Kumaraguru College of Technology

Department of Computer Science and Engineering
Coimbatore-641 006

ISO 9001:2000

INTERFACING REALTIME OPERATING SYSTEM WITH NORMAL OPERATING SYSTEM

Project work done at

SRJ INFOJNANA SYSTEM Pvt. Ltd.
COIMBATORE.

PROJECT REPORT

Submitted in partial fulfillment of the

Requirements for the award of the degree of

Master of Science in Applied Science

Software Engineering

Bharathiar University, Coimbatore

Submitted by

L.K.SAKTHIVEL
Reg.No-0037S0099

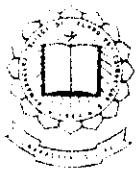
INTERNAL GUIDE

Mr.K.R.Baskaran.B.E.,M.S
Dept.of Computer Science& Engineering,
Kumaraguru College of Technology,
Coimbatore.

EXTERNAL GUIDE

Mr.S.Ramajayam B.E,
SRJ INFOJNANA SYSTEM Pvt. Ltd,
COIMBATORE.

CERTIFICATE



Department of Computer Science and Engineering

KUMARAGURU COLLEGE OF TECHNOLOGY

Coimbatore – 641 006

CERTIFICATE

PROJECT REPORT 2003

Certified that this is a bonafide report of
the project work done by

L.K.SAKTHIVEL
(Reg. No. 0037S0099)

Mr.K.R.Baskaran.B.E.,M.S
Project guide
Computer Science & Engineering

Prof. S. Thangasamy , Ph.D.,
Head of the Department
Computer Science & Engineering

Place: Coimbatore
Date: 25.09.03

Submitted for viva-voce examination held on

29.09.03

Internal Examiner

External Examiner

COMPANY CERTIFICATE

INFOGNANA

SRJ INFO JNANA SYSTEM (Pr) Ltd.,
COIMBATORE STOCK EXCHANGE BUILDING,
683 - 686, TRICHY ROAD, 3rd FLOOR,
COIMBATORE - 641 005
Ph 2320067, 2320085
e-mail info@infognana.com

September 24, 2003

To Whomsoever It May Concern

This is to certify that Mr. L.K.Sakthivel of Kumaraguru College Of Technology, Coimbatore has done a project with us on **"Interfacing Realtime Operating System with Normal Operating System "** for SRJ Info Jnana System (Pr) Ltd from June 2003 to Sep 2003.

He has successfully completed his project and was punctual and dedicated.

For **SRJ INFOGNANA SYSTEM (Pr) LTD**

Babu Unnikrishnan
Chief Executive – Developments

DECLARATION

DECLARATION

I here by declare that the project entitled **"INTERFACING REAL TIME OPERATING SYSTEM WITH NORMAL OPERATING SYSTEM"**, submitted to Kumaraguru College of Technology, Coimbatore Affiliated to Bharathiar university as the project work of Master of Science in Applied Science Software Engineering ,is a record of original work done by me under the supervision and guidance of **Mr.Ramajayam.B.E,** SRJ INFOJNANA SYSTEMS Pvt Ltd., Coimbatore and **Mr.K.R.Baskaran.B.E.,M.S.,** CSE Department Kumaraguru College of Technology, Coimbatore and the project work has not found the basis for the award of any Degree/ Diploma / Associate ship / Fellowship or similar title to any candidate of any University.

Place: Coimbatore

Date: 25.09.03

(L.K.SAKTHIVEL)

Reg.No:0037S0099

Countersigned by

(Internal Guide)

Mr.K.R.Baskaran.B.E.,M.S.,

Kumaraguru College of Technology,

Coimbatore.

ACKNOWLEDGEMENT

ACKNOWLEDGEMENT

I am immensely grateful to **Dr.K.K.Padmanaban BSc(Engg) , M.Tech., Ph.D.,** Principal , Kumaraguru College of Technology for his valuable support to come out with this project.

I really feel delighted in expressing my heartful thanks to **Dr.S.Thangaswamy Ph.D**, Prof & Head of Department of Computer Science and Engineering for his endless encouragement in carrying out this project successfully.

My heartfelt thanks to our project coordinator **Mrs.S.Devaki B.E., M.S,** Assistant Professor, for his unfailing enthusiasm, encouragement and guidance that paved me to the completion of this project.

I am indent to express my heartiest thanks to **Mr.K.R.Baskaran.B.E.,M.S,** my project guide who rendered his valuable guidance and support to do this project work extremely well.

I am greatly indebted to **Mr.V.Paramasivam** chairman and **Mr.Babu** chief executive, INFOGNANA (P) Ltd, Coimbatore, for getting us into his esteemed institution. I also thank **Mr.Ramajayam** B.E who was my guide and he has helped me a lot in my project.

I am also thankful to all the faculty members of the Department of Computer Science and Engineering, Kumaraguru College of Technology, Coimbatore for their valuable guidance, support and encouragement during the course of my project work..

My humble gratitude and thanks to my parents who have supported, to complete the project and to my friends, for lending me valuable tips, support and cooperation through out my project work.

SYNOPSIS

SYNOPSIS

The goal underlying this project is to achieve an effective interface between Real Time Operating System and normal Operating System. The interface is based on well-known paradigm, which is used for most of the distributed applications client/server model. The interfacing is for file transfer and message transfer between these two operating systems. Depending on the application, the normal operating system may act as client or server.

A normal Operating System is a system program that provides an interface between the application programs with system (hardware). A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time in which the result is produced. If the timing constraints are not met, failure is said to have occurred. Real Time Operating System, usually abbreviated as RTOS is an operating system that guarantees an application's specific response time to specific stimuli.

By interfacing, the timeliness of RTOS can be effectively utilized for the normal operating system. The latency time of RTOS is less (6.5 micro seconds) than that of normal operating System. To handle multiple external events occurring simultaneously, pseudo parallelism is present. There is Rate Monotonic Scheduling to compute the processor power required for handling events in advance.

CONTENTS

TABLE OF CONTENTS

PAGE #

INTRODUCTION

1. INTRODUCTION

1.1 PROBLEM DEFINITION

The objective of this project is to enable an effective communication between the normal Operating System and Real Time Operating System by interfacing them.

A normal Operating System is a system program that provides an interface between the application programs with system (hardware). A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time in which the result is produced. If the timing constraints are not met, failure is said to have occurred.

The communication is either in the form of message transfer or in the form of file transfer. The interface is based on well-known paradigm, client/server model. Windows, a commonly used operating system is considered as client and Proprietary Silicon Operating System, usually abbreviated as pSOS act as server.

Use of socket does the communication. The socket that is used for the application is a stream socket. Stream sockets provide for a data flow without record boundaries - a stream of bytes. Streams are guaranteed to be delivered and to be correctly sequenced and unduplicated. Receipt of stream messages is guaranteed, and streams are well suited for handling large volumes of data. The communication is done in connection-oriented style by TCP.

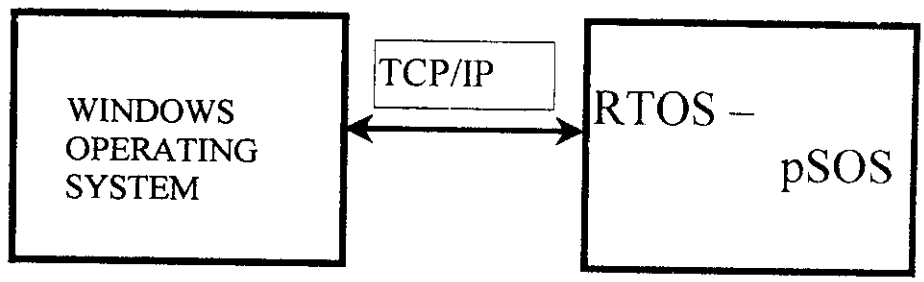
1.2 ORGANAISATION PROFILE

SRJ Infognana systems (Pr.) Ltd., is primarily a software development company with its prime focus on embedded systems. The company has been promoted in the year 2000 by NRIs from USA and UK. The promoters have more than 8 years experience in this industry. Infognana has an experienced team of professionals to provide Embedded, ERP, E-COM software solutions. With its headquarters in Coimbatore, India and a liaison office in Dallas, Texas U.S.A, it has registered a steady growth since its inception and has always been reckoned as a reliable provider of the latest Information Technology services.

Infognana specializes as a software service provider covering a broadband spectrum of technologies, with focus on emerging technologies. It provides embedded software solutions for real-time applications, ERP solutions for small and medium scale enterprises and E-commerce solutions for B2B and B2C applications.

Infognana also provide custom – built solutions as per the client's specifications.

1.3 SYSTEM DIAGRAM



(client) **(server)**

Fig 1.1 System Diagram

Fig 1.1 shows the outline of the system. Here the windows is considered as client and pSOS as server and both are connected with TCP/IP. According to the application it can be act as a client or server.

SYSTEM ANALYSIS

2.0 SYSTEM ANALYSIS

2.1 EXISTING SYSTEM:

The existing system has only interfacing between same operating system and it won't share facilities like multitasking, priority preemption. It has less interrupt latency time, normal memory capacity with in a single interfacing, so we go for proposed system.

And we previously do interfacing real time os with normal os with the help of Visual Basic. In Visual Basic coding is hard to implement comparing to VC++ because VC++ is more advanced than Visual Basic. So we go for the proposed system.

2.2 PROPOSED SYSTEM:

To share the advantage of both Real Time OS and normal OS and to have effective communication between them we had interfaced it through TCP (connection oriented protocol). And we transfer message and file from RTOS to normal OS and vice versa according to our application.

With the help of this project we can allow the task to set priority and according to our need we can choose either message to transfer or file to transfer from client to server with guaranteed delivery and acknowledgement.

System analysis is conducted with the following objectives in mind:

- Identifying the customer needs,
- Evaluation of the system concept for feasibility,
- Perform economical analysis,
- Allocate function to hardware, software, people, database, and other system elements
- Establish cost and schedule constraints
- Create system definition that forms foundation for all subsequent engineering work.

