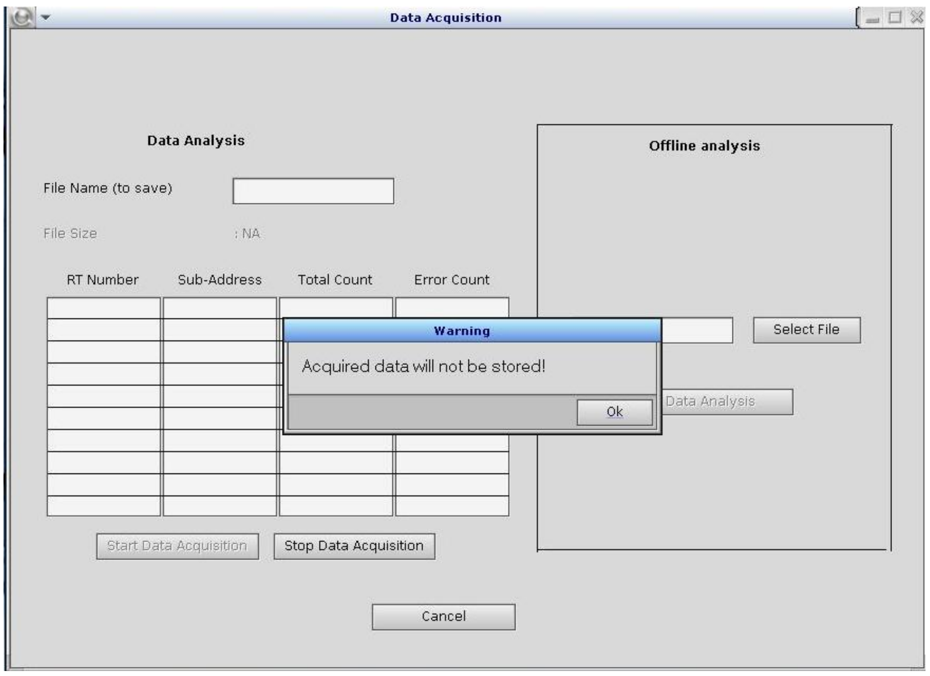


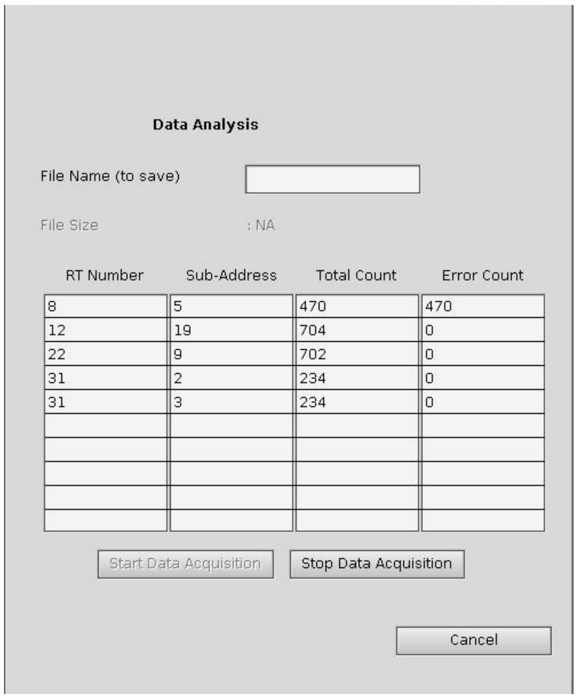
Fig 2.1 Real time data analysis home window

2.1 Real time bus monitoring



Acquired data will not be stored!

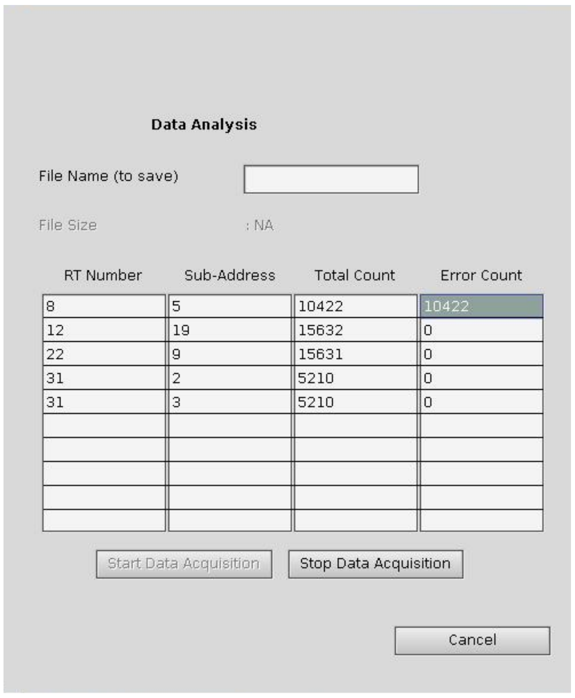
Fig 2.1.1 Warning for starting bus monitor without file to save option


Data Analysis

File Name (to save)

File Size : NA

RT Number	Sub-Address	Total Count	Error Count
8	5	470	470
12	19	704	0
22	9	702	0
31	2	234	0
31	3	234	0

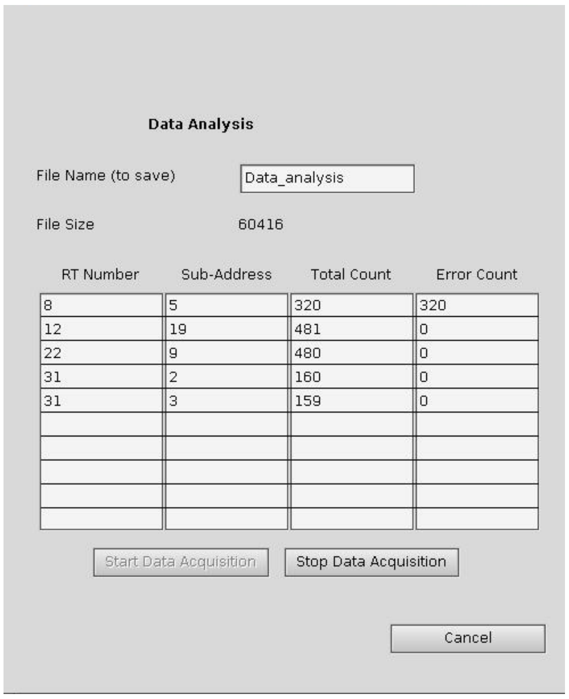

Data Analysis

File Name (to save)

File Size : NA

RT Number	Sub-Address	Total Count	Error Count
8	5	10422	10422
12	19	15632	0
22	9	15631	0
31	2	5210	0
31	3	5210	0

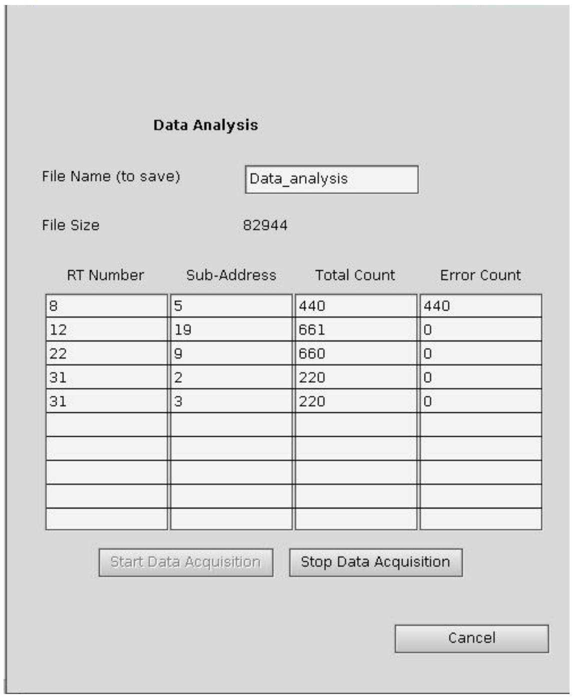
Fig 2.1.2 Data being filled in the corresponding column


Data Analysis

File Name (to save) Data_analysis

File Size 60416

RT Number	Sub-Address	Total Count	Error Count
8	5	320	320
12	19	481	0
22	9	480	0
31	2	160	0
31	3	159	0


