New Jersey Institute of Technology
Digital Commons @ NJIT

Theses Electronic Theses and Dissertations

1-31-1992

Real-TV : RTP/L3's visual monitor

Richard Czop
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

Part of the Databases and Information Systems Commons, and the Management Information Systems Commons

Recommended Citation
Czop, Richard, "Real-TV : RTP/L3's visual monitor" (1992). *Theses*. 2241.
https://digitalcommons.njit.edu/theses/2241

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science &
Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

Real-TV: RTP/L^3's Visual Monitor

Richard Czop
Department of Computer and Information Science
New Jersey Institute of Technology
Newark, New Jersey 07102

Master's Thesis

Table of Contents

Introduction 3

Real-Time Monitoring 6

Previous Work 8
- Software Solutions ... 8
- Hardware Solutions ... 11

RTP/L3 Simulation Components 13
- Software Components ... 13
- Hardware Components ... 14
- VRTA Components ... 14

Real-TV's Functional Specification 15
- Functions Performed ... 15
- User Inputs ... 17
- User Outputs ... 17
- System Network Topology ... 18
- Ring Statistics ... 18
- Node Statistics ... 19
- Process Tracking ... 19
- System Files ... 20
- Current Implementation ... 21

Real-TV's Design Overview 22
- Simulation Name Server ... 23
- Token Ring Network Topology Table ... 25
- Our Representation ... 25
- Real World Representation ... 25
- NTT for the Simple Token Ring ... 26
- Component Legend ... 27

Monitor Listener-Server Entity 28
Monitor/Simulation Initialization & Setup 29
Reporting Protocol within the Software Simulation 31
Types of Messages 31
Transferring the Network Topology to a Monitor 32
Resolving Reporting Functions 33

System Design - Low-level **34**
Data Structures 35
Monitor Data Structure 35
Node Data Structures 37
Ring Structures 38
Reporting Mechanism Structure 39

Module Specification **40**
File: Topology.c 40
Function Name: CheckNodePress 40
Function Name: RegisterNode 41
Function Name: PaintTopology 41
Function Name: DrawTop 42
Function Name: DrawChildRing 42
Function Name: DrawBridge 43
Function Name: DrawNode 43
File: MISC.C 44
Function Name: Unregisterbutton 44
Function Name: RegisterButton 44
Function Name: DetectButtonPress 45
Function Name: PaintZoomWindow 45
Function Name: PaintTokenWindow 45
Equipment Configuration 46
Implementation Language & Window System 46

System Evaluation 46

Network Topology Table 47

NodeZoom 47

Ring Statistics 47

ProcessTracking 47

Other modules 48

Special Cases 48

Future Extensions 49

Conclusion 49

References 50

Appendix A - Source Code Listing

Approvals

Date Submitted: 1/6/92

Date Approved: 1/6/92

Approved by: ______________________

Abstract

Today many real-world activities are being monitored and controlled by some type of micro-processor based system. Emphasis is placed on manufacturing and controlling through the use of computers. The inherent nature of these real-world applications makes them difficult to control and monitor. They must be built meeting strict timing constraints and must be predictable. We need to develop simplistic, reliable, and cost effective methods for building, evaluating, monitoring and controlling these complex systems. These tools should provide the user with system insight. At the New Jersey Institute of Technology Real-Time Computing Laboratory, we are building a system that will provide the aforementioned tools in a complete real-time environment. This paper gives the user a brief overview of the system that is currently under construction and gives a detailed look into the component called Real-TV. Real-TV is a software component that will be used as a monitoring tool to provide users with system insight. Processes and other system statistics, occurring within the simulated predictable real-time software/hardware environment, can be tracked and monitored.

Keywords and Phrases: *RTP/L^3, Monitor, Simulation, Reporting Mechanism, Real-time* .

1. Introduction

As computer technology rapidly expands into every aspect of real-world activities, we see the proliferation of micro-processor based control systems. Emphasis is being placed on manufacturing and controlling through the use of computer systems. We need to develop simplistic, reliable, and cost effective methods for building, evaluating, monitoring and controlling such complex systems. Real-time computing has emerged as the saviour and sole provider for such control environments.