SOFTWARE REQUIREMENTS

3.0 SOFTWARE REQUIREMENTS

3.1 HARDWARE SPECIFICATIONS

CPU	- Pentium III 1.5GHz
VGA Card	- 32MB PCI/AGP
HDD	- 20GB
RAM	- 128MB
DISPLAY	- 15" SVGA Color Monitor
NETWORK	- 10/100Mbps PCI Ethernet Card

3.2 SOFTWARE SPECIFICATIONS

OPERATING SYSTEM	- Windows95, Windows NT
REALTIMEOPERATINGSYSTEM	- pSOS
TOOLS	- Visual C++ 6.0

3.3 WINDOWS PROGRAMMING

CONCEPT OF SOCKET

In simple, Socket is an end-point of communication. Once a socket has been created, it can be used to wait for an incoming connection or to initiate a connection. A socket used by a server to wait for an incoming connection is known as a Passive Socket, while a socket used by a client to initiate a connection is called as Active Socket. The only difference between these sockets lies in how applications use them; the sockets are created the same way initially.

Two socket types are available:

- **Stream sockets**

Stream sockets provide for a data flow without record boundaries - a stream of bytes. Streams are guaranteed to be delivered and to be correctly

sequenced and unduplicated. Receipt of stream messages is guaranteed, and streams are well suited for handling large volumes of data. Streams are based on explicit connections, socket A requests a connection to socket B and socket B accepts or rejects the connection request.

- **Datagram sockets**

Datagram sockets support a record-oriented data flow that is not guaranteed to be delivered and may not be sequenced as sent or unduplicated. Datagram also are not guaranteed to be reliable, they can fail to arrive. Datagram data may arrive out of order and possibly duplicated, but record boundaries in the data are preserved, as long as the records are smaller than the receiver's internal size limit. Datagram are "connectionless" - no explicit connection is established.

Stream sockets are preferable to Datagram sockets when the data must be guaranteed to arrive and when data size is large.

When a socket is created, it does not contain any detailed information about how it will be used. In particular, the socket does not contain information about the protocol port numbers or IP addresses of either the local machine or the remote machine. According to the protocol families, the definition of the endpoint address may vary. To allow protocol families the freedom to choose representations for their addresses the socket abstraction defines an address family for each type of address. A protocol family can use one or more address families to define address representations.

A Generic Address Structure:

The Socket System defines a generalized format that all endpoint addresses use.

The generalized format consists of a pair is

(address family, endpoint address in the family)

Where **address family** field contains a constant that denotes one of the preassigned address types. **Endpoint address** field contains an endpoint address

using the standard representation for the specified address type.

```
struct sockaddr              /* structure to hold an address */
{                    u_short sa_family;    /* type of address */
          charsa_data[14];          /*   value of address*/
};
```

Each protocol family that uses sockets defines the exact representation of its endpoint addresses, and the socket software provides corresponding structure declarations. Each TCP/IP endpoint address consists of a 2-byte field that identifies the address type (AF_INET), a 2-byte port number field, a 4-byte IP address field, and an 8-byte field that remains unused.

Predefined **structure Sockaddr**_in specifies the format:

```
struct sockaddr_in              /* struct to hold an address */
{
        u_short sin_family;   /*  type of address */
        u_short sin_port;             /*protocolport  number */
        u_long sin_addr;              /* IP address */
        char sin_zero[8];             /* unused (set to zero)*/
};
```

WINDOWS SOCKETS

The Windows Socket Specification defines a binary-compatible network-programming interface for Microsoft Windows. It encompasses both familiar Berkeley socket style routines and a set of Windows-specific extensions designed to allow the programmer to take the advantage of the message-driven nature of Windows.

The Windows Socket is created using Microsoft Foundation Class library, usually abbreviated as MFC, are a set of predefined classes upon which Windows programming with is built. These classes represent an object-oriented approach to Windows programming that encapsulates the Windows API. The process of writing programs involves creating ad using MFC objects, or objects of classes

derived from MFC.

An MFC program is an executable application for Windows that is based on the Microsoft Foundation Class (MFC) Library. The Microsoft Foundation Class Library (MFC) is an "application framework" for programming in Microsoft Windows. Written in C++, MFC provides much of the code necessary for managing windows, menus, and dialog boxes; performing basic input/output; storing collections of data objects; and so on. To provide the application-specific functionality the data and function members that customize the classes are added.

The two main classes from MFC are an application class, CWinApp and a window class, CFrameWnd.The class CWinApp is fundamental to any Windows program written using MFC. An object of this class includes everything necessary for starting, initializing and closing the application. So, to produce an application a class should be derived from CWinApp. CFrameWnd, the window class provides a window as interface to the user. Since CFrameWnd provides everything for creating and managing a window for application, the derived class constructor is to be added.

The structure of an MFC program incorporates two application-oriented entities: *a document and a view.* A document is the name given to the collection of data in the application with which the user interacts. The 'document' implies not only of textual nature, but also the data for game, a geometric model or anything. The document of an application is derived from the class Cdocument in the MFC library.

The data members that the application requires and the member functions to support processing of that data can be added. A view relates to a particular document object. A view is a mechanism for displaying some or all of the data stored in a document. It defines how the data is to be displayed in a window and how the user can interact with it. The view of an application can be derived from Cview class. A document object can have multiple view objects associated with it. Each view object can provide a different presentation or subset of the same document data.

The Microsoft Foundation Class Library (MFC) Supports programming with the Windows Sockets API by supplying two classes. They are:

- **CAsyncSocket**

This class encapsulates the Windows Sockets API. CAsyncSocket is for programmers who want the flexibility of programming directly to the sockets API but also want the convenience of callback functions for notification of network events. Other than packaging sockets in object-oriented form for use in C++, the only additional abstraction this class supplies is converting certain socket-related Windows messages into callbacks.

- **CSocket**

This class, derived from **CAsyncSocket**, supplies a higher-level abstraction for working with sockets. Using a socket with an archive greatly resembles using MFC's file serialization protocol. This makes it easier to use than the **CAsyncSocket** model. CSocket inherits many member functions from **CAsyncSocket** that encapsulate Windows Sockets APIs.

Since, the application uses Stream Socket(Fig 2.1), CSocket class is used.

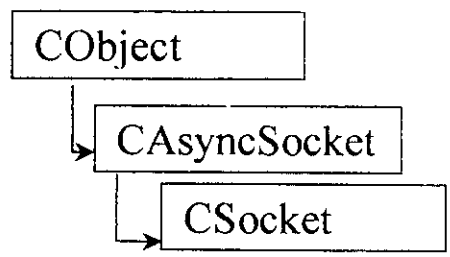


Figure 2.1 Stream Socket

To use a CSocket object, the CSocket( ) constructor is called. After constructing a socket object, Windows socket is created by create( ) and attach it. Then it calls bind( ) to bind the socket to the specified address.Since the socket is a client, the base class member CasyncSocket::Connect () is called to connect the socket object to a server socket.

a) Sequence of Operations for a Stream Socket Communication

Up to the point of constructing a **CSocketFile** object, the following sequence is accurate (with a few parameter differences) for both **CAsyncSocket** and **CSocket**. From that point on, the sequence is strictly for **CSocket**. The following table illustrates the sequence of operations for setting up communication between a client and a server.

pSOS - Proprietary Silicon Operating System

pSOSystem is a modular, high-performance real-time operating system designed specifically for embedded microprocessors. It provides a complete multitasking environment based on open systems standards. pSOSystem is designed to meet three overriding objectives:

- Performance
- Reliability
- Ease – of – Use

The pSOSystem software employs a modular architecture. It is built around the pSOS+ real-time multi-tasking kernel and a collection of companion software components. Software components are standard building blocks delivered as absolute position - independent code modules.

They are standard parts in the sense that they are unchanged from one application to another. Each software component utilizes configuration table that contains application and hardware related parameters to configure itself at startup.

System Architecture:

The pSOSystem software employs a modular architecture. It is built around the pSOS+ real-time multi-tasking kernel and a collection of companion software components. Software components are standard building blocks delivered as absolute position-independent code modules. They are standard parts in the sense that they are unchanged from one application to another.

This black box technique eliminates maintenance by the user and assures reliability, because hundreds of applications execute the same, identical code. Unlike most system software, a software component is not wired down to a piece of hardware. It makes no assumptions about the execution/target environment. Each software component utilizes a user-supplied configuration table that contains application and hardware-related parameters to configure itself at startup.

Every component implements a logical collection of system calls. To the application developer, system calls appear as re-entrant C functions callable from an application. Any combination of components can be incorporated into a system to match the real-time design requirements.

The pSOSystem components are:

- pSOS+ Real-time Multitasking Kernel
- pSOS+m Multiprocessor Multitasking Kernel
- pNA+ TCP/IP Network Manager
- pRPC+ Remote Procedure Call Library
- pHILE+ File System Manager
- pREPC+ ANSI C Standard Library

pSOS+ kernel is a real-time, multitasking operating system kernel. It maintains a highly simplified view of application software, irrespective of the application's inner complexities.

To the pSOS+ kernel, applications consist of three classes of program elements:

- Tasks
- I/O Device Drivers
- Interrupt Service Routines (ISRs)

A task is the smallest unit of execution that can compete on its own for system resources. A task lives in a virtual, insulated environment furnished by the pSOS+ kernel. Within this space, a task can use system re-sources or wait for them to become available. if necessary, without explicit concern for other tasks.

Resources include the CPU, I/O devices, memory space, and so on. Conceptually, a task can execute concurrently with, and independent of. other tasks. The pSOS+ kernel simply switches between different tasks. Although each task is a logically separate set of actions, it must coordinate and synchronize itself, with actions in other tasks or with ISRs, by calling pSOS+ system services.

pSOS+m kernel is designed for functionally divided multiprocessing systems. The pSOS+m kernel is designed so that tasks that make up an application can reside on several processor nodes and still exchange data, communicate, and synchronize exactly as if they are running on a single processor.

pSOSystem real-time operating system provides an extensive set of networking facilities for addressing a wide range of interoperability and distributed computing requirements.