Data Analysis

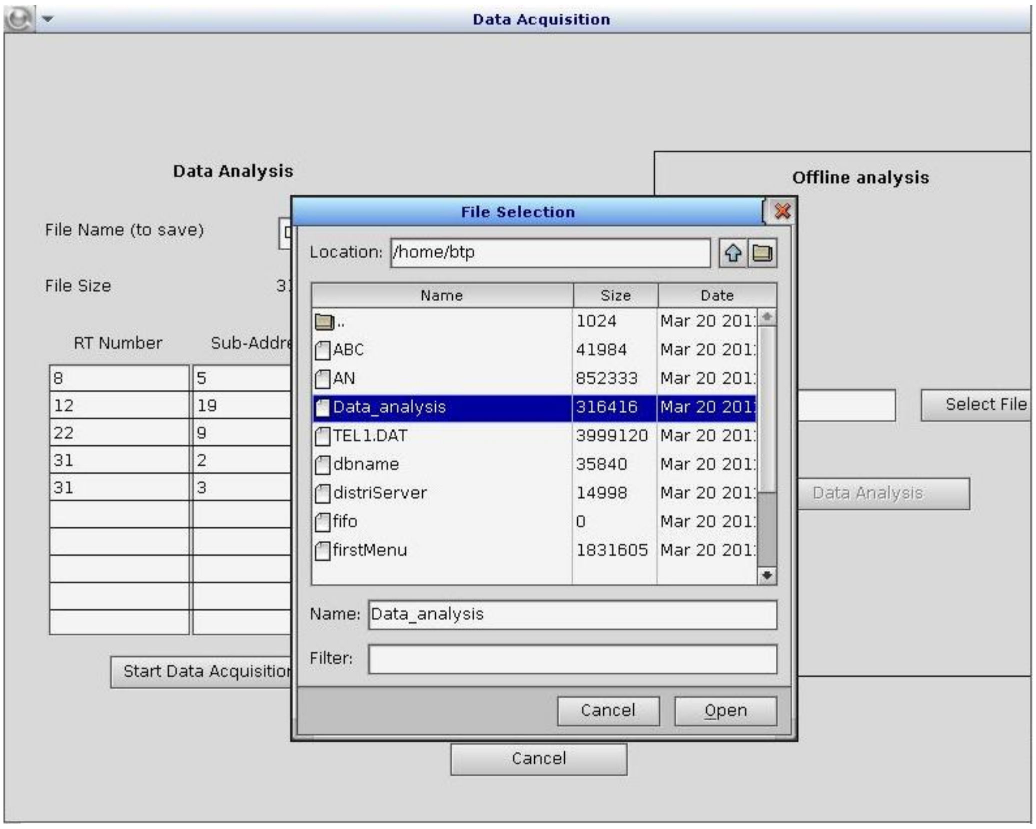
File Name (to save) Data_analysis

File Size 82944

RT Number	Sub-Address	Total Count	Error Count
8	5	440	440
12	19	661	0
22	9	660	0
31	2	220	0
31	3	220	0

Fig 2.1.3 Data being filled in the corresponding column
Here a file name is specified and hence the file size is also updated

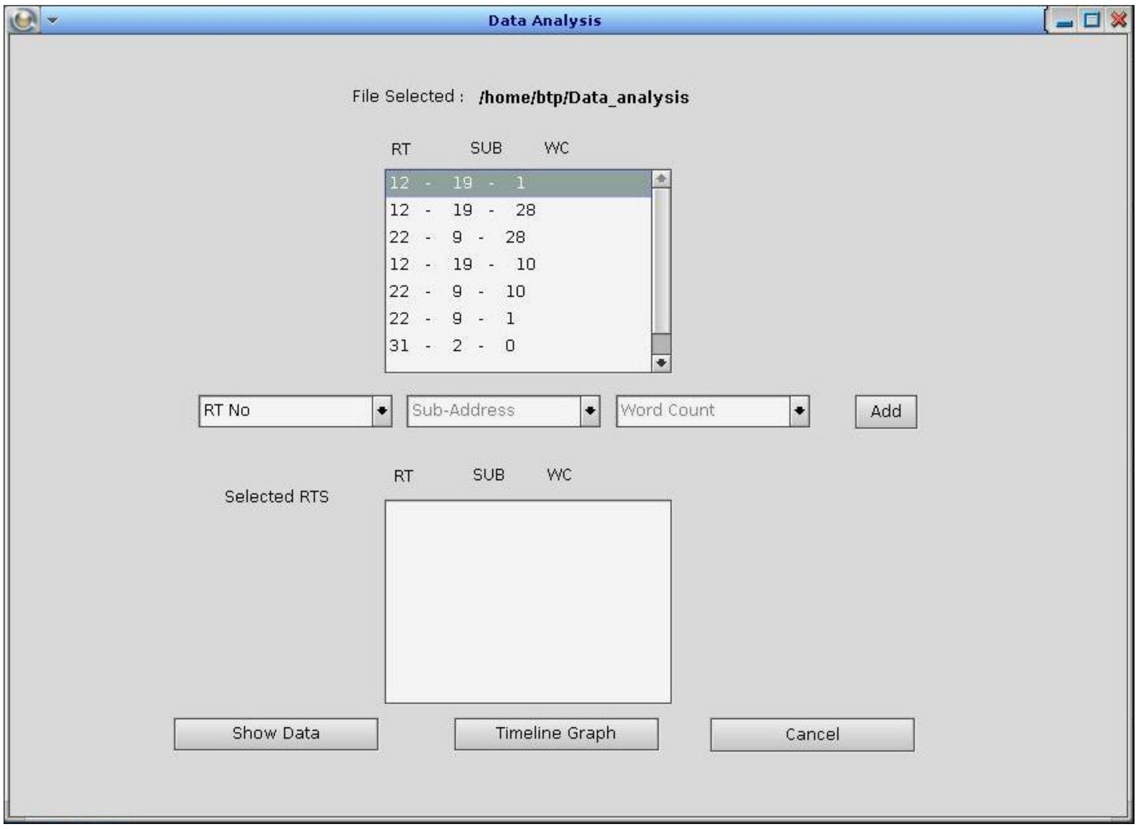
2.2 Offline analysis



RT Number	Sub-Addr
8	5
12	19
22	9
31	2
31	3

Name	Size	Date
..	1024	Mar 20 201
ABC	41984	Mar 20 201
AN	852333	Mar 20 201
Data_analysis	316416	Mar 20 201
TEL1.DAT	3999120	Mar 20 201
dbname	35840	Mar 20 201
distriServer	14998	Mar 20 201
fifo	0	Mar 20 201
firstMenu	1831605	Mar 20 201