The inherent nature of these real-world applications makes these systems difficult to control and monitor. The process control program must receive data, process it, and provide a suitable control reply within a priori given time frame [9] or else it fails. However, a failure in such a system is usually *not* tolerable, in fact it may cause destruction to the controlled process or environment. Therefore, these systems must be built meeting strict timing constraints and must be predictable.

Real-time control systems, in the past as well as today, have been designed and developed using little or no development tools. Most of these complex real-time control systems have failed to meet the criteria of being reliable and predictable without enduring much redesign and reprogramming. The lack thereof, is a problem common to most research and commercially available systems today. There is always a need to gain insight into the system's activities, and to detect and analyze bottlenecks. Benefits of analyzing typical programs were recognized as early as 1955 (see [26]). It would be beneficial for the designer and developer of these real-time control systems to have development aids aimed at providing debugging, testing and performance evaluation for their systems.

The system we are currently developing at the New Jersey Institute of Technology Real-Time Laboratory is made up of components that will aid these age old anomalies. The system we are constructing will be a universally-accessible tool that can be used as:

- a teaching tool
- capable of performing research experiments
- highly flexible for prototyping systems
- used for verifying theoretical research

It is the intent of this system and its components to aid future designers and developers who must endure countless hours creating these complex real-time control systems. The aid will be in the form of tools that can be used time again during design, development, testing, debugging and future modifications that the system will incur during its life cycle. Furthermore, the system is being designed and built from the start to be a highly modular, extendible and interchangeable feature rich system. Several

components are briefly described: *the language, the predictable machine and the schedulability analyzer.* For a complete reference see [1], [8], [9], [12], [17], [18], and [19]

RTP/L^3's Language

To support these novel ideas, Stoyenko [1] has designed a new real-time language, called **RTP/L^3** (Real-Time Programming/Language^3), that will support the aforementioned criteria. It is a distributed real-time language that conforms to the ideas discussed in [1]. The language is partitioned into a hierarchy of subsets, along the following three (hence the 3 in RTP/L^3) orthogonal programming axes [1] see **figure 1** below:

- Real-time features, such as activation times, time bounds, deadlines and other critical timing constraints
- Parallel features, that is, multi-tasking, concurrency control and task synchronization
- Conventional features (such as variables, statements and subprograms)