These facilities include

a. **TCP/IP Support**
b. **SNMP**
c. **FTP**
d. **Telnet**
e. **TFTP**
f. **RPCs**
g. **NFS**
h. **STREAMS**

pHILE+ is the file system manager which involves mounting and unmounting of files and also all the file operations.

pREPC+ is used to integrate library I/O functions into the target environment. It is designed to operate in a multitasking environment, and it

compiles with the C standard Library specified by the American National Standards Institute.

pRPC+ subcomponent provides extra functionality to pNA+ component by remote procedure calls. By using the above components, the required functionality of the application can be obtained.

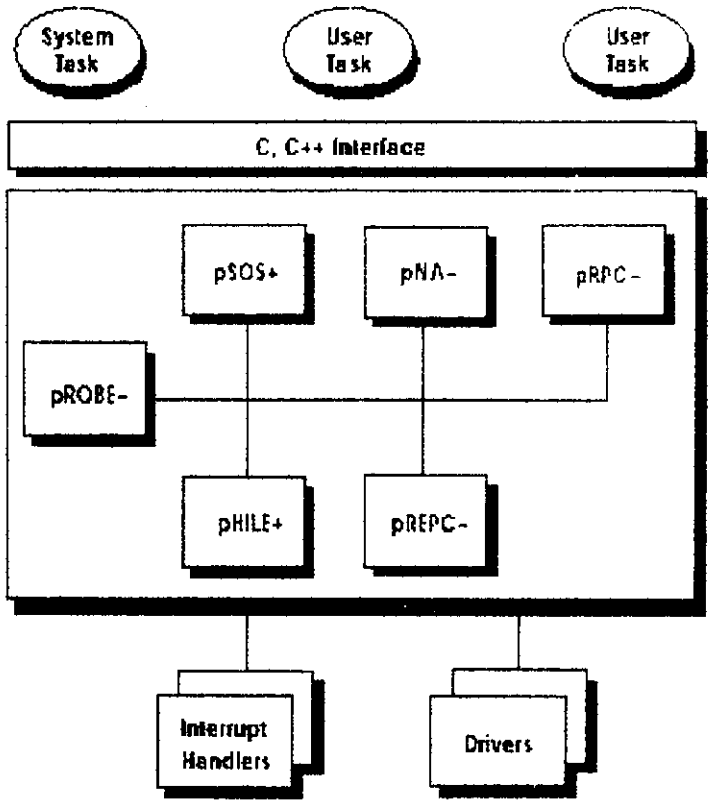


fig 3.1 pSOS Architecture

The pSOS system environment can be referred as in fig 3.1

In addition to these core components, pSOSystem includes the following:

1. Networking protocols
2. Run-time loader
3. User application shell
4. Support for C++ applications
5. Boot ROMs
6. pSOSystem templates for custom configurations
7. Chip-level device drivers

Integrated Development Environment:

The pSOSystem integrated cross-development environment can reside on a UNIX-or DOS-based computer. It includes C and C++ optimizing compilers, a target CPU simulator, a pSOS+ OS simulator, and a cross-debug solution that supports source-and system-level debugging.

The pSOSystem-debugging environment centers on the pROBE+ system-level de-bugger and optional high-level debugger. The high-level debugger executes on the host computer and works in conjunction with the pROBE+ system-level debugger, which runs on a target system. The combination of the pROBE+ debugger and optional host debugger provides a multitasking debug solution those features:

- Automatic tracking
- Traces and breaks
- Breaks on task state changes and operating system calls
- Monitoring of language variables and system-level Objects
- Profiling for performance tuning and analysis
- System and task debug modes
- The ability to debug optimized code

The pROBE+ debugger, in addition to acting as a back end for a high-level debugger on the host, can function as a standalone target-resident debugger that can accompany the final product to provide a field maintenance capability.

a) Task Scheduling:

The pSOS+ kernel employs a priority-based, preemptive scheduling Algorithm. In general, the pSOS+ kernel ensures that, at any point in time, the running task is the one with the highest priority among all ready-to-run tasks in the system. However, you can modify pSOS+ scheduling behavior by selectively enabling and disabling preemption or time-slicing for one or more tasks.

b) Task Priority:

A priority must be assigned to each task when it is created. There are 256 Priority levels - 255 is the highest, 0 the lowest. Certain priority levels are reserved for use by special pSOSystem tasks. Level 0 is reserved for the IDLE daemon task furnished by the pSOS+ kernel. Levels 240 - 255 are reserved for a variety of high priority tasks, including the pSOS+ ROOT.

A task's priority, including that of system tasks, can be changed at runtime by calling the t_setpri system call. When a task enters the ready state, the pSOS+ kernel puts it into an indexed ready queue behind tasks of higher or equal priority. All ready queue operations, including insertions and removals, are achieved in fast, constant time. No search loop is needed. During dispatch, when it is about to exit and return to the application code, the pSOS+ kernel will normally run the task with the highest priority in the ready queue. If this is the same task that was last running, then the pSOS+ kernel simply returns to it.

Otherwise, the last running task must have been either blocked, or one or more ready tasks now will have higher priority. In the first (blocked) case, the pSOS+ kernel will always switch to run the task currently at the top of the indexed ready queue. In the second case, technically known as preemption, the pSOS+ kernel will also perform a task switch, unless the last running task has its preemption mode disabled, in which case the dispatcher has no choice but to return to it.

Note that a running task can only be preempted by a task of higher or equal (if timeslicing enabled) priority. Therefore, the assignment of priority levels is crucial in any application. A particular ready task cannot run unless all tasks with higher pri-ority are blocked. By the same token, a running task can be preempted at any time, if an interrupt occurs and the attendant ISR unblocks a higher priority task.

c) Task Management:

In general, task management provides dynamic creation and deletion of tasks, and control over task attributes. The available system calls in this group are:

t_create Create a new task.

t_ident Get the ID of a task.

t_start Start a new task.

t_restart Restart a task.

t_delete Delete a task.

t_suspend Suspend a task.

t_resume Resume a suspended task.

t_setpri Change a task's priority.

t_mode Change calling task's mode bits.

t_setreg Set a task's notepad register.

t_getreg Get a task's notepad register.

pNA+ NETWORK PROGRAMMING

pSOSystem real-time operating system provides an extensive set of networking facilities for addressing a wide range of interoperability and distributed computing requirements. These facilities include TCP/IP Support - pSOSystem's TCP/IP networking capabilities are constructed around the pNA+ software component. pNA+ includes TCP, UDP, IP, ICMP, IGMP, and ARP accessed through the industry standard socket programming interface. pNA+ offers services to application developers as well as to other pSOSystem networking options such as RPC, NFS, FTP, and so forth.

In addition, pNA+ supports the Management Information Base for Network Management of TCP/IP-based Internets (MIB-II) standard. pNA+ also works in conjunction with pSOSystem cross development tools to provide a network-based download and debug environment for single- or multi-processor target systems.

The networking is done by the component pNA+. The software

architecture of pNA+ consists of four layers (Fig 3.2).

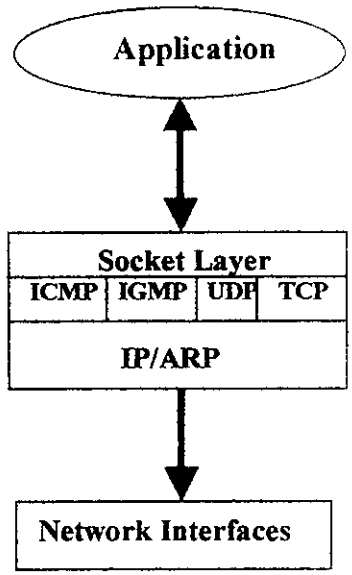


Fig 3.2 Network Layer in pSOS

a) Socket Layer:

The socket layer provides the application-programming interface. This layer provides services, callable as re-entrant procedures, which the application uses to access Internet and contains enhancements specifically for embedded real-time applications.

Sockets are created via the socket () system call. A socket descriptor is returned, which is then used by the creator to access the socket. Only, the socket's creator can use the returned socket descriptor. Sockets are created without addresses. Until an address is assigned or bound to a socket, it cannot be used to receive data. A socket address consists of a user-defined 16-bit port number and a 32-bit Internet address.

b) Transport Layer:

The transport layer supports the two Internet Transport protocols, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). These protocols provide network independent transport services. They are built on top of the Internet Protocol (IP). This layer provides packet routing, fragmentation and reassembly of long Datagram through a network or Internet.

c) Network Interface:

The Network Interface (NI) layer isolates the IP layer from the physical characteristics of the underlying network medium. It is hardware dependent and is responsible for transporting packets within a single network. Because it is hardware dependent, the network interface is not part of pNA+ proper. Rather, it is provided by the user, or by ISI as a separate piece of software. In addition to the protocols described, pNA+ supports the Address Resolution Protocol (ARP), the Internet Control Message Protocol (ICMP), and the Internet Group Management Protocol (IGMP).

By using pNA+ component the socket is created in pSOSystem and the messages can be transferred.

pHILE –FILE SYSTEM

To create files in pSOS, that is used to store the data for future use the pHILE+ component is used. According to File system manager, a file system consists of a set of files, and a volume is a container for one file system. A volume can be a single device (such as a floppy disk), a partition within a device (such as a section of a hard disk), or a remote directory tree (such as a file system exported by an NFS server). The pHILE+ file system manager recognizes the following four types of volumes:

IMPLEMENTATION DETAILS

pHILE+ Format Volumes:

These are devices that are formatted and managed by using proprietary data structures and algorithms optimized for real-time performance. pHILE+ format volumes offer high throughput, data locking, selectable cache write through, and contiguous block allocation. pHILE+ format volumes can be a wide range of devices from floppy disks to write-once optical disks.

a) MS-DOS Volumes:

These devices are formatted and managed according to MS-DOS FAT file system conventions and specifications. pHILE+ supports both FAT12 and FAT16. MS-DOS volumes offer a method for exchanging data between a pSOS+ system and a PC running MS-DOS. Because of their design, MS-DOS volumes are less efficient than pHILE+ volumes; they should be used only when data interchange is desired .The pHILE+ file system manager supports the MS-DOS hard disk and floppy disk formats.

b) NFS Volumes:

NFS volumes allow to access files on remote systems as a Network File System (NFS) client. Files located on an NFS server will be treated exactly as though they were on a local disk. Since NFS is a protocol, not a file system format, it can access pHILE+, MS-DOS, or CD-ROM format files.

c) CD-ROM Volumes:

These are devices that are formatted and managed according to ISO-9660 CD-ROM file system specifications. The data may be written either in MS_DOS format volumes or pHILE format volumes. Depending on the volume type, the hard disk or floppy disk is formatted.

pHILE+ volume is used whenever possible because they are faster and more efficient than MS-DOS volumes. If there is a data interchange involving a PC - that is, if the data will be written on the target but read later on a PC, MS-

DOS volume is used. In the MS-DOS volume, the disk is formatted using DOS commands. If a using pHILE+ volume, the disk is formatted using either DOS commands or the I/O control commands of a SCSI driver.