Fig 2.2.1 Offline file selection file dialog



RT	SUB	WC
12	19	1
12	19	28
22	9	28
12	19	10
22	9	10
22	9	1
31	2	0

Fig 2.2.2 Offline analysis window home for the selected file

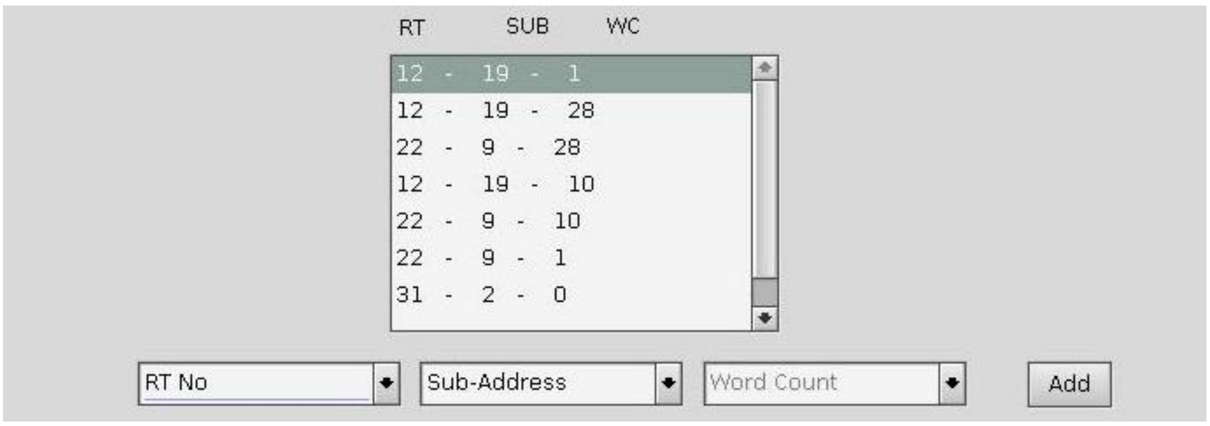


Fig 2.2.3 Available RT-SUB-WC combinations in the selected file

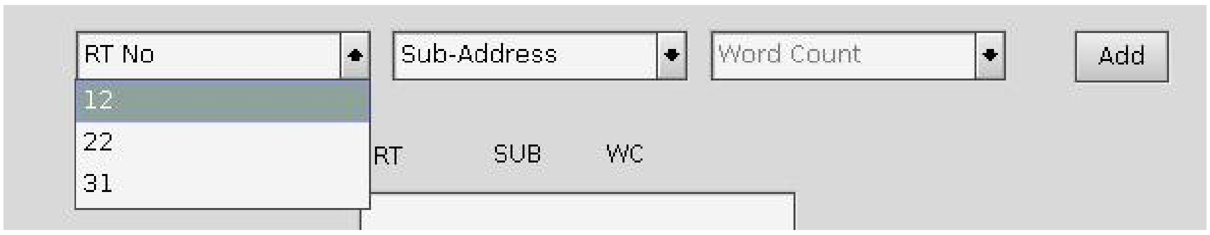


Fig 2.2.4 Selecting the RT

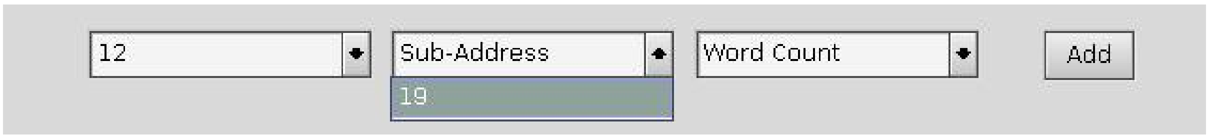


Fig 2.2.5 Selecting the SUB address for the selected RT

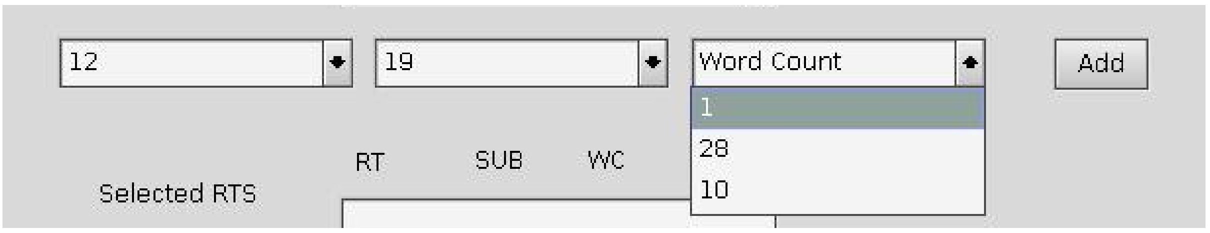


Fig 2.2.6 Selecting the WC for selected sub address, RT

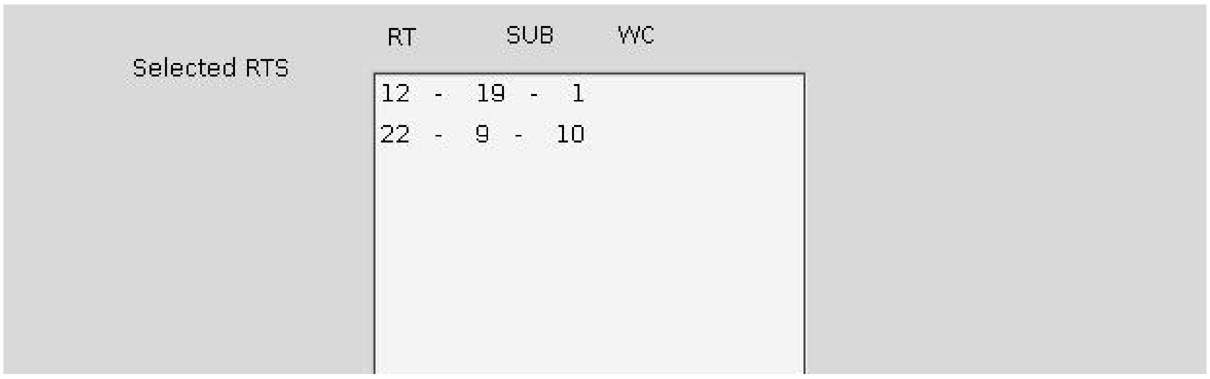


Fig 2.2.7 Selected combinations added

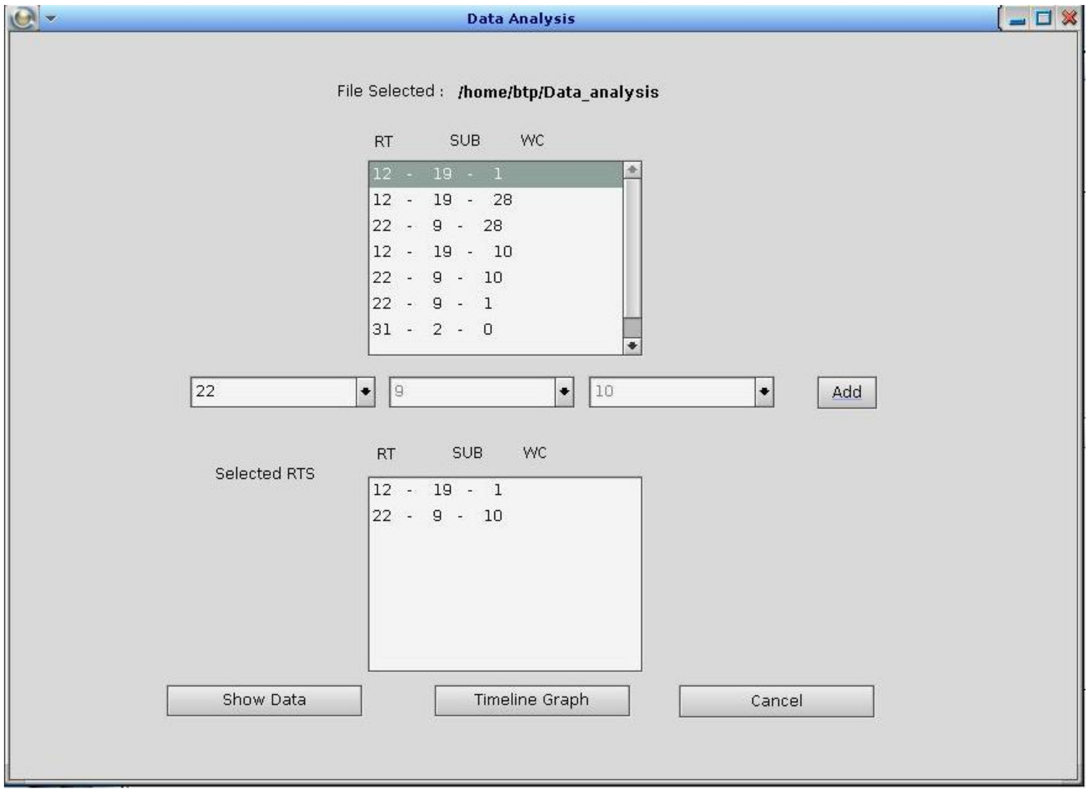
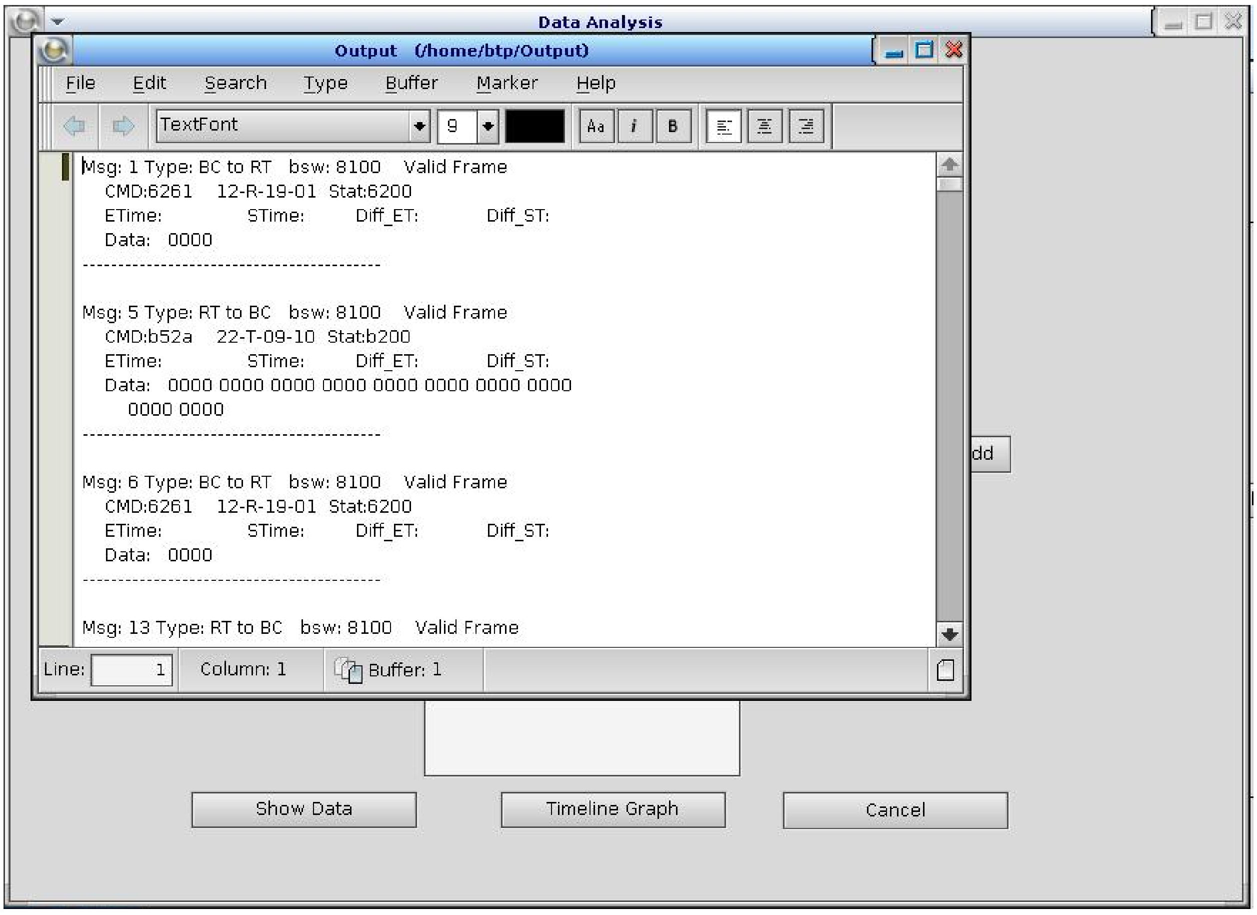