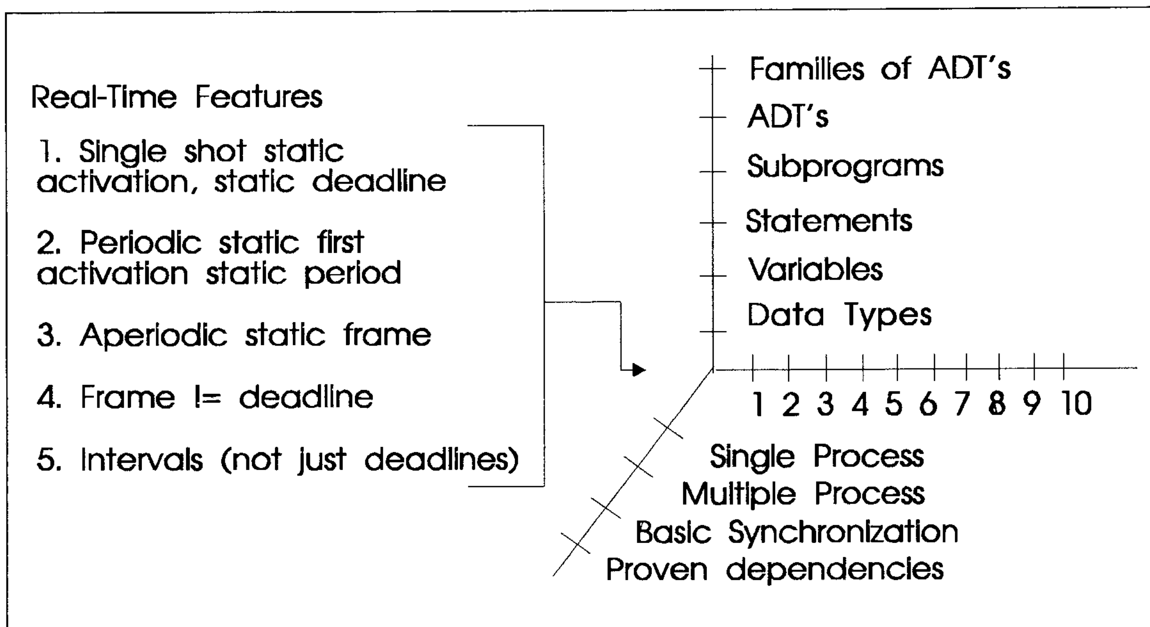

Figure 1

RTP/L^3's Hardware Machine

To complement and support RTP/L^3 a predictable software/hardware machine has been designed and is concurrently being constructed [12], [19]. The machine consists of a set of application task processors and a kernel process. The processors communicate via predictable memories. All device access is time-bounded. The architectural components are connected via a time-sliced bus, where pre-scheduled slices are allocated both processors and DMA devices [1]. The hardware system bus is based

on a token-ring network. Each ring on the network has its own token that revolves and services data transportation. Each station on the communication bus is connected to the processor through an interface.

This network architecture guarantees rotational times due to the inherent nature of the underlying token-ring. Each node on the overall hardware network is itself a token-ring. To reduce access times and enhance scalability of the network, we also allow mega-rings, where each node is itself a token-ring [1].

Schedulability Analyzer

Finally, the most important overall design issue is that RTP/L^3 is schedulability analyzable. This means that any program built using RTP/L^3 can be analyzed for predictable schedulability. That is, every program expressed in the language may be analyzed statically for its adherence to the timing constraints [1]. See [1],[8], [17], [18] for detailed description and explanation of schedulability analyzers.

The prototype system described in [1] includes a *user interface, a programming language, a hardware/software machine,* and *a schedulability analyzer tool.* Another component that is currently under construction is a visual monitor called Real-TV (Real-Time Visualizer)[1]. This component is an integral part of the software/hardware machine and plays an important role within the overall system. It is needed to allow current and future users of the RTP/L^3 software/hardware machine, to interact with and monitor processes and activities occurring within. Through a graphical user interface experimenters and students will have an insight as to what is going on within the system.

The monitor being built will let the user visualize what is occurring within the simulated real-time software/hardware system. When the simulation is executing, it will not directly output any of the results of current activities to the host machine. Rather, Real-TV will communicate with one of the active simulations and display/monitor all the user selected activities occurring in that real-time software/hardware system.

1 Monitor, visualizer, and Real-TV will be used interchangeably and will all have the same meaning.

2. Real-Time Monitoring

Today's emphasis is on delivering quality software. Program testing, debugging and performance analysis, are as much a part of every programmers daily regimen as is the editor they use to input their source code. The literature of the early 80's seemed to shy away from these novel topics. Glass [7] goes as far as writing an entire paper based on the "lost world" of software testing and debugging. However, in the last several years much more research literature has been directed in this direction. This is partially based on the need of such tools to help aid builders of these complex time dependant systems, and partially based on the skyrocketing need of such systems within manufacturing, scientific, medical, and environmental control industries [8].

Many of todays computer systems are used to measure and control real-world real-time processes. You can imagine how much a real-time software system would benefit from exploiting such a simple idea as using a monitoring tool. If the real-time software system could be tested and monitored accurately while the system is being developed, it would add to the overall quality, reliability and predictability of the complete system.

Plattner and Nievergelt, define monitoring as

> "*...the interaction between a target process and a monitor process by means of predicates and actions. This interaction can be specified in a satisfactory way only if the execution monitor is tailored to the programming language of the target program*" *[16].*

One problem with this definition is the lack of generality. Each programming language would need a language specific monitor process responsible for gathering the statistics. A better approach would be to create a general mechanism and format for monitoring programs written in different programming languages running on heterogeneous hardware platforms. This however, is not the argument of this paper and is left open for further research.

Another definition given by Haban and Wybranietz, defines monitoring as "*...the extracting of data during program execution*" [14]. This is a much simpler definition in that it says nothing of how this extraction takes place.