FORMATTING PROCESS

a) Physical format or Low-level format

A physical format puts markings on the storage medium (typically a magnetic surface) that delineate basic storage units, usually sectors or blocks. On hard disks, physical formatting is purely a hardware operation and is almost always done at the factory. On floppy disks, the physical format can be performed normally. Physical formatting very rarely needs to be redone. If it is redone, it destroys all data on the disk

b) Partitioning (Hard Disks Only)

A hard disk can be divided into one or more partitions, which are separate physical sections of the disk. Each partition is treated as a logically distinct unit that must be separately formatted and mounted. Each partition can contain only one volume. The partitions on a disk can contain volumes of different types. That is, some partitions can contain MS-DOS volumes while others contain pHILE+ volumes.

c) Writing the Volume Parameter Record:

Just as the partition table provides information about each partition on a hard disk, a volume parameter record in the first sector of each volume (partition or floppy disk) describes the geometry of that volume, which is information such as volume size and the starting location of data structures on the volume.

d) Creating an Empty File System Within Each Disk Partition:

Each volume must be initialized to contain either an MS-DOS or a pHILE+ format file system. The formatted volume is initialized with user-supplied parameters, and setting up the necessary control structures and other information needed by the pHILE+ file system manager for subsequent file operations on the volume. A volume must be initialized before it can be mounted. A volume must be mounted before any file operations can be applied to it. Permanent volumes (on non-removable media) need mounting only once. Removable volumes can be mounted and unmounted as required. The pHILE+ file system manager maintains a mounted volume table, whose entries track and control mounted volumes in a system.

The pHILE+ file system manager defines two types of files:

- Ordinary files.
- Directory files.

An ordinary file contains user-managed data. A directory file contains information necessary for accessing ordinary and/or other (sub) directory files under this directory. Every volume contains at least one directory file called the ROOT directory.

Files may not cross over volumes and therefore cannot be larger than the volumes on which they reside. Using a pathname uniquely identifies every file. Pathnames are either absolute or relative. An absolute pathname always begins with a volume name and specifies a complete path through the directory tree leading to a file or directory.

The file number follows the partition separated by a dot. A relative pathname identifies a file or directory by specifying a path relative to a predefined directory on a predefined volume, together called the current directory. The current directory is unique for each task.

e) Naming the Files:

On pHILE+ format volumes, an ASCII string consisting of 1 to 12 characters names a file. The characters can be either upper or lowercase letters, any of the digits 0 - 9, or any of the special characters. (Period), _ (underscore), $ (dollar sign), or -(dash). A name must begin with a letter or a period. Names are case sensitive — Abc and ABC represent different files. When a pathname is specified, the volume, directory, and filenames all are separated by either a forward slash (/) or a backslash (\).

Files located on MS-DOS volumes are named according to standard MS-DOS naming conventions. After naming the volumes, the basic services such as open, close, read, write and File identifier can perform positioning the file.

Before a file can be read or written, it must be opened with the open_f() system call. Open_f ( ) accepts as input a pathname that specifies a file, and a mode parameter, which has meaning only when opening files located on NFS volumes. open_f ( ) returns a small integer called a file ID (FID) that is used by all other system calls that reference the file.

A file may be opened by more than one task at the same time. Each time a file is opened, a new FID is returned. When a file is opened for the first time, the pHILE+ file system manager allocates a data structure for it in memory called a file control block (FCB).

The FCB is used by the pHILE+ file system manager to manage file operations and is initialized with system information retrieved from the volume on which the file resides. All subsequent open calls on the file use the same FCB; it remains in use until the last connection to the file is closed. At that time, the FCB is reclaimed for reuse. The close_f ( ) system call is used to terminate a connection to a file; it should be used whenever a file connection is no longer needed

pROBE+ component(debugger):

The pROBE+ debugger is a comprehensive system debugger and analyzer for the pSOSystem environment from Integrated Systems, Inc. Co resident with the kernel on the target system, the pROBE+ debugger uses detailed structural information about the pSOS+ kernel to let you observe and control system execution. Its extensive capabilities are valuable not only in helping you debug and fine-tune performance during product development, but also for field monitoring and maintenance if you leave it installed.

pROBE+ Commands:

- Memory display/modify
- Register display/modify
- Breakpoints
- Start/resume execution
- Query system information
- System profiling
- Miscellaneous commands

a) Command Syntax

This section shows the format of each command along with its input parameters.

b) Memory Display/Modify Commands

DM [.width] {address | range}

PM [.width] address value

FM [.width] range [not] value

SM [.width] range value

MM [.width] range address

DI [address]

DL [.width] [PREV | offset]

VL [.width] [PREV | offset]

c) Register Display/Modify Commands

DR

DR [keyword]

or

keyword ::= OFFSET | FPU PR reg [task] value A-4

d) Breakpoint Commands

DB address [origin]

DB SE function origin parameter

DB DI {task | *}

DB TI {ticks | time-of-day}

DB MA {address | range} [task] [origin] [access] [direction]

CB [type] [break]

e) Start/Resume Execution Commands

GO [new PC],[,tb1[,tb2]]

GS [new PC]

ST [count]

pSOS+ Query Commands

QC [MPC | PSOS | PROBE | PHILE | PREPC | PNA | PSE | PMONT]

QD

QM

QO [type]

QP

QQ [queue [NODATA]]

QR [region]

QS [semaphore]

QT [task]

QV

a) Profiling Commands

CP

LP

b) Miscellaneous Commands

SC function [parameter]

FL [flag {ON | OFF}]

IL [level]

SYSTEM CALLS

System Calls used in Windows:

The interface between the operating system and the user program is defined by the set of "extended instructions" that the operating system provides. These extended instructions have been traditionally known as **system calls**

a) The Socket Call

An application calls **socket( )** to create a new socket that can be used for network communication. The call returns a descriptor for the newly created socket. Arguments to the call specify the protocol family and the protocol or type of service it needs (i.e., stream or datagram).

b) The Connect Call

After Creating a socket, a client calls **connect( )** to establish an active connection to a remote server. An argument to connect allows the client to specify

the remote endpoint, which includes the remote machine's IP address and protocol port number. Once connection is made, a client can transfer data across it.

c) The Write Call

Both client and server use **write( )** to send data across a TCP connection. The application passes descriptor of a socket to which the data should be sent, the address of the data to be sent, and the length of the data. Usually, clients use write( ) to send requests, while servers use it to send replies.

d) The Read Call

After a connection is established, the server uses **read( )** to receive a request that the client sends by calling write( ). After sending its request, the client uses read( ) to receive a reply.

The three arguments are passed to the read( ) call. The first specifies the socket descriptor to use, the second specifies the address of a buffer, and the third specifies the length of the buffer. read ( ) extracts the data bytes that have arrived at that socket, and copies them to the user's buffer area.

e) The Close Call

Once a client or server finishes using a socket, it calls **close ( )** to deallocate it. If only one process is using the socket, close( ) immediately terminates the connection and deallocates the socket. If several processes share a socket, close () decrements a reference count and deallocates the socket when the reference count reaches zero.

f) The Bind Call

When a socket is created, it does not have any notion of endpoint addresses. An application calls **bind ( )** to specify the local endpoint address for a

socket. The call takes arguments that specify a socket descriptor and an endpoint address. Primarily, servers use bind( ) to specify the well-known port at which they will await connections.

g) The Listen Call

When a socket is created, the socket is neither active i.e., ready for use by a client nor passive i.e., ready for use by a server until the application takes further action. Connection-oriented servers call listen( ) to place a socket in passive mode and make it ready to accept incoming connections.

To ensure that no connection request is lost, a server must pass an argument to listen ( ) that tells the operating system to enqueue connection requests for a socket. Thus, one argument to the listen( ) call specifies a socket to be placed in passive mode, while the other specifies the size of the queue to be used for that socket.

h) The Accept Call

The server calls accept() specifies the socket from which the connection should be accepted. accept( ) creates a new socket for each new connection request, and returns the descriptor of the new socket to its caller.

The server uses the new socket only for the new connection; it uses the original socket to accept additional connection requests. Once it has accepted a connection, the server can transfer data on the new socket.

System Calls used in pSOS:

a) add_ni - Adds a network interface

Description

This system call is used to dynamically add a network interface to the pNA+ network manager. The characteristics of the network interface are specified in the data structure pointed to by ni.

This routine calls the network interface driver's NI_INIT routine for driver initialization.

b) Shr_socket - Obtains a new socket descriptor for an existing Socket

Description

This system call is used to obtain a new socket descriptor for an existing socket. The new socket descriptor can be used by the task with the specified ID to reference the socket in question.

This system call is provided for applications that implement UNIX- style server programs, which normally incorporate the UNIX fork( ) call.

c) socket - Creates a socket

Description

The socket() system call creates a new socket and returns its socket descriptor. The socket is an endpoint of communication.

d) t_start - Starts a task

Description

This system call places a newly created task into the ready state to wait scheduling for execution. The calling task does not have to be the creator (or parent) of the task to be started. However, a task must be started from the node on which it was created.

e) t_create - Creates a task

Description

This service call enables a task to create a new task. t_create() allocates to the new task a Task Control Block (TCB) and a memory segment for its tack(s). The task stack sizes and scheduling priority are established with this call. t_create() leaves the new task in a dormant state; the t_start() call must be used to place the task into the ready state for execution.