Fig 2.2.8 Packages selected are added



```
Msg: 1 Type: BC to RT   bsw: 8100    Valid Frame
   CMD:6261    12-R-19-01  Stat:6200
   ETime:            STime:        Diff_ET:          Diff_ST:
   Data:   0000
----------------------------------------

Msg: 5 Type: RT to BC   bsw: 8100    Valid Frame
   CMD:b52a    22-T-09-10  Stat:b200
   ETime:            STime:        Diff_ET:          Diff_ST:
   Data:   0000 0000 0000 0000 0000 0000 0000 0000
      0000 0000
----------------------------------------

Msg: 6 Type: BC to RT   bsw: 8100    Valid Frame
   CMD:6261    12-R-19-01  Stat:6200
   ETime:            STime:        Diff_ET:          Diff_ST:
   Data:   0000
----------------------------------------

Msg: 13 Type: RT to BC   bsw: 8100    Valid Frame
```

Fig 2.2.9 Show selected displaying formatted data in a PED window

3. REAL TIME DATA ANALYSIS

3.1 Package Selection

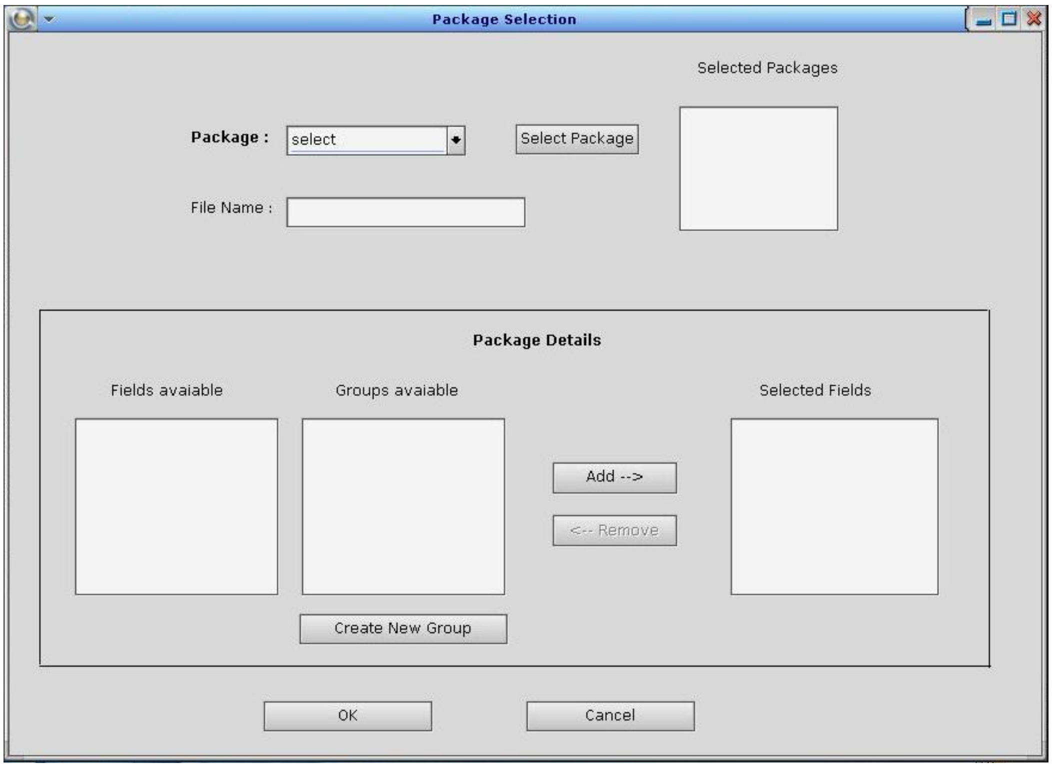


Fig 3.1 Real time data analysis home

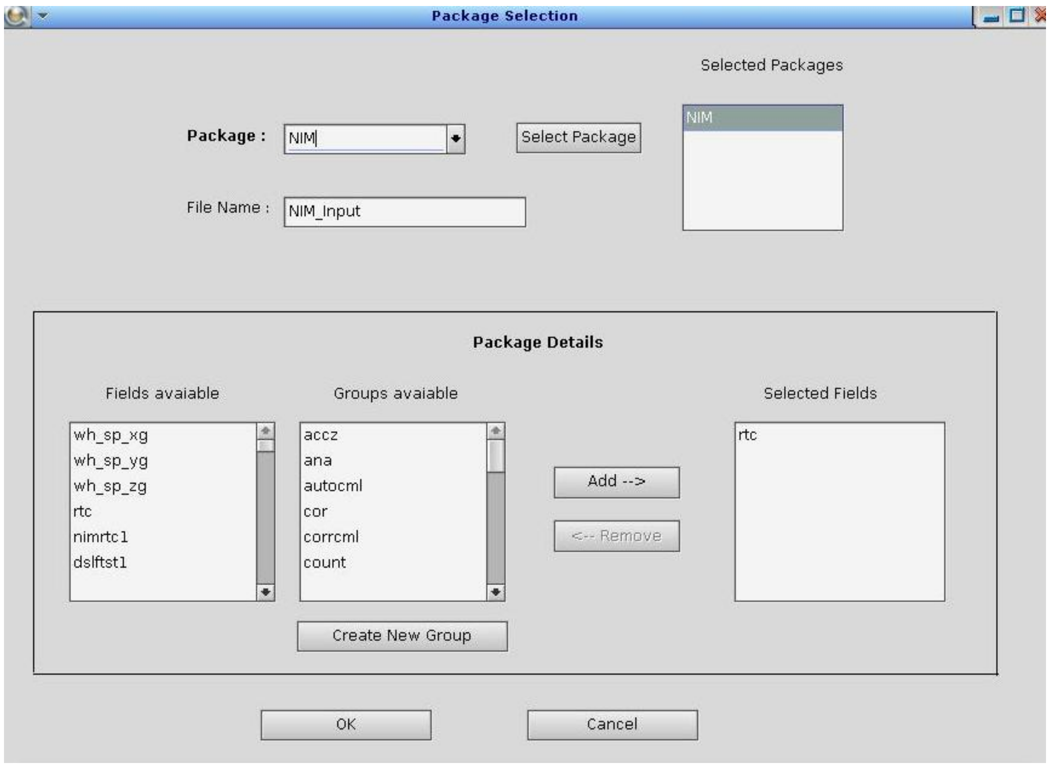


Fig 3.1.2 Package NIM selected – rtc field selected

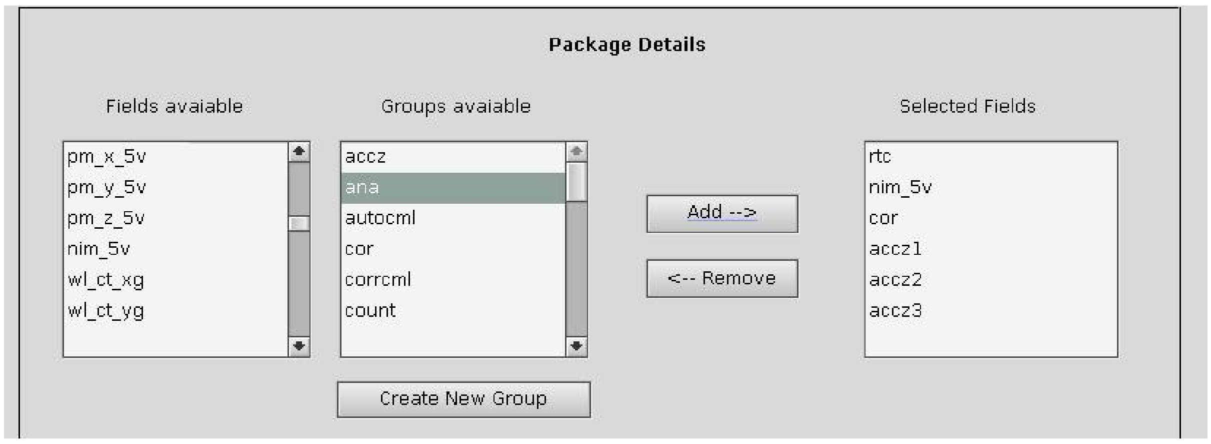


Fig 3.1.2 Group add – Group added accz

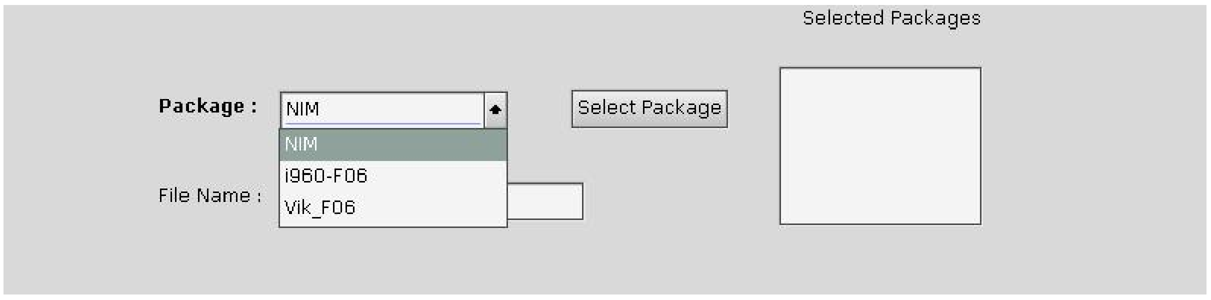


Fig 3.1.3 Packages available – Dropdown

3.2 Group Creation

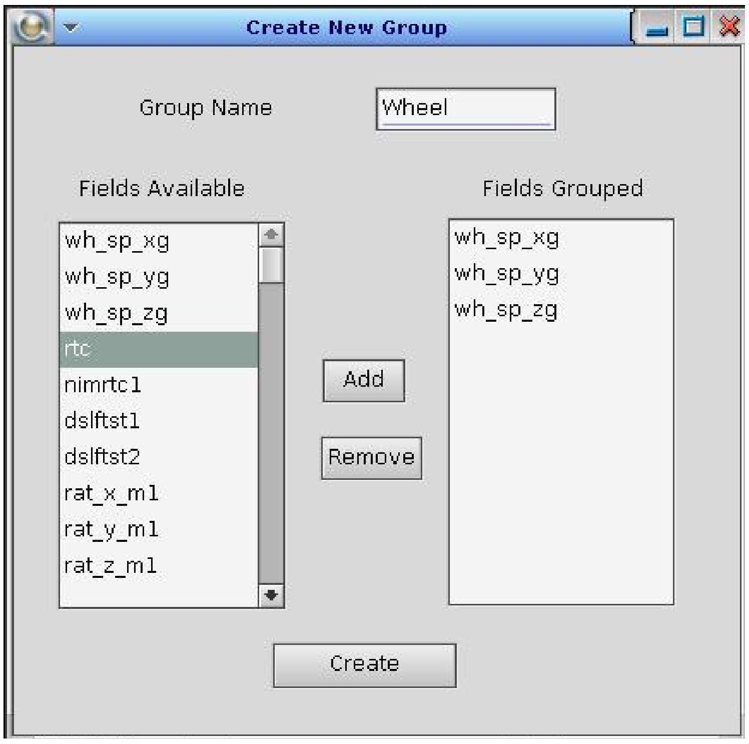
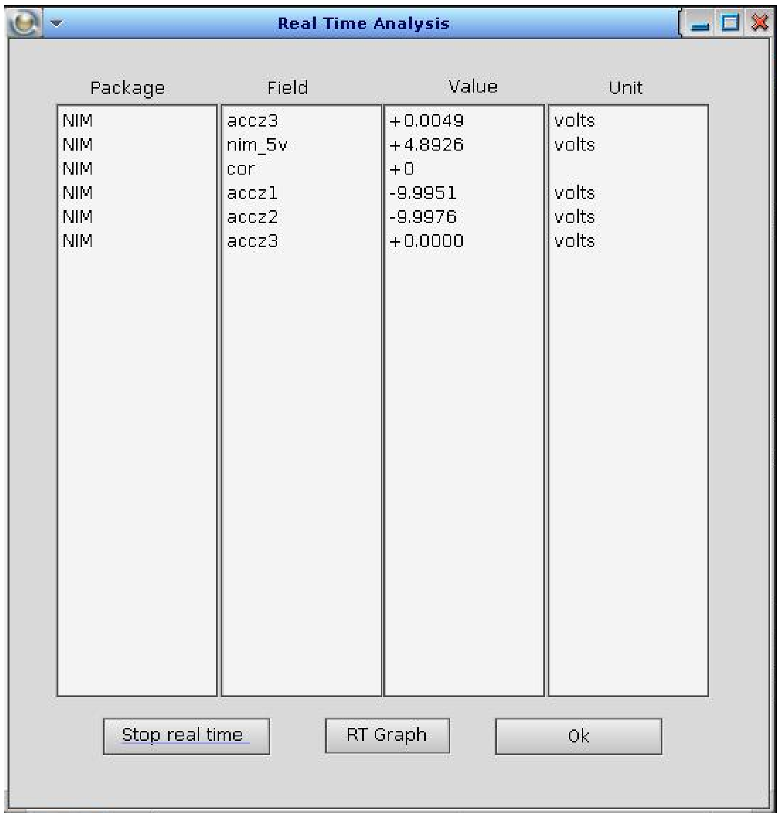


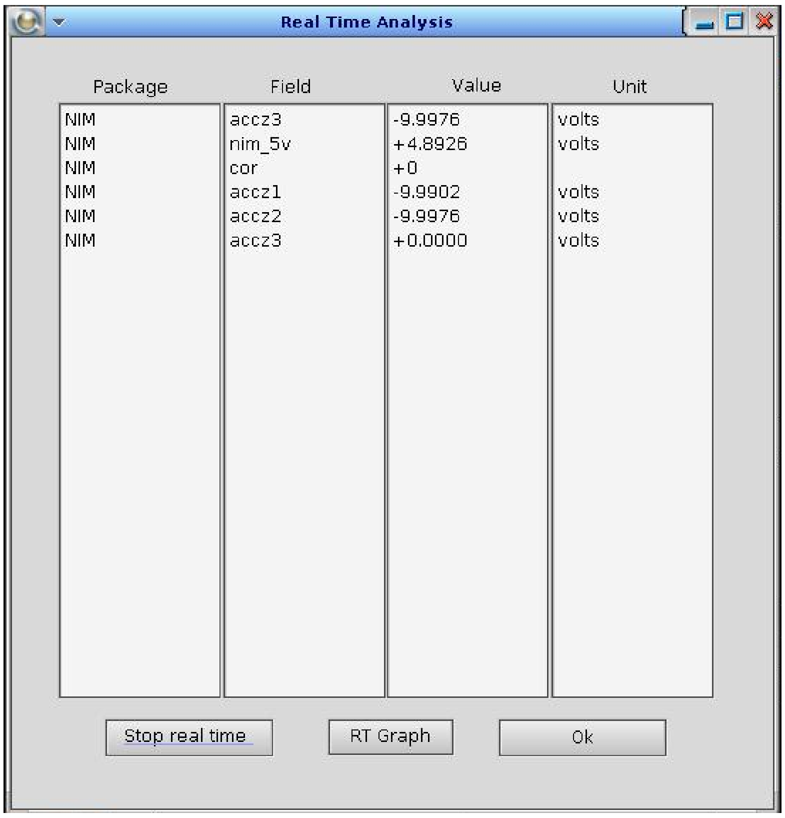
Fig 3.2.1 Group creation

3.3 Real Time Data Display



Package	Field	Value	Unit
NIM	accz3	+0.0049	volts
NIM	nim_5v	+4.8926	volts
NIM	cor	+0	
NIM	accz1	-9.9951	volts
NIM	accz2	-9.9976	volts
NIM	accz3	+0.0000	volts