A third definition given by Plattner states that

> "*a real-time monitor enables us to observe the behavior of the production version of a real-time process, in order to collect genuine data, usable for statistics (system and performance), or to detect illegal or unexpected process states (identifying erroneous behavior, debugging)*" *[4].*

All three of these definitions are similar in there functional description of monitoring; however, each describes a different method of monitoring and gathering the data to monitor.

A monitor of a distributed system such as RTP/L^3s Real-TV needs to collect, interpret, and display information concerning interactions among concurrently executing processes. A well designed distributed monitor should be capable of collecting communication statistics, detecting deadlock, and providing views of the nondeterministic process execution. This information and its display can support debugging, testing, performance evaluation, and dynamic documentation of the distributed system [15].

To reiterate, real-time processes are processes whose correctness depends on their behavior with respect to time. These processes must meet certain timing constraints as imposed by the external real-time environment being controlled. The monitoring system should, therefore not affect the timing constraints imposed by the target process being monitored. These timing constraints are those that will guarantee the correct outcome of the overall process. We are not concerned with the logical functional outcome of the system, but are concerned with preserving the time that those functions perform within. If a process was interrupted for a brief moment to allow it to dump its current status of internal variables to some reporting function, then this would not constitute a method that meets the above criteria. This sudden pause in the process would add a time period to the overall runtime. The timing constraints imposed by the external environment may and probably would be violated. This example illustrates what we mean by preserving timing constraints.

If the monitoring system conforms to the aforementioned criteria we call it *real-time monitoring*. A real-time monitor enables us to track behavior of an executing real-time process, and analyze it for statistics, errors, missed deadlines and other critical outcomes, without violating any imposed timing constraints. It would be appropriate to build a system that could be incorporated in the development and debugging stages of program development, as well as during execution of the completed system that fulfilled these requirements. However,monitoring, testing, and debugging a real-time system is a very difficult undertaking because of the imposed timing constraints, asynchronous concurrent activities, lack of central control [14], and the systems nondeterministic behaviour. Today almost all such monitoring devices share computing resources and add unwanted interference.

3. Previous Work

There have been countless papers written on the subject of real-time monitoring. The central issues being--designing and developing a monitor in such a way as to not intrude on the underlying real-time process. By *non-intrusive* , we mean that when the real-time process or processes are executing, the monitor will not interfere with any outcomes or jeopardize any timing constraints. The overall process output should still be stable, predictable and reliable when the monitor is or is not running.

Some possible side effects can be [3]:

- *Performance side effects:* the tool uses some of the system resources, this reduces the effective performance of the system as seen by the user.
- *Functional side effects:* the tool changes the functional behavior of the system, in particular introduces errors.

Several papers reviewed deal with monitors that are both non-invasive and non-intrusive by means of software and/or hardware solutions. Most authors believe that the monitoring system should be built as part of the initial base system [4], [2], [3], [14] but is usually omitted because of economic and development time issues. Even today, monitoring facilities are rarely integrated into programming languages and systems. If the monitor was built as part of the overall system, then support mechanisms could easily be provided without later redesign to support such features.

3.1. Software Solutions

Software monitors present information in an application-oriented manner. These monitors are usually contained within the measured system, sharing with it the same execution environment; thus producing some degree of interference in both the timing and space of the monitored program [14]. Many software monitors are used to facilitate debugging in systems [20], [21], [22], [23]. Others have developed Hybrid tools, [24], [25] which can benefit from advantages of both hardware and software monitors. Following are several overviews of papers based on "software approaches" to real-time monitoring.

As mentioned above, several research papers in the literature provide software solutions. Plattner [4] believes that the monitoring act should be conducted on a symbolic level. The operator must communicate with the monitor in terms of source code. The author also believes that high-level languages should be used and that they be extended to handle such requests.

The mechanisms used by [4] to express monitoring activities are based on two distinct parts: *a predicate and an action*. The predicate is a boolean expression. These

actions may save values, increment counters or simply terminate the monitor. These make up the foundation of debugging and data analysis. A typical session in this system must continually observe the state of the process and evaluate the predicates that are submitted to the monitor by the operator. The other operations performed, trace low level events that change the process state [4]. These are used in the future to reconstruct what has just occurred.