PROPOSED APPROACH TO THE PROJECT

4.0 PROPOSED APPROACH TO THE PROJECT

4.1 PURPOSE:

The purpose of this project is to make an effective communication between the normal operating system and Real Time operating system by interfacing them.

4.2 SCOPE:

Scope of future applications:

- Connection oriented sockets
- Connectionless oriented sockets
- Image file

DETAILS OF THE DESIGN

5.0 DETAILS OF DESIGN

5.1 Connection Establishment:

When two tasks wish to communicate, the first step is for each task to create a socket. The next step depends on the type of sockets that were created. Most often stream sockets are used; in which case, a connection must be established between them. Connection establishment is usually asymmetric, with one task acting as a *client* and the other task a *server*.

The server binds an address (i.e. a 32-bit internet ad-dress and a 16-bit port number) to its socket (as described above) and then uses the listen() system call to set up the socket, so that it can accept connection re-quests from clients. The listen() call takes as input a socket descriptor and a backlog parameter. Backlog specifies a limit to the number of connection requests that can be queued for acceptance at the socket. A client task can now initiate a connection to the server task by issuing the connect() system call. Connect() takes a socket address and a socket descriptor as input.

The socket address is the address of the socket at which the server is listening. The socket descriptor identifies a socket that constitutes the client's endpoint for the client-server connection. If the client's socket is unbound at the time of the connect() call, an address is automatically selected and bound to it.

In order to complete the connection, the server must issue the accept() system call, specifying the descriptor of the socket that was specified in the prior listen() call. The accept() call does not connect the initial socket, however. Instead, it creates a new socket with the same properties as the initial one. This new socket is connected to the client's socket, and its descriptor is returned to the server. The initial socket is thereby left free for other clients that might want to use connect() to request a connection with the server. If a connection request is pending at the socket when the accept() call is issued, a connection is established.

If the socket does not have any pending connections, the server task blocks, unless the socket has been marked as non-blocking, until such time as a client initiates a connection by issuing a connect() call directed at the socket.

Although not usually necessary, either the client or the server can optionally use the getpeername () call to obtain the address of the *peer* socket, that is, the socket on the other end of the connection.

The following illustrates the steps described above.

SERVER	CLIENT
Socket(domain,type,protocol);	Socket(domain,type,protocol);
Bind(s,addr,addrlen);	
Listen(s,backlog);	Connect(s,addr,addrlen);
Accept(s,addr,addrlen);	

5.2 Data Transfer:

After a connection is established, data can be transferred. The send() and recv() system calls are designed specifically for use with sockets that have already been connected. The syntax is as follows:

send(s, buf, buflen, flags);

recv(s, buf, buflen, flags);

A task sends data through the connection by calling the send() system call. send() accepts as input a socket descriptor, the address and length of a buffer containing the data to transmit, and a set of flags.

A flag can be set to mark the data as "out-of-band," that is, high-priority, so that it can receive special handling at the far end of the connection. Another flag can be set to disable the routing function for the data; that is, the data will be dropped if it is not destined for a node that is directly connected to the sending node.

The socket specified by the parameter s is known as the *local* socket, while the socket at the other end of the connection is called the *foreign* socket. When send () is called, the pNA+ component copies the data from the buffer specified by the caller into a send buffer associated with the socket and attempts

When send () is called, the pNA+ component copies the data from the buffer specified by the caller into a send buffer associated with the socket and attempts to transmit the data to the foreign socket.

If there are no send buffers available at the local socket to hold the data, send () blocks, unless the socket has been marked as non-blocking. The size of a socket's send buffers can be adjusted with the set-sockopt () system call.

A task uses the recv () call to receive data. recv() accepts as input a socket descriptor specifying the communication endpoint, the address and length of a buffer to receive the data, and a set of flags. A flag can be set to indicate that the recv () is for data that has been marked by the sender as out-of-band only. A second flag al-lows recv() to ''peek'' at the message; that is, the data is returned to the caller, but not consumed.

If the requested data is not available at the socket, and the socket has not been marked as non-blocking, recv() causes the caller to block until the data is received. On return from the recv() call, the server task will find the data copied into the specified buffer.

5.3 FILE TRANSFER

If the file has to be transferred, an object of CFile class is used. The CFile class provides an interface for general-purpose binary file operations. CFile is the base class for Microsoft Foundation file classes. It directly provides unbuffered, binary disk input/output services, and it indirectly supports text files and memory files through its derived classes.

A CFile object is constructed by calling CFile constructor. After the file object is created, it can be used for operations, such as reading, writing etc., by appropriate calls. The function close( ), close the file associated with this object and make the file unavailable for reading or writing.

The CFileDialog class encapsulates the Windows common file dialog box. Common file dialog boxes provide an easy way to implement File Open and File

Save As dialog boxes (as well as other file-selection dialog boxes) in a manner consistent with Windows standards.

The CFileDialog class can be used "as it is" with the constructor provided, or a class can be derived from CFileDialog and write a constructor to suit needs. In either case, these dialog boxes will behave like standard Microsoft Foundation class dialog boxes because they are derived from the CCommonDialog class.

To use a CFileDialog object, first create the object using the CFileDialog constructor. After the dialog box has been constructed it can be set or modify any values in the m_ofn structure to initialize the values or states of the dialog box's controls. The m_ofn structure is of type OPENFILENAME.

After initializing the dialog box's controls, call the DoModal member function to display the dialog box and allow the user to enter the path and file. DoModal returns whether the user selected the OK (IDOK) or the Cancel (IDCANCEL) button.

Operations Methods

DoModal --------- Displays the dialog box and allows the user to make a selection.

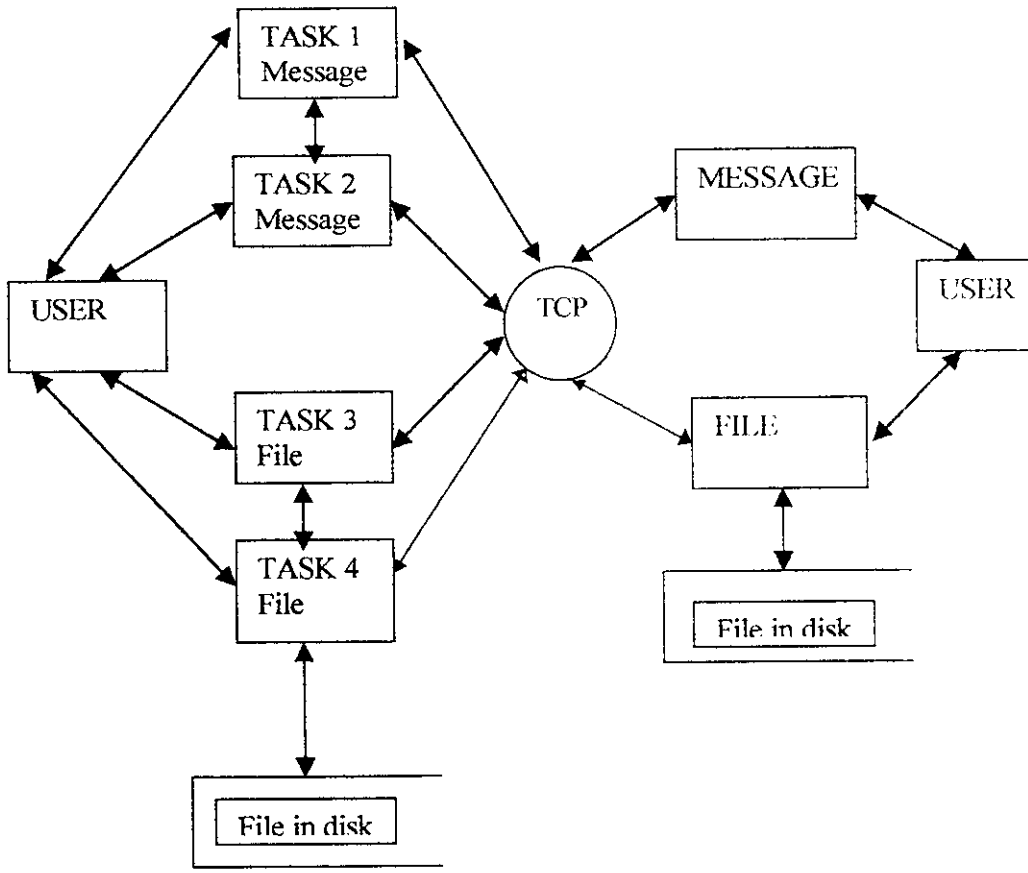
GetPathName ------- Returns the full path of the selected file.

GetFileName--------- Returns the file name of the selected file.