Fig 3.3.1 Screen with real time values



Package	Field	Value	Unit
NIM	accz3	-9.9976	volts
NIM	nim_5v	+4.8926	volts
NIM	cor	+0	
NIM	accz1	-9.9902	volts
NIM	accz2	-9.9976	volts
NIM	accz3	+0.0000	volts

Fig 3.3.2 Values updated at run time

3.4 Real Time Graph

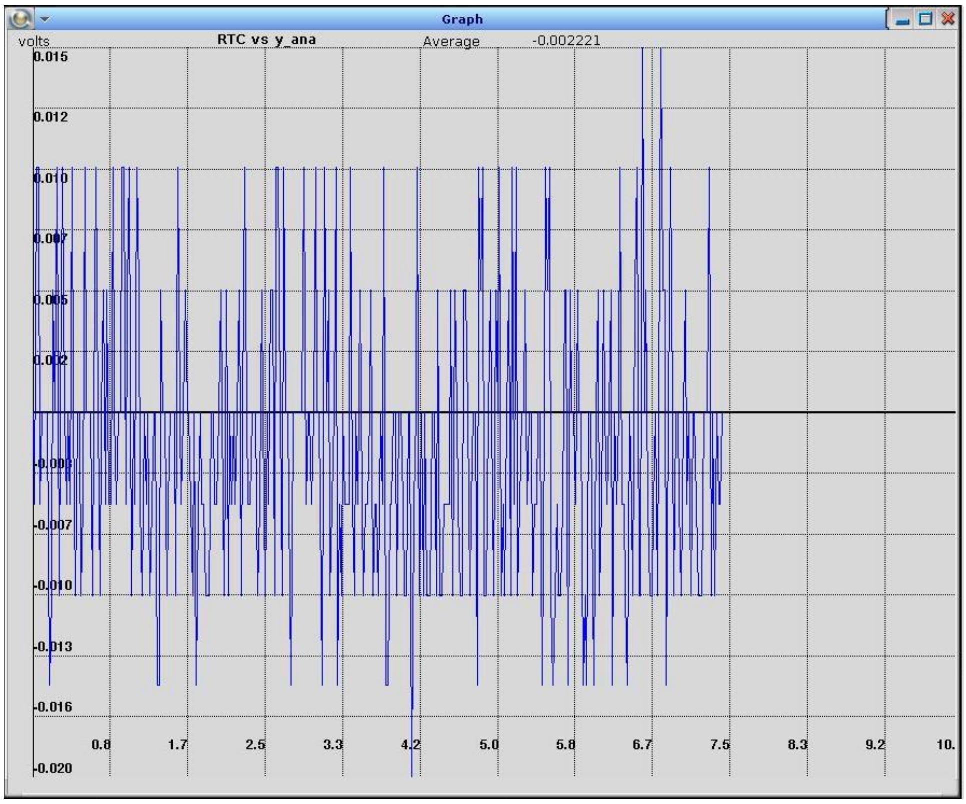


Fig 3.4.1 Real Time Graph

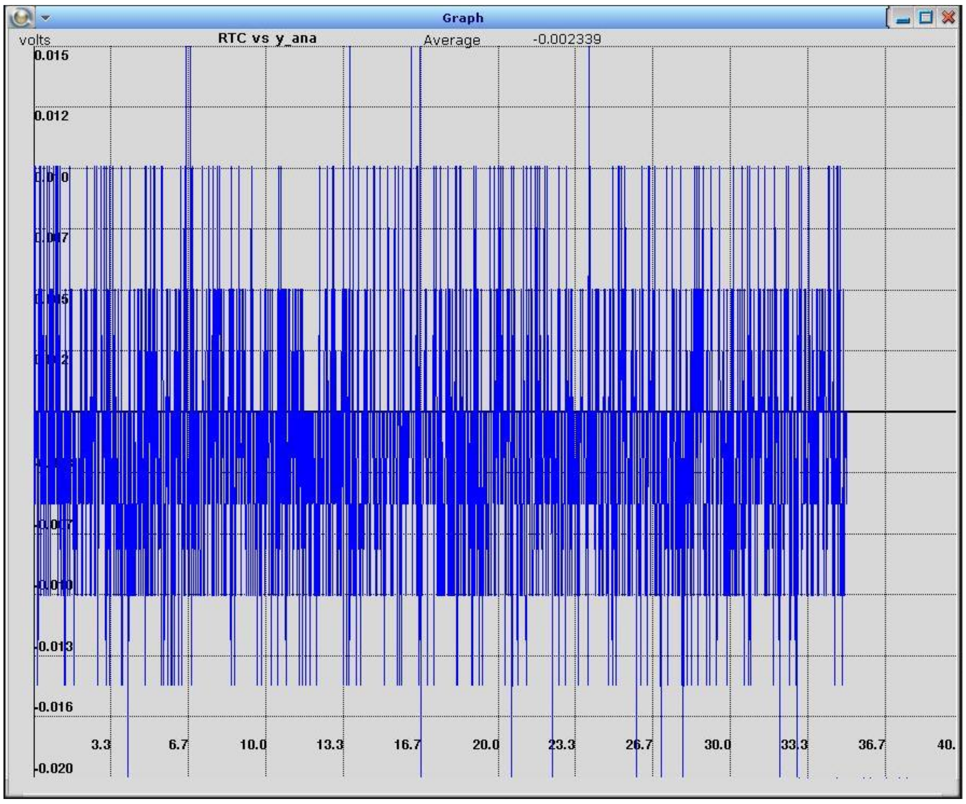


Fig 3.4.2 Real Time Graph over a long time

Code

1. Algorithm
    1. Data type conversion
    2. Graph plotting
2. Sample code

1. ALGORITHMS USED

The way server and client work have been discussed in detail in previous sections. Discussion of their working as algorithm will be both voluminous and repetitive. Hence that is avoided.

Algorithms discussed below are the ones, devised by us, that pertain to solving specific problems within client and server.

1.1 Data conversion

Telemetry data is a series of bytes. From the telemetry data, the bytes that corresponding to a parameter can be obtained. Using those bytes the value of the parameter has to be calculated. The algorithm that is used for the same is discussed below.