This system, like several others, also uses simple hardware hooks i.e. a "bus listener" to eavesdrop on memory transactions and all bus traffic. This "eavesdropping" produces no contention and therefore is considered non-intrusive by nature.

Svobodova [3] describes several structured guidelines to be used when designing a performance and reliability monitor. The monitors described are considered observers rather than controllers. She also believes that the monitor activities should be an inseparable part of the systems operations. Not only can monitors perform the above described functions of debugging and testing but, there should be a facility for a special type of monitoring--exception handling. In order for the exception handling facility to work properly it requires some cooperation from the underlying hardware components. These components must report back to the handler describing what errors or exceptions have occurred. However one weakness to the mentioned components is that they are all separately described. It is my belief that in order to build a sophisticated monitor it is necessary to incorporate each one of these types of monitors into one *complete monitor.*

Today, most if not all, computer systems are based on some sort of communication backbone. The RTP/L^3 project described earlier in fact relies on the token-ring network as the sole means of bus communication and interconnection. Monitoring of the communication network is another element that fits the monitoring paradigm. Reliability of the communication network and individual nodes on the interconnected network all have significant effect on the performance of the system [3]. Therefore they are prime candidates for monitoring unforeseeable events that may occur over the communication network.

The authors of [2] describe a monitoring system being built as part of their distributed real-time operating system. The ARTS system has built-in kernel primitives that aid in the monitoring process. Interestingly the ARTS system monitor must cooperate with a schedulability analyzer. To provide a monitoring system that will not interfere with deadlines their system incorporates time delays for each monitoring activity. The monitoring process is built in such a way that each periodic monitoring activity is taken into account for the schedulability analyzer [2]. This guarantees that the processes will meet their respective imposed "hard" deadlines and that the schedulability analyzer can verify these same deadlines.

Like the previous monitor they also have implemented their monitor as an integral part of the system. When the monitor is in place their is no functional or performance

side effects. However, because no software method guarantees 100% un-intrusiveness it too suffers from some contention issues.

The overall structure of the ARTS monitor has three main parts: the *Event Trap, the Reporter, and the Visualizer*. The Event Trap is the entity responsible for trapping and recording significant occurrences. It is built into the operating system kernel. The Reporter is responsible for sending the Event Trap data to the Visualizer. Finally, their monitor is also built using graphical elements. They say that no one is interested in seeing massive amounts of raw data, rather it is more meaningful to visualize such data using visual aids. This portion of the monitoring system is responsible for portraying the significant data to the user [2].

The Visualizer interprets the raw data and presents it in such a way as to inform the user. Some data represented by the Visualizer is as follows:

- Debugging data
- Execution diagram
- CPU utilization for periodic and aperiodic tasks
- Number of successful completions
- Missed deadlines

It is also capable of replaying a recorded execution history at a later time. This makes it possible to trace a process at a later time to find bugs or functional errors.

One problem with the approach taken by the ARTS system is that it addresses only monitoring of applications software that is static. All the monitor conditions can be statically deduced from system symbol tables, link, and load information and statically loaded into the qualification control unit.

Another system called EXDAMS allows reenactment of the execution and the display of information of interest to the user by extracting history files generated by a modified compiler. It produces object code that leaves behind a history file. This approach illustrates a monitoring system that is extensible: new features that interpret monitoring data in different ways can be easily added [28].

Joyce et al., presents a monitoring system that provides an interactive, animated display of an executing distributed program and enables user control of nondeterminism. It is also extensible in that the detection and collection of the monitoring information is separate from the analysis and display of this information to the users [15].

The above mentioned monitoring system was designed within the Jade programming environment. The monitoring system supports the observation and control of message passing within a distributed application system that consists of a set of

concurrently executing processes. All monitoring data is gathered through a channel process that resides on each machine being monitored. All information from the application processes executing on that machine is gathered by this channel. A channel distributes this information to one or more consoles (which may be running on an entirely different machine), and each console receives information from one or channels. A console then examines and interprets the monitoring data it receives and presents it to the user.

Processes loaded with this monitoring mechanism suffer slight execution speed penalties. Typically, processes under development are monitorable, whereas system processes and application processes, which have already been tested or that are installed in a production environment, are unmonitorable.