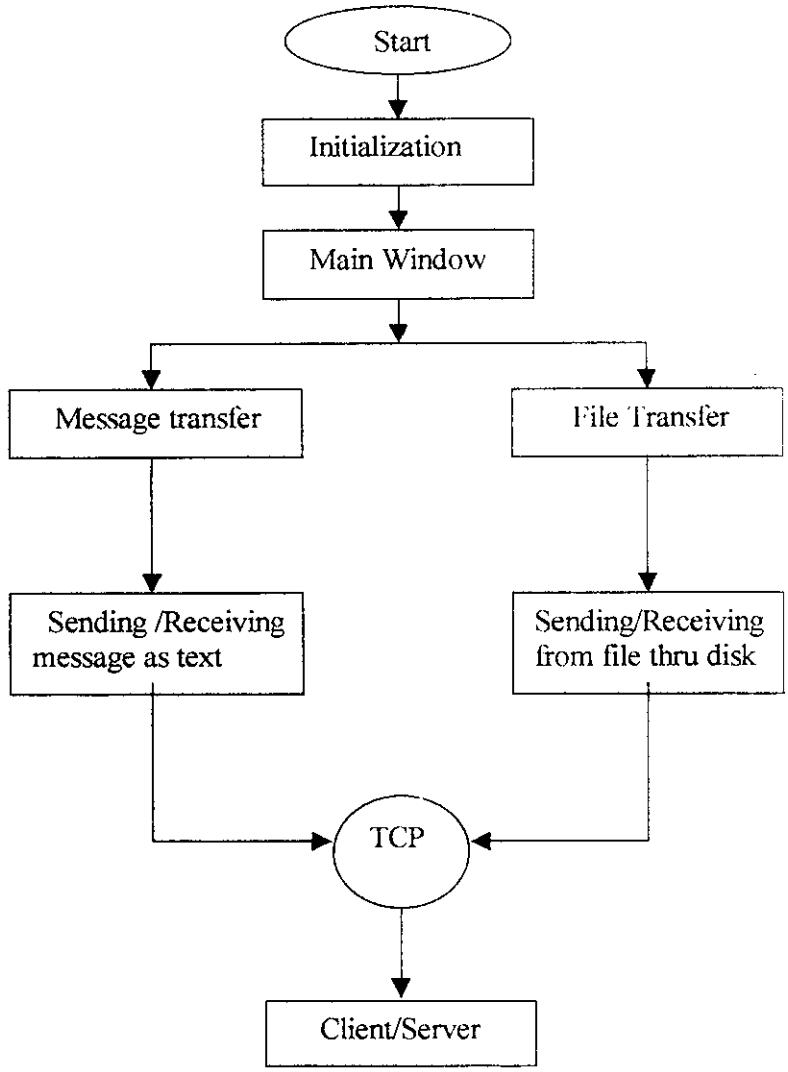
GetFileExt---------- Returns the file extension of the selected file.

GetFileTitle--------- Returns the title of the selected file and etc...

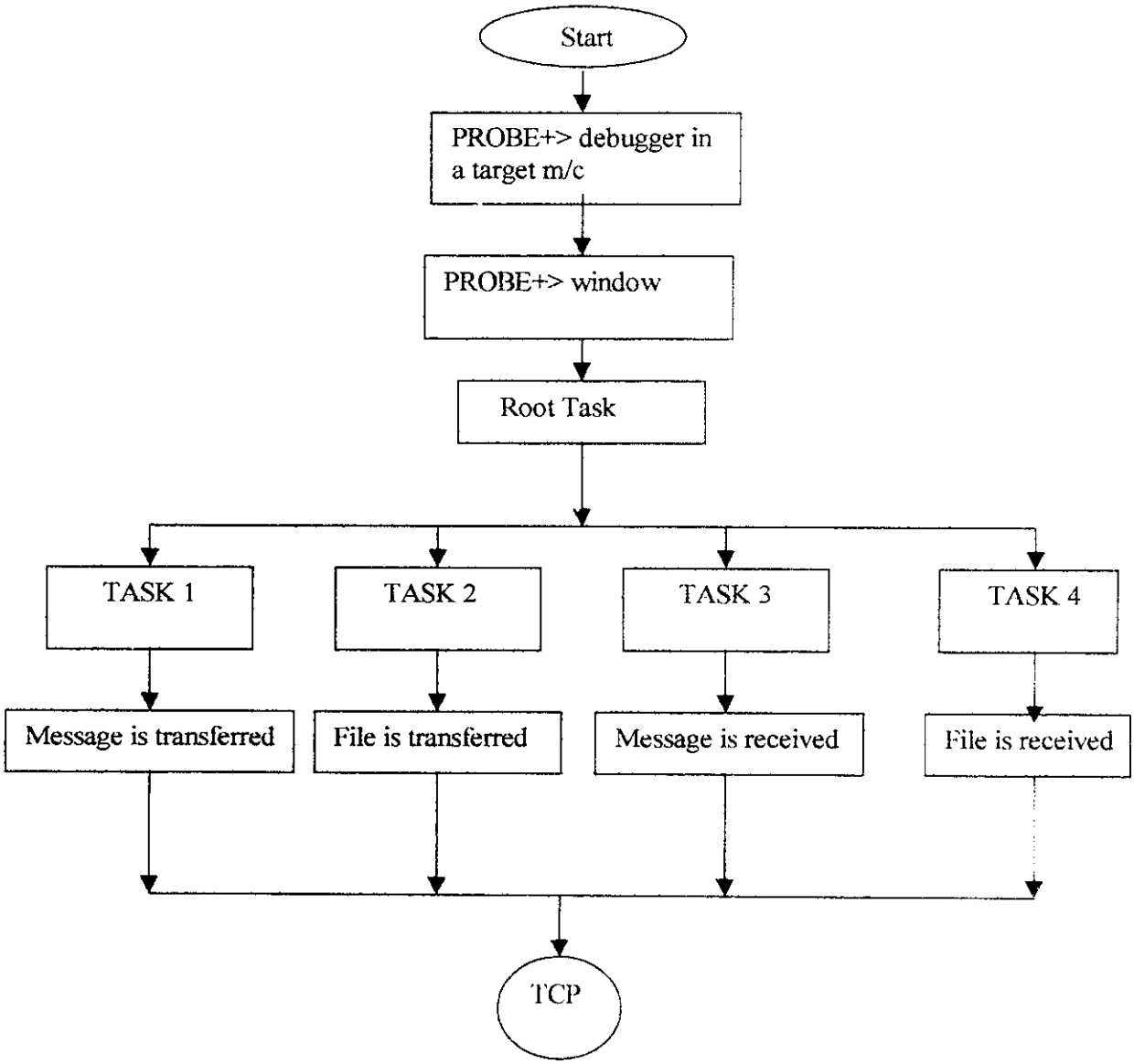
5.4 DATAFLOW DIAGRAM



System Flow Diagram for windows



System Flow Diagram for pSOS



6.0 IMPLEMENTATION DETAILS

Implementation is the stage where the theoretical design is converted into a working system implementation is the process of converting a new or a revised system design into an operational one. Conversion means changing from one system to another.

The objective is to put the tested system into operational. it consist of testing the development programs with sample data.

- Detection and correction of errors while implementing.
- Making necessary changes in the system.
- Checking of report with the existing system.
- Training the personnel user.
- Creating computer compatible files.
- Installation of hardware and software utilities.
- Connecting both pSOS with VC++.
- Users were given efficient training for the use of the effective system.

TESTING

7. TESTING

The testing process focuses on the logical internals of the software, assuring that all the statements have been tested and also the functional internals by conducting test encounter errors. The process also ensures that defined input will produce actual results that agree with required results. Testing is an important part of the development lifecycle.

7.1 UNIT TESTING:

Unit testing is a strategy by which the individual components, which make up the system, are tested first to ensure that the system works up to the desired extent. This test also ensures the integrity of data that is store temporarily. Some of the ways test the system area as follows

- Giving inconsistent data and out of range values in the form and module level.
- Rising unhandled cases explicitly.
- Boundary cases.

7.2 INTEGRATION TESTING:

Integration testing is a systematic technique for constructing the program structure while conducting tests to uncover errors associated with interfacing. The objective is to take unit tested modules and build a program structure that has been dictated by design. Integration testing consists of series of tests performed on the whole system to check the validity of the system after integrating all the different modules that are developed.

7.3 VALIDATION TESTING:

Validation succeeds when software functions in a manner that can be reasonably expected by the customer. Validation testing is used to check the performance of the input screens by giving wrong type of inputs and the system

should not respond to it by giving error messages like the given data is not correct and so on.

7.4 SYSTEM TESTING:

System testing is actually a series of different tests whose primary purpose is to fully exercise the developed system. System testing is performed on the whole system to check the validity of the system. Although each test has different purpose we must check that all the elements have been integrated to form a system.

To evaluate the practical performance of various data transfers by this interface, the messages are transferred between both client and server. And the file is transferred from the client, and it is stored in server. The different type of file is sent, such as log files, text files, document files. The average latency time is 5.5 microseconds.

CONCLUSION AND FUTURE OUTLOOK

8.0 CONCLUSIONS AND FUTURE OUTLOOK

8.1 CONCLUSION

The basic requirement that the interaction between real time operating system and normal operating system happen through an interface is satisfied; the application has fulfilled its cause. The system is designed to be flexible and can therefore support any level of expansion and customization. Also maintenance is improved drastically. Reduce the delay in the time dependent processing.

The programming techniques used in the design of the system provide a scope for further expansion, and implementation of any changes, which may occur in the future. Documentation has been done at each level of the system development in a way that could be understood and followed easily.

8.2 FUTURE OUTLOOK

One of the enhancements is done in communication style. The interface is developed for stream sockets, which is connection-oriented and it is chosen for effective delivery. This implementation can be used for connectionless style by Datagram socket. The remote procedure calls can use this interface for processing.

The communication between client and server is by means of data i.e., the form of text. It may be extended for transfer of image files between the two, which takes significant amount of time in normal operating system.

REFERENCES

9.0 **REFERENCES**

9.1 BOOKS

- Young, Michael, "Mastering Visual C++ 6.0", New Delhi, BPB Publications, First Indian Edition , 1997.

- Pappas, Chris, Murray, William, "Visual C++ 6.0: The Complete Reference", New Delhi, Tata McGraw-Hill, 2001.

- pSOS, New York, Infognana.

- TCP/IP, New York, Infognana.