A few data types used by NIM and NGCP are

- Integer
- Float
- Hexadecimal

```
Algorithm convert(input, num_bytes, scale, ptype,
                  normalising_factorising_factor)
    Generates the value of the parameter that corresponds to the passed
value in engineering units
begin
    set return_value as 0
    set value as get_long(input, num_bytes)

    switch ptype
        case FLOATING:
                return_value = convert_to_float(value,
                               normalising_factorising_factor,
                               scale)
                break
        case INTEGER:
                return_value = convert_to_integer(value,
                               normalising_factorising_factor,
                               scale)
                break
        case COUNT:
                return_value = convert_to_integer(value,
                               normalising_factorising_factor,
                               scale)
                break
        case FLAG:
                return_value = convert_to_flag(value,
                               normalising_factorising_factor,
                               scale)
                break
    end switch
    return return_value
end convert
```

```
Algorithm get_long(input, num_bytes)
    Generates the raw value that corresponds to the given bytes of data
begin
    set num as 0
    set i as num_bytes - 1
    while i >= 0
        do
        num |= input[i]
        if( i != 0)
            num = num << 8
        i--
    end while
    return num
end get_long
```

```
Algorithm convert_to_float(value, normalising_factor, scale)
    Converts the raw value into a floating point value
begin
    set output as 1
    if normalising_factor equals 0.0
        set output as value * scale
    else if normalising_factor equals 1.0
        if value > 32767
            set output as ( -1 ) * ((value * scale) / pow(2,16))
        else
            set output as ((value * scale) / pow(2,15))
        end if
    end if
    return output
end convert_to_float
```

```
Algorithm convert_to_integer( value, normalising_factor, scale)
    Converts the raw value into a integral value
begin
    set output as 1
    if normalising_factor equals 0.0
        set output as value * scale
    else
        set output as value
    return output
end convert_to_integer
```

1.2 Graph plotting algorithm

The following algorithm deals with the mapping the (RTC, input values) pair to corresponding point on the graph window. The input value can be output of any sensor onboard. RTC, input value pairs are got for every 20 milliseconds of RTC.

```
while new input is there
    add the input to the array
    update the average
    if input's x co-ordiante >= max_x
        update max_x
    end if
    if input's y co-ordiante >= max_y
        update max_y
    else if input's y co-ordiante < min_y
        update min_y
    end if
    rescale(point)
end while
```

Algorithm rescale(input)

```
    Global variables - max_x, max_y, low_x, low_y : denoting the maximum and minimum x and y
                       co ordinates
                        x_axis, y_axis : denotes the position of x and y axis on the graph
                       window
                        width and height are the dimensions of the window in which graph has to
                       be plotted
    Input -            has the x and y value corresponding to the input got
    Output             Point that has to be plotted on the screen.
                       it denotes the point on the screen to which the input values correspond
```

```
begin
set x as width / max_x
    if low_y < 0 and max_y > 0
        set ly as height / low_y
        set my as height / max_y
        point-> x = input-> x * x + x_axis
        if point-> y < 0
            point-> y = input-> y * ly + y_axis
        else
            point-> y = y_axis - input-> y * my
        end if
    else if max_y <= 0
        set ly = height / low_y
        point-> x = input-> x * x + x_axis
        point-> y = input-> y * ly + y_axis
    else if low_y >= 0
        set ly = height / max_y
        point-> x = input-> x * x + x_axis
        point-> y = y_axis - input-> y * ly
    end if
end rescale
```

2. SAMPLE CODE

Since the project is software designed for use of ISRO and the lines of code of server and client combined is in the range of few thousands, spread across a lot of files, a small part of the code is presented as sample.

The class shown below is the class that manages plotting of data on the graph. The class keeps store of the values to be plotted, calculates the point corresponding to them and rescales the points when the graph window is re-sized. It also manages the x and y axes scale.

```
class graph
{
    private:
            int point_count;
            float low_x, max_x;
            float low_y, max_y;
            float sum;

            float x_point[13];
            float y_point[12];
            float x_point_pos[13];
            float y_point_pos[12];
            float x_axis, y_axis;

            point_det act_point[7000];
            point_det cur_point[7000];

            int redraw;
            int width, height;

            void fix_units();
            void scale_points();
            void graph_reset();

    public:
            graph();
            graph(float a, float b);
            ~graph();

            void add_point(float a, float b);
            point_det* get_points();
            point_det get_points(int pos);
            int no_points();
            float get_average();

            float* get_x_point();
            float* get_y_point();
            float* get_x_point_pos();
            float* get_y_point_pos();

            int redraw_needed();
            void set_axes(float a, float b);
            void set_dim_graph(int a, int b);

            int get_x_axis();
            int get_y_axis();
            int get_width_axis();
            int get_height_axis();
            void set_redrawn();
};


/*
 * point.cpp
 *
 *  Created on: Feb 29, 2012
 *      Author: user
 */
#include "point.hpp"
#include <stdlib.h>
```

```
void graph::add_point(float a, float b)
{
    act_point[point_count].x = a;
    act_point[point_count].y = b;
    sum += b;
    point_count ++;

    if(point_count == 7000)
    {
        graph_reset();
        return;
    }

    if( a >= max_x )
    {
        redraw = 1;
        max_x += 5;
        return;
    }

    if( b < low_y )
    {
        redraw = 1;
        low_y = b;
        return;
    }
    else if ( b > max_y )
    {
        redraw = 1;
        max_y = b;
        return;
    }
}

graph::graph(float a, float b)
{
    point_count = 0;
    x_axis = a;
    y_axis  = b;

    height = 210;
    width = 482;

    max_x = 5;
    max_y = -100;
    low_x = 0;
    low_y = 100;

    redraw = 0;
    sum = 0;
}

graph::graph()
{
    point_count = 0;
}

graph::~graph()
{

}

void graph::scale_points()
{
    float x = width / max_x;
    if(low_y < 0 && max_y > 0)
    {
        float ly = height / ( low_y );
        float my = height / ( max_y );
        for(int i = 0; i < point_count ; i++)
        {
            cur_point[i].x = act_point[i].x * x;
            cur_point[i].x += x_axis;
            if( act_point[i].y < 0 )
            {
                cur_point[i].y = act_point[i].y * ly;
                cur_point[i].y += y_axis;
            }
            else
            {
                cur_point[i].y = act_point[i].y * my;
                cur_point[i].y = y_axis - cur_point[i].y ;
            }
        }
    }

    else if(max_y <= 0)
    {
        float ly = height / ( low_y );
        for(int i = 0; i < point_count ; i++)
        {
            cur_point[i].x = act_point[i].x * x;
            cur_point[i].x += x_axis;
            cur_point[i].y = act_point[i].y * ly;
            cur_point[i].y += y_axis;
        }
    }

    else if(low_y >= 0)
    {
        float ly = height / ( max_y );
        for(int i = 0; i < point_count ; i++)
        {
            cur_point[i].x = act_point[i].x * x;
            cur_point[i].x += x_axis;
            cur_point[i].y = act_point[i].y * ly;
            cur_point[i].y = y_axis - cur_point[i].y ;
        }
    }
}

point_det* graph::get_points()
{
    if(redraw == 1)
        fix_units();
    scale_points();
    return cur_point;
}

point_det graph::get_points(int pos)
{
    scale_points();
    return cur_point[pos];
}

int graph::no_points()
{
    return point_count;
}

float* graph::get_x_point()
{
    return x_point;
}

float* graph::get_y_point()
{
    return y_point;
}

void graph::fix_units()
{
    int i;
    float x_inter = (max_x) / 12;
    float x = width / max_x;
    for(int i = 0 ; i < 13; i++)
    {
        x_point[i] = (i) * x_inter;
        x_point_pos[i] = x_point[i] * x;
        x_point_pos[i] += x_axis;
    }

    if(low_y < 0 )
    {
        float ly = height / ( 0 - low_y );
        float y_inter = ( 0 - low_y) / 6;
        for(i = 0 ; i < 6; i++)
        {
            y_point[i + 6] = (- 1) * (i + 1 ) * y_inter;
            y_point_pos[i + 6] = (-1) * y_point[i + 6] * ly ;
            y_point_pos[i + 6] += y_axis;
        }
    }
    if(max_y > 0)
    {
        float my = height / ( max_y );
        float y_inter = (max_y) / 6;
        for(i = 0 ; i < 6; i++)
        {
            y_point[i] = (i + 1) * y_inter;
            y_point_pos[i] = y_point[i] * my;
            y_point_pos[i] = y_axis - y_point_pos[i] ;
        }
    }
}

int graph::redraw_needed()
{
    return redraw;
}

void graph::set_axes(float a, float b)
{
    x_axis = a;
    y_axis = b;
    redraw = 1;
}

void graph::set_dim_graph(int a, int b)
{
    width = a;
    height = b / 2;
    redraw = 1;
}

int graph::get_x_axis()
{
    return x_axis;
}

int graph::get_y_axis()
{
    return y_axis;
}

int graph::get_width_axis()
{
    return width;
}

float *graph::get_x_point_pos()
{
    return x_point_pos;
}

float *graph::get_y_point_pos()
{
    return y_point_pos;
}

int graph::get_height_axis()
{
    return height * 2;
}

float graph::get_average()
{
    return (sum / point_count);
}

void graph::graph_reset()
```

```
{
    point_count = 0;

    max_x = 150;
    max_y = -100;
    low_x = 0;
    low_y = 100;

    redraw = 0;
    sum = 0;

    for(int i = 0; i < 7000; i++)
    {
        cur_point[i].x = 0;
        cur_point[i].y = 0;
        act_point[i].x = 0;
        act_point[i].y = 0;
    }
}

void graph::set_redrawn()
{
    redraw = 0;
}
```