I believe that a monitor is a tool that should be available to be used from conception of the system through delivery and monitoring of the production system therefore, this system does not suffice. The benefits of a monitoring system come from being able to use it during any phase without lose of timing constraints and program logic.

As with all software approaches the monitor and its support mechanism add some delay. Even if the delay is included into the overall running time and analyzed by the schedulability analyzer it still adds time that would otherwise not be included. Therefore the "software approach" may not be suitable for all real-time systems. This is why others have devised ways of providing real-time monitoring without the added time delays that may not be tolerated in some system. These follow the "hardware approach" of real-time monitoring and debugging.

3.2. Hardware Solutions

Hardware solutions, on the opposite end of the spectrum, rely on a device that is designed to have minimal or no effect on the host system. Most, however, provide at best, only limited, low-level information about the activities occurring on the host system. Simple observation of system buses, or probes connected to the processor, memory ports, or I/O channels, do not solve monitoring problems in these systems. These monitors often use sophisticated features of the hardware to get valuable information. In addition, hardware monitors cannot handle the dynamic creation and deletion of processes, the migration of program parts in memory, and the use of memory management units [14].

In the early sixties commercially available hardware monitors came into widespread use, and the field of performance evaluation developed. See [27] for a complete comprehensive bibliography of the early literature.

Tsia et. al. [5], believe hardware solutions to monitoring real-time systems are a much better solution. Not only do they provide non-intrusive monitoring but they also alleviate the need for the programmer to insert print statements and breakpoints into the program code [5]. The software solutions described above all add interference to the target system and share the computing resources. The following hardware methods provide a better solution. They provide non-intrusive behavior while providing comprehensive real-time monitoring and debugging solutions.

In [5] the detection of significant events is performed by what they call a *programmable qualification control logic*. The qualification unit here is responsible for handling all information gathered during each cycle execution of the processes. This unit is supported by dedicated hardware rather than by software as described earlier. This auxiliary hardware records and feeds the monitor with execution data. The unit is a microcomputer-based module with the ability to do as just described [5]. This approach has been used for a number of years in the hardware industries and in logic analyzers as tools for debugging.

The architecture to support this hardware based monitoring has two components: *the interface module, and the development module.* The interface module is the front-end and the development module is the back-end. The first module is a specific piece of hardware that interfaces with the host real-time system hardware. It detects the internal states of the target system based on predefined user conditions. The second development module is the host computer for the interface module. This module is the general-purpose microprocessor-based system that contains all the supporting software for the initialization of the interface module and postprocessing activities [6].

By connecting the monitoring system to the internal bus of the target system and collecting data directly from them this hardware solution does not add any contention or intrusion. And finally because the monitoring system does not steal CPU time from the target real-time computing system, it does not interfere with the target system execution [6]. For a full detailed description of the underlying hardware see [6].

Haban and Wybranietz present another hardware supported monitoring mechanism. They have developed a special Test and Measurement Processor (TMP) which was designed to be an integral part of each node in a multicomputer system. The TMPs are completely transparent with a minimal, less than 0.1 percent, overhead to the measured system. This device is responsible for monitoring, recording, and evaluating the activities of the host node as well as its communication activities [14]. To alleviate the communication contention that would result from the TMPs sharing the same communication medium as the underlying communication network each TMP is connected via a separate network to a central monitoring station. This central station is an arbitrarily selected workstation of the collected network nodes. It also is used for interactive monitoring, global measurement and distributed debugging [14].

To collect relevant data the monitoring mechanism uses events that are generated by the monitored software. These events are later categorized, time-stamped, processed and displayed for users to evaluate.

Both software and hardware approaches to supporting monitoring have been presented. Although neither approach may be right for every need each carries its own benefits and disadvantages. These presentation have been given in order to give the reader an overview on how people use monitors and how these monitors are supported.

4. RTP/L^3 Simulation Components

Currently RTP/L^3 has several software and several hardware components being designed and developed. Keep in mind that each component must be predictable, (as described earlier) thus each component must be thoroughly thought out and designed before development can begin.

4.1. Software Components

There are several software components that make up the entire RTP/L^3 project currently under development.