9.2 WEBSITES

- www.windriver.com

- www.codeproject.com

- www.embedded.com\pSOS

- www.dedicated-systems.com\pSOS

APPENDIX

10.0 APPENDIX

10.1 SOURCE CODES:

pSOS Codes:

```
#include<psos.h>
#include<prepc.h>
#include"sys_conf.h"
#include<pna.h>
#include<stdio.h>
#include<string.h>
#include<struct.h>

int m,l,n;
ULONG tid1,sid,t1,t2,t3,t4,sid1,tid2,rid,sid2,tid3;
ULONG sid2,rid1,t5,t6,t7,rid4,t10;
int size=16,i,t,s,s1,z;
static int j,k;
struct sockaddr_in addr,dummy,dummy1;
struct sockaddr_in addr1,addr2;

void server();
void client1();
void client2();
/*ULONG add_ni(&lan);*/
char packet[4][4][15];
char msg3[4][4][15];
root()
{
t_create("server",90,1500,1500,0,&tid1);
t_start(tid1,T_PREEMPT,server,0);
```

```
t_create("client1",40,1024,1024,0,&tid2);
t_start(tid2,T_PREEMPT,client1,0);
/*t_create("client2",40,1024,1024,0,&tid3);
t_start(tid3,T_PREEMPT,client2,0);*/
t_suspend(0);
}
void server()
{
struct server namnum1;
/* Structure Created in Server */
size = sizeof(dummy);
addr.sin_family = AF_INET;
addr.sin_port = htons(2000);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
for(i=0;i<=7;i++)
{
addr.sin_zero[i]=0;
}
/* Socket created in Sever */
sid=socket(AF_INET,SOCK_STREAM,TCP);
if(sid==-1)
printf("\n socket not created");
else
printf(" socket is created \n");

/* Bind the 16-bit Port Number and 32-bit IP address */

t=sizeof(addr);
t1=bind(sid,&addr,t);
if(t1==-1)
```

```
printf("\nbinded not sucessfully\n");
else
printf(" binded\n");

/* Listen the Number of Conection */

t2=listen(sid,2);
if(t2==-1)
printf (" \n server is not listening\n");
else
printf(" listening\n");

/* Accepted Number of Connection in Clients */
/* Accepted Client one */
/*s=sizeof(dummy);
t3=accept(sid,&dummy,&s);
if(t3==-1)
printf("\n not accepted\n");
else
printf(" Accepted from  client one \n");*/
/* Received the message from client one */
/*t4=recv(t3,&msg,10,0);
if(t4==-1)
printf("\n an error occured in recv\n");
else
printf("\n Received the Message:%s \n ",msg);*/

/* Accepted Client Two */

s1=sizeof(dummy1);
```

```
t6=accept(sid,&dummy1,&s1);
if(t6==-1)
printf("\n not accepted\n");
else
printf(" Accepted from  client one \n");

/* Receive the message from Client Two */

t7=recv(t6,&msg3,1000,0);
if(t7==-1)
printf("\n an error occured in recv\n");

/*printf(" message is :%s "*(((msg3+n)+m)+l));*/
/*for(z=0;z<=15;z++)*/
        /*printf(" number is %d \t ",namnum1.b);*/
for(m=0;m<=3;m++)
        {
              for(l=0;l<=3;l++)
              {
               printf(" message are:%s\t ",*(*(msg3+m)+l));
               }
        }
/*else
printf("\n Received the message: %s \n ",msg3);*/
t_suspend(0);
}
/* Client one */
void client1()
{
struct client namnum;
```

```
printf(" Client one \n");

/* Structure Created in Client one */

addr1.sin_family = AF_INET;
addr1.sin_port = htons(2000);
addr1.sin_addr.s_addr = htonl(INADDR_ANY);
for(i=0;i<=7;i++)
{
addr1.sin_zero[i]=0;
}
/* Socket  Created  in Client One */
sid1=socket(AF_INET,SOCK_STREAM,TCP);
if(sid1==-1)
printf("\n socket is not created");
else
printf(" socket is created client one \n");
/* conection Establishment from Server */
rid=connect(sid1,&addr1,sizeof(addr1));
if(rid==-1)
printf("not connect\n");
else
printf("connect in client one \n");
/* Send the message from Client One */
/*scanf(" %s \n ",msg1);*/

if((send(sid1,&packet,1000,0)==-1))
printf("  message is not send in  client one \n");
else
printf("  message is  send in  client one \n");
```

```
t_suspend(0);
}
/* Client Two */
void client2()
{

printf(" Client TWO \n");*/
Structure Created in Client Two */
addr2.sin_family = AF_INET;
addr2.sin_port = htons(2000);
addr2.sin_addr.s_addr = htonl(0x7fa80024);
for(i=0;i<=7;i++)
{
addr2.sin_zero[i]=0;
}

/* Socket Created in Client TWO */

/*sid2=socket(AF_INET,SOCK_STREAM,TCP);
if(sid2==-1)
printf("\n socket is not created In Client Two \n");
else
printf(" socket is created client Two \n");*/
/* Connection Establishment from Server in Client Two */
```

```
/*rid4=connect(sid2,&addr2,sizeof(addr2));
if(rid4==-1)
printf("not connect in client two\n");
else
printf("connect in client Two  \n"); */
/* send the message from Client Two */
/*printf("Enter the message in Client two\n");*/
/*scanf(" %s \n ",msg2);*/
/*if((send(sid2,&msg2,10,0)==-1))
printf("  message is not send in  client two \n");
else
printf("  message is  send in  client two  \n");
t_suspend(0);}
```

VC++ Codes:

```
void CMySock::OnReceive(int nErrorCode)
{
        // TODO: Add your specialized code here and/or call the base class
        if( bReceiveBuffer == FALSE ) //receive header
        {
                sockRecv.Receive( sMsg , 1024 , 0 );
                CString str = CString(sMsg);
                int length = str.GetLength();
                CString sType = str.Left(3);
```

```
        strcpy( msgType , LPCTSTR(sType));
        CString sLength = str.Mid( 3 , 8 );
        msgLength = atol( sLength);
        CString sFileName ;
        if( sType == CString("FIL"))
        {
                sFileName = str.Right( length - 11 );
                strcpy( fileName , LPCTSTR(sFileName));
        CString fName = CString( FILE_PATH ) + sFileName ;
        MessageBox( hWnd , "File Received" , "File Receive" ,
        MB_OK );
        SendMessage( hWnd , WM_MSG , NULL , LPARAM(
        IDC_FILE ));
if(file.Open( LPCTSTR( fName ) , CFile::modeCreate |
CFile::modeWrite  ) == 0 )
{
 MessageBox( hWnd , "Error Creating File" , "Message" , MB_OK );
}
                file.Close();
        }

        else if( sType == CString("ACK"))
        {
                if( bFile == TRUE )//send file
        {
                int ctLen=0;
        CString fName = fileName ;
if(file.Open( LPCTSTR( fName ) , CFile::modeRead  ) == 0 )
 MessageBox( hWnd , "Error Creating File" , "Message" , MB_OK );
```

```
        else
        {
                        DWORD fileLen = file.GetLength();
                        do
                        {
                                if(fileLen>1024)
                                        ctLen=1024;
                                else
                                        ctLen=fileLen;

                                file.Read(sMsg,ctLen);
                                sockRecv.Send(sMsg,ctLen);
                                fileLen=fileLen-ctLen;
                        }while( fileLen > 0 );
                }
        }
        else
        {
                //send message
        }
    }
    else
    {
//MessageBox( hWnd , "Received Message" , "Message" , MB_OK );
SendMessage( hWnd , WM_MSG , NULL , LPARAM( IDC_MSG_HD ));
            //bReceiveBuffer = true ;
    }
}
else //receive data
{
```

```
        int len = sockRecv.Receive( sMsg , 1024 );
        receivedLen += len ;

        if( strcmp( msgType , "FIL" ) == 0 )
        {
                file.Write( sMsg , len ) ;
                if( receivedLen >= msgLength )
                {
                        if( strcmp( msgType , "FIL" ) == 0 )
                                file.Close();
                        bReceiveBuffer = FALSE ;
                        receivedLen = 0 ;
                }
        }
        else
        {
        SendMessage( hWnd , WM_MSG , NULL , LPARAM( IDC_MSG ));

        }
}
CSocket::OnReceive(nErrorCode);}
void CSocDlg::OnFileSend()
{

        // TODO: Add your control notification handler code here
    CFile file ;
        if(file.Open( LPCTSTR(pathName) , CFile::modeRead ) != FALSE )
        {
                char buff[20];
                msgLength = file.GetLength();
```

```
            CString sMsg = TYPE_FILE ;
            sprintf( buff , "%8d" , msgLength );
            sMsg = sMsg +  CString(buff);
            sMsg = sMsg + m_FILE ;
            int msgLen = sMsg.GetLength();
            file.Close();
            int i = sockRecv.Send( LPCTSTR(sMsg) , msgLen );
            strcpy(fileName , pathName );
            bFile = TRUE ;
        }
        else
        {
            MessageBox("Error opening File");
        }

}

void CSocDlg::OnMsgSend()
{

        // TODO: Add your control notification handler code here
        UpdateData(TRUE);

        UpdateData(FALSE);
}
```

```
        UpdateData(FALSE);
}
```

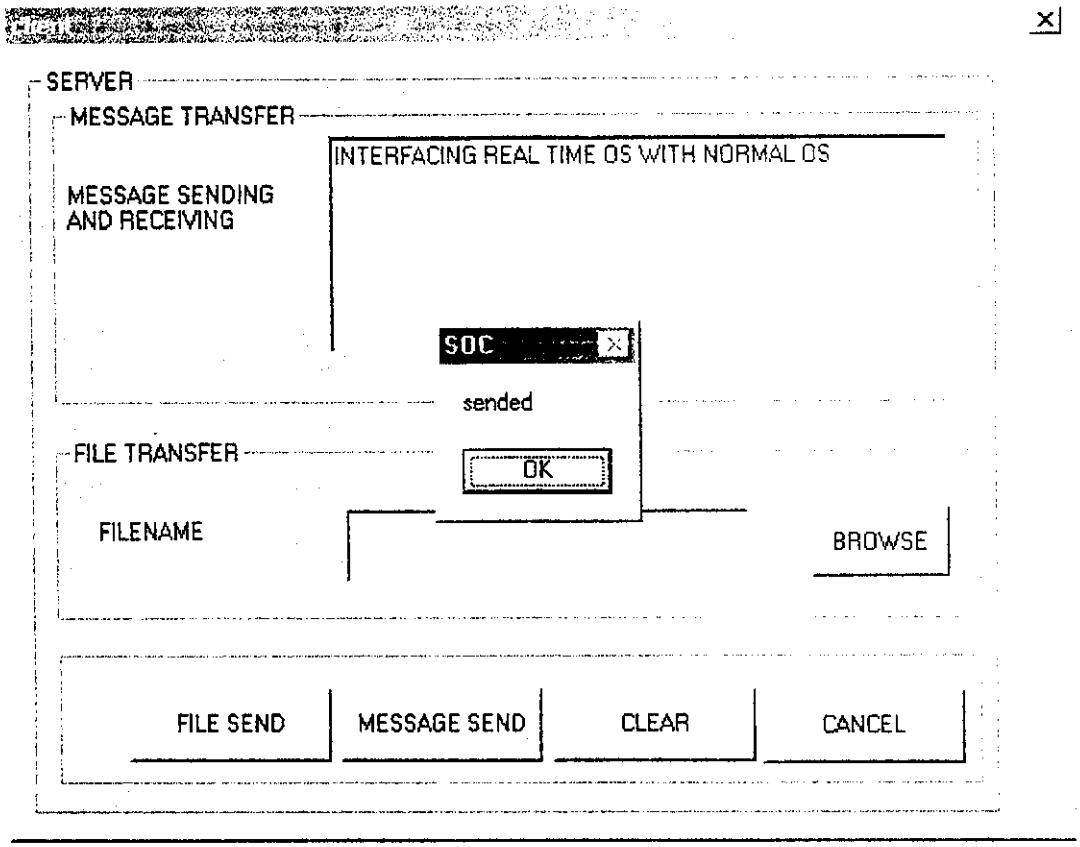
10.2 SCREEN DESIGN AND OUTOUTS

TRANSFERING MESSAGE.