Bibliography

- MIL-STD-1553 tutorial, Condor engineering
- MIL 1553 bus protocol
- Anderson, Paul and Anderson, Gail. Navigating C++ & Object Oriented Design
- Krten, Rob. Getting Started with QNX Neutrino 2 : A Guide for Real Time Programmers
- Teqsite, Indus. Driver User Manual : Dual Channel MIL-STD-1553 BC/RT/MT CPCI Intelligent Module
- Volume 1, 2nd edition. QNX Watcom C Library Ref. Watcom International Corporation
- BUS – 69083, 69082, 69080 s/w Manual. Ace Runtime Library

Appendix

1. Posix calls
    1. Photon library calls
    2. Bus interface card calls
    3. Thread library calls
    4. Other callls

1. POSIX CALLS

1.1 Photon library calls

1. **PtInit ()** : Initialize the widget library
   int PtInit(char const *name);

2. **PtFileSelection ()** : Create a file-selection dialog
   int PtFileSelection (PtWidget_t *parent,
   PhPoint_t const *pos, char const *title, char const *root_dir, char const *file_spec,
   char const *btn1, char const *btn2, char const *format, PtFileSelectionInfo_t *info, int flags);

3. **PtSetResource ()** : Set one resource for a widget
   #define PtSetResource (widget, type, value, len) …

4. **PtGetResource ()** : Retrieve one resource value for a widget
   #define PtGetResource (widget, type, value, len) …

5. **PtNotice () :** Display a message and wait for acknowledgement
   void PtNotice (PtWidget_t *parent, PhPoint_t const *location, char const *title,
   PhImage_t const *image, char const *message, char const *msgFont, char const *btnText,
   char const *btnFont, int flags);

6. **PtListAddItems ()** : Add items to a list
   int PtListAddItems (PtWidget_t *widget, const char **items, int item_count, unsigned int position);

7. **PtListDeleteAllItems ()** : Delete all items from a list
   int PtListDeleteAllItems (PtWidget_t *widget);

8. **PtListDeleteItemPos ()** : Delete a range of items from a list
   int PtListDeleteItemPos (PtWidget_t *widget, int item_count, int position);

9. **PtListSelectPos ()** : Select the item at a given position
   void PtListSelectPos (PtWidget_t *widget, int pos);

10. **PtListUnselectPos ()** : Unselect the item at the given position
    void PtListUnselectPos (PtWidget_t *widget, int pos);

11. **PtListShowPos ()** : Show the item at the given position
    void PtListShowPos (PtWidget_t *widget, int pos);

12. **PtGiveFocus ()** : Give focus to a widget
    PtWidget_t *PtGiveFocus (PtWidget_t *widget, PhEvent_t *event);

13. **PtBkgdHandlerProcess ()** : Process all outstanding Photon events
    void PtBkgdHandlerProcess (void);

14. **PtDestroyWidget ()** : Remove a widget from the widget family hierarchy
    int PtDestroyWidget (PtWidget_t *widget);

1.2 Bus interface card calls

1. **DP566_DetectCards ()** : Detect number of cards, and number of channels for each card detected
```
#include <dp566.h>
int DP566_DetectCards ( unsigned short *out_usCards );
```

2. **DP566_DetectChannels ()** : Detect the number of dual channel cards
```
#include <dp566.h>
int DP566_DetectChannels ( unsigned short in_usCardNo, unsigned short *out_usChannels );
```

3. **DP566_OpenHandle ()** : Open the handle for the specified board number and return the handle to the user
```
#include <dp566.h>
int DP566_OpenHandle  (unsigned short in_usCardNo, unsigned short in_usChannelNo,
           unsigned short *out_usDevHandle );
```

4. **DP566_GetVersion ()** : Gives the version number of the firmware and the driver
```
#include <dp566.h>
int DP566_GetVersion ( unsigned short in_usDevHandle, float *out_fFirmVer, float *out_fDrvVer );
```

5. **DP566_Reset ()** : Reset the DSP
```
#include <dp566.h>
int DP566_Reset ( unsigned short in_usDevHandle );
```

6. **DP566_SetMode()** : Select the mode to activate as BC/RT/MT
```
 #include <dp566.h>
 #include <dp566var.h>
int DP566_SetMode ( unsigned short in_usDevHandle, unsigned short in_usMode );
```

7. **DP566_AttachInterrupt ()** : Attach interrupt handler to DP566 interrupt
```
 #include <dp566.h>
 #include <dp566var.h>
int DP566_AttachInterrupt ( unsigned short in_usDevHandle, pid_t *in_irq_proxy );
```

8. **DP566_SetRespTimeout ()** : Update the response timeout value setted
```
 #include <dp566.h>
 #include <dp566var.h>
int DP566_SetRespTimeout  (unsigned short in_usDevHandle, unsigned short in_usRespTimeout );
```

9. **DP566_BMEnableRT ()** : Enable the RTs to be noted
```
#include <dp566.h>
#include <dp566var.h>
int DP566_BMEnableRT ( unsigned short in_usDevHandle, unsigned short *in_ usRTSelTbl );
```

10. **DP566_SetBlockInt ()** : Update the time tag register with the specified value
```
 #include <dp566.h>
 #include <dp566var.h>
int DP566_SetBlockInt ( unsigned short in_usDevHandle, unsigned short in_usNoOfMsg );
```

11. **DP566_SetTimeTagResl ()** : Update the time tag register with the specified value
```
#include <dp566.h>
#include <dp566var.h>
int DP566_SetTimeTagResl ( unsigned short in_usDevHandle, unsigned short in_usTimeTagVal );
```

12. **DP566_EnaDisInterrupt ()** : Mask or unmask the ACE controller interrupts to the host
```
 #include <dp566.h>
 #include <dp566var.h>
int DP566_EnaDisInterrupt ( unsigned short in_usDevHandle, unsigned short in_usIntRegVal );
```

13. **DP566_DetachInterrupt ()** : Detach interrupt handler

```
#include <dp566.h>
#include <dp566var.h>
int DP566_AttachInterrupt ( unsigned short in_usDevHandle );
```

14. **DP566_CloseHandle ()** : Close the specified handle

```
#include <dp566.h>
int DP566_OpenHandle ( unsigned short *out_usDevHandle );
```

15. **DP566_BMStartStop ()** : Start or stop the transmission in BC

```
#include <dp566.h>
#include <dp566var.h>
int DP566_BCStartStop  (unsigned short in_usDevHandle, unsigned short in_usStrtStop );
```

16. **DP566_GetErrMsg ()** : Returns the error string for the given error code

```
#include <dp566.h>
int DP566_GetErrMsg ( int usErrCode, char *ErrMsg );
```

1.3 Threads

1. **pthread_create() : T**hread Creation

```
#include <pthread.h>
int pthread_create( pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void* ),
        void* arg);
```

2. **pthread_self() :** getting thread ID

```
#include <pthread.h>
pthread_t pthread_self( void );
```

3. **pthread_join()** : Thread Join

```
#include <pthread.h>
int pthread_join( pthread_t thread, void** value_ptr );
```

4. **pthread_attr_init() :** Initialize Thread Attribute Object

```
#include <pthread.h>
int pthread_attr_init( pthread_attr_t *attr );
```

5. **pthread_attr_setschedpolicy() :** Set thread scheduling policy

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setschedpolicy( pthread_attr_t* attr, int policy );
```

6. **Structures**

```
#include <sched.h>
struct sched_param {
        int32_t  sched_priority;
        int32_t  sched_curpriority;
        union {
                int32_t  reserved[8];
                struct {
                        int32_t  __ss_low_priority;
                        int32_t  __ss_max_repl;
                        struct timespec    __ss_repl_period;
                        struct timespec    __ss_init_budget;
                }       __ss;
        }       __ss_un;
}

#define sched_ss_low_priority   __ss_un.__ss.__ss_low_priority
#define sched_ss_max_repl       __ss_un.__ss.__ss_max_repl
#define sched_ss_repl_period    __ss_un.__ss.__ss_repl_period
#define sched_ss_init_budget    __ss_un.__ss.__ss_init_budget
```

1.4 Other calls

1. **spawnlp ()** : Spawn a child process, give a list of arguments and a relative path
   ```
   #include <process.h>
   int spawnlp ( int mode, const char *file, const char *arg0,  const char *arg1… ,  const char *argn, NULL );
   ```

2. **mkfifo ()** : Create a FIFO special file
   ```
   #include <sys/types.h>
   #include <sys/stat.h>
   int mkfifo ( const char *path, mode_t mode );
   ```