- Schedulability Analyzer
- OS Kernel Processor
- I/O Processor
- Command Processor
- Scheduler

The *schedulability analyzer* was described earlier see [1], [8], [17], [18] for a complete description and motivation. This component is currently being designed and development will begin shortly.

The *kernel processor* will be responsible for servicing requests made by the CU for process loading. When a CU is ready and idle the kernel processor will load the program into secondary memory for the CU.

The *I/O processor* will be responsible for handling keyboard input and for displaying terminal output. One example is a user wanting to load a program. The user would type in a command, such as load, from the keyboard, then the I/O processor will properly handle this input.

The *command processor* is responsible for interfacing with both the I/O processor and the OS kernel. It will properly dissect and handle requests from the I/O processor and issue commands to the kernel processor for handling these inputs.

Finally, the *scheduler* is responsible for directing processes to available system components. It will resolve any contention and will appropriately schedule each process.

4.2. Hardware Components

There are also several essential hardware components being designed and developed:

- Hierarchical Token Ring Network and Communications
- Control Unit
- Arithmetic Logic Unit
- Main Memory
- Secondary Memory
- Device Processor
- Registers
- PSW

The hardware system must be 100% predictable [2]. The predictability is guaranteed by the following rules [11]:

- A global clock keeps track of time elements known as quanta.
- A quantum is the smallest unit of time and can be interpreted as a nanosecond, a second, a minute or whatever other timing unit seems appropriate.
- Every virtual processor knows beforehand how long each operation will take to execute (quanta).
- We assume that the circuitry of the hardware will not malfunction.
- Token Ring based network for communication.

The two major hardware components are: *UNIX/Virtual Real-Time Architecture Interface and the Virtual Real-Time Architecture (VRTA).*

4.2.1 VRTA Components

The *VRTA* consists of several components listed above [11]:

- Control Units: These fetch and execute assembled programs.

2 Predictability within the software simulation

- Arithmetic Units: These execute only arithmetic statements (the CUs execute logical branching requests.)
- Memory Processor: Use memory paging.
- OS Kernel (Processor): Loads a program from UNIX into the VRTA secondary memory then schedules it for processing.
- Predictable Bus Architecture: Hierarchical Token Ring
- Device Processor: Forms the interface between UNIX keyboard input and screen output and the real-time VRTA keyboard/screen I/O.

5. Real-TV's Functional Specification

Following is a detailed description of the functions Real-TV performs and how they are accessed.

5.1. Functions Performed

Following is a list of all the functions performed by Real-TV:

Connection to a Simulation*-- monitor connects to a user selected simulation and begins to monitor user selected activities. If the user chooses he may obtain a list of all the simulations that are currently executing, by querying the SNS. This list is maintained by the SNS as described earlier.

Displaying of the Network Geometry (User Level View) -- a graphical representation of the simulations underlying hardware topology. This view represents the entire network in a way as the user can manipulate objects and use them to monitor the activities occurring. This view differs from the one below in that it presents only the portion of the topology that can fit on the screen. If the network is very complex and large the Overall Level View will instead be used. Overall Level View is needed in order to present the user with a view that can be zoomed into to obtain a suitable User Level View.

Displaying of the Network Geometry (Overall LevelView)* -- graphical representation of the entire network. This view may be one in which the user must zoom in onto a particular segment that he wants to monitor. This view provides a look at the entire topology of the underlying network topology. Once a user selects a portion of the network that he is interested in the zoomed in view will be conceived as described above, User Level View.

Static Ring Statistics -- ring statistics such as utilization, current activity, average free token time, max freetoken time, average data packet size, max data packet size.

Continuous Ring Monitoring -- bar graph of data packet size, throughput, and data within the token.

Static Node Statistics -- node statistics such utilization, current activity, average idle time, max idle time, average busy time, max busy time.

Continuous Node Statistics* -- bar graph of the utilization, throughput, and current activity occurring at the node.

Static Bridge Statistics* -- bridge statistics such as utilization, current activity, average data packet size, max data packet size, and number of routings.

Continuous Bridge Monitoring* -- bar graph of data packet size, throughput, etc.

Process tracking -- track migration of process through network.

Process statistics -- process statistics such as process id, process name, current activity, network position, controlling CU address, processing time, active time, last operation performed, missed deadlines*, success rate*, and critical paths*.

Recording of current activities*-- record all activities for later playback

Playback of recorded activities*-- playback previously recorded activities for offline analysis

Token Statistics -- bar graph showing size of data in token, and token state (busy or idle).

Exception Handling Facility* -- Process and view exceptions raised during runtime. This aids the user during debugging.

Debugging Facility* -- provides an inside view for programmers of what variables contents are, what information is packaged inside a packet, etc.

Keyboard Interface -- provide a window which allows interactive keyboard input

User configuration file* (passed on command line) -- allows the user to have predefined activities that he wants to monitor

Connection to simulation from within monitor* -- allow user to connect to a different simulation from within the monitor by selecting from a list of the currently active simulations.

Note: * -- denotes a future function or extension.

5.2. User Inputs

Most user input is directed via the mouse and the three mouse buttons. Some examples include:

Menu selections -- user positions the mouse pointer inside a menu button and presses the left mouse button. These commands perform all major functions.

Dialog button selections -- same as above. These commands perform activities corresponding to specific dialog boxes.

Node Statistics -- user positions the mouse pointer inside one of the nodes and presses the middle mouse button or right mouse button. Middle mouse button gives the user static statistical values while the right mouse button gives continuous reporting capabilities per that node.

Ring Statistics -- user positions the mouse pointer inside one of the rings and presses the middle mouse button or right mouse button. Middle mouse button gives the user static statistical values while the right mouse button gives continuous reporting capabilities per that ring.

Bridge Statistics -- user positions the mouse pointer inside one of the bridges and presses the middle mouse button or right mouse button. Middle mouse button gives the user static statistical values while the right mouse button gives continuous reporting capabilities per that bridge.

Process Tracking -- user selects Process from the main menu.

5.3. User Outputs

Since the entire system is built with a graphically based user interface outputs are viewed from the very start. It is up to the user to manipulate the objects on the screen and use them to monitor the software/hardware simulation. The user is basically in control of all outputs. He can manipulate any of the network objects and get a internal view of what is happening at any point in time.

When the simulation starts it displays the overall underlying hardware topology. This topology is made up of:

- Nodes on the ring
- Rings on the network
- An entire network

Each node on the network is part of only one ring, however a bridge connects two disparate rings and therefore is a member of both rings. For a full description of each type of node that makes up the hardware system see section labeled VRTA Components.

The user may select to view statistics for a particular component and will view the data via graphs, histograms and/or static statistical data. Some examples include:

5.3.1 System Network Topology

Upon initialization of the system a graphical representation of the entire underlying network topology will be displayed. This is where the user begins to manipulate objects and monitor the system. The user can select nodes on the network or make menu selection to monitor process activities.

5.3.2 Ring Statistics

When the user presses the right mouse button when the mouse pointer is within one of the rings, statistics are displayed about that ring. These include:

- Ring number
- Average data in a packet
- Continuous bar graph displaying the current data size in the packet
- Token state (idle or busy)
- Source address
- Destination address
- Data in the packet.

The ring statistics enable one to study the traffic on the ring, possibly pointing out a need for a faster medium or possibly detecting an over used or crowded ring.

5.3.3 Node Statistics

When the user presses the middle mouse button when the mouse pointer is within one of the rings statistics are displayed about that node

(a bridge is not a node). These include:

- Node type
- Parent ring
- Maximum idle time
- Average idle time
- Maximum busy time
- Average busy time
- Current activity

These reporting features enable a user to pin point over worked network components, under worked components, or possibly debug operations performed at this particular node.

5.3.4 Process Tracking

When the user selects *Process*, by pressing the left mouse button while the mouse pointer is within the menu button labeled Process, he will be able to select one process from the currently active processes[3]. The process state will be monitored and tracked. Process statistics such as:

- Process ID
- Process name
- Processes active time
- Current activity
- Current Processing node address
- Controlling CU address
- Current network position of process
- Elapsed processing time for the process
- Last node that processed the process

---

3 Selection of