WINDOWS AS SERVER AND pSOS AS CLIENT

STEP 1:

WINDOWS (VC++):



STEP 2:

RTOS(pSOS):

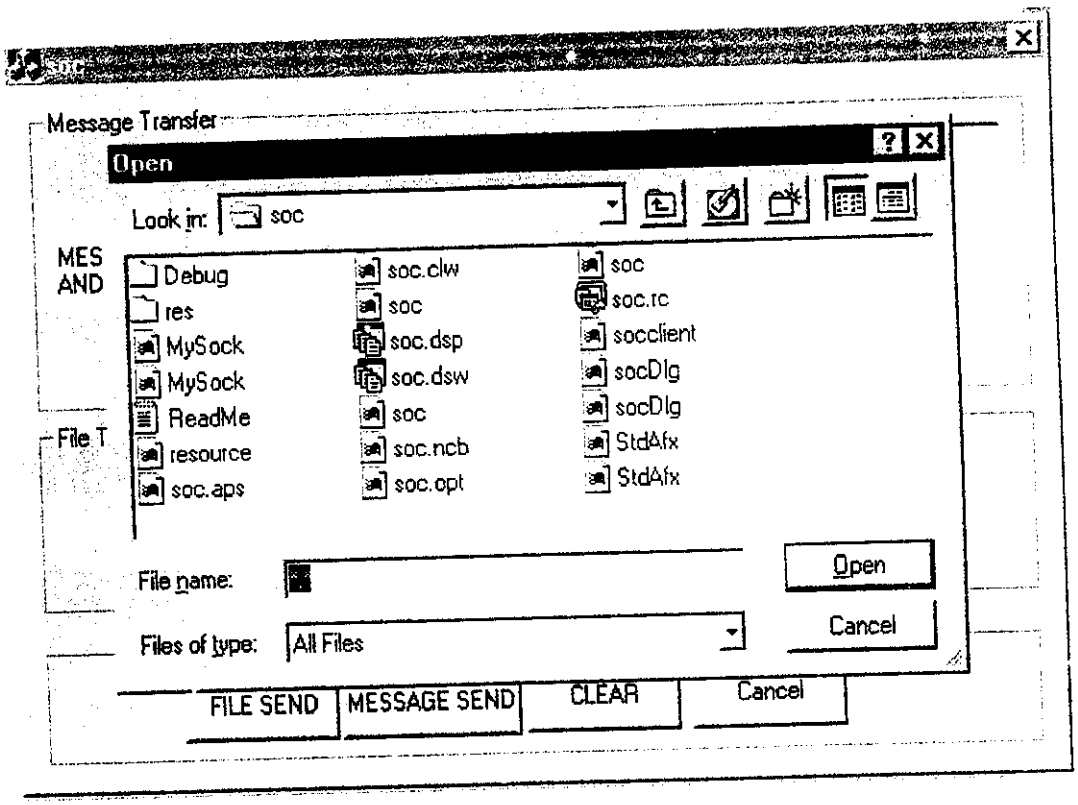
```
pROBE+> gs
Kernel Event Break Running: 'ROOT' -#00020000
-------------------------------------------------
-------------------------
pSOS Initialized Event
-------------------------------------------------
-------------------------
EAX=00000000 ESP=0025EAF4 SS=0008 EFLAGS= 00000200
EBX=00000000 EBP=00000000 CS=0010 FS=0008
IDTR=00098B90:07FF TR =0000
ECX=00000000 ESI=00000000 DS=0008 GS=0008
GDTR=00099390:007F LDTR=0000
EDX=00000000 EDI=00000000 ES=0008
EIP=00042230-00042230: 55 pushl %ebp

pROBE+> go

task one:

message received

INTERFACING REAL TIME OS WITH NORMAL OS.

Task two:

File not received
```

```
Service Break Service: T_SUSPEND Task: 00000000-
#00020000
Running: 'ROOT' -#00020000
-------------------------------------------------
-------------------------
tid = 00000000
-------------------------------------------------
-------------------------
EAX=80000006 ESP=0092CF38 SS=0008 EFLAGS= 00000216
EBX=00000000 EBP=0092CF3C CS=0010 FS=0008
IDTR=00059460:07FF TR =0000
ECX=0092CEF0 ESI=00000000 DS=0008 GS=0008
GDTR=00059C60:007F LDTR=0000
EDX=0000000E EDI=00000000 ES=0008
EIP=0003E5C6-0003E5C6: CD90 int $0x90
```

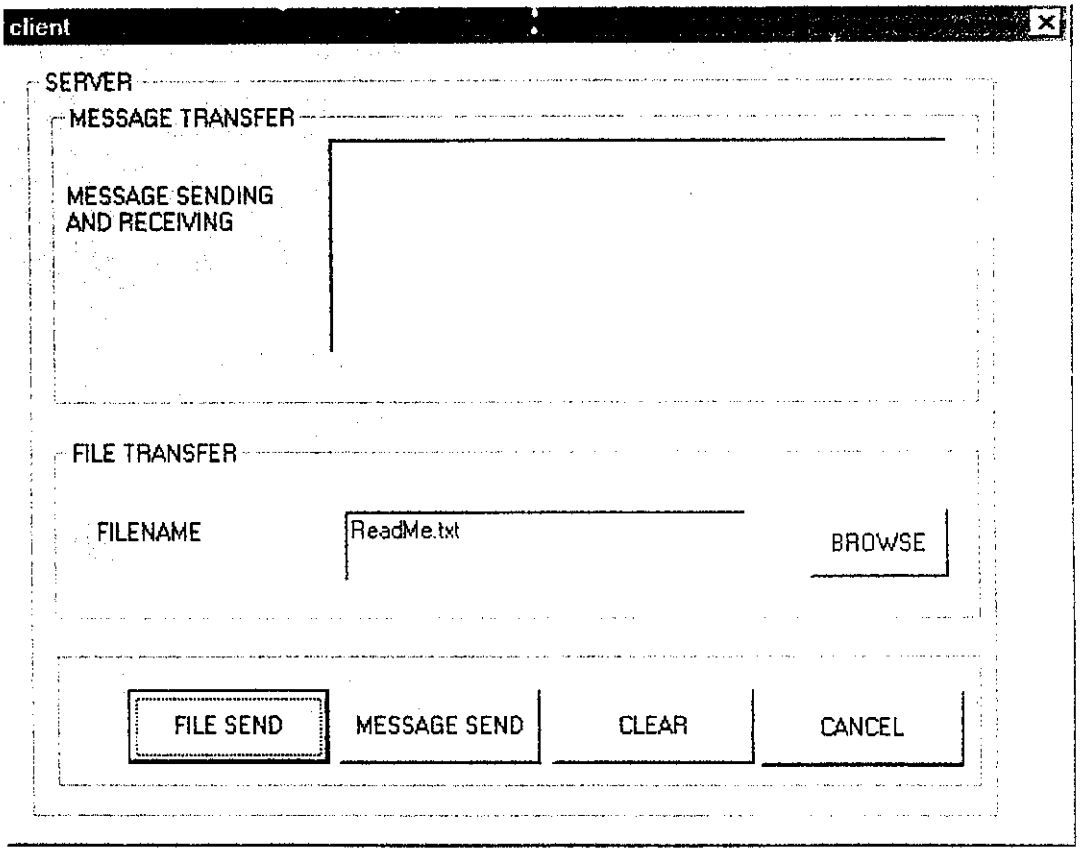
TRANSFERRING FILE.

WINDOWS:

STEP 1:



STEP 2:



STEP 3:

RTOS (pSOS) :

```
pROBE+> gs

Kernel Event Break Running: 'ROOT' -#00020000
------------------------------------------------------------
------------------
pSOS Initialized Event      59
```

```
------------------------------------------------------------
---------------------
EAX=00000000 ESP=0025EAF4 SS=0008 EFLAGS= 00000200
EBX=00000000 EBP=00000000 CS=0010 FS=0008
IDTR=00098B90:07FF TR =0000
ECX=00000000 ESI=00000000 DS=0008 GS=0008
GDTR=00099390:007F LDTR=0000
EDX=00000000 EDI=00000000 ES=0008
EIP=00042230-00042230: 55 pushl %ebp

pROBE+> go

task one:

message NOT received

task two:

file received

Readme.txt

Service Break Service: T_SUSPEND Task: 00000000-#00020000
Running: 'ROOT' -#00020000
------------------------------------------------------------
---------------------
tid = 00000000
------------------------------------------------------------
---------------------
EAX=80000006 ESP=0092CF38 SS=0008 EFLAGS= 00000216
```

```
EBX=00000000 EBP=0092CF3C CS=0010 FS=0008
IDTR=00059460:07FF TR =0000
ECX=0092CEF0 ESI=00000000 DS=0008 GS=0008
GDTR=00059C60:007F LDTR=0000
EDX=0000000E EDI=00000000 ES=0008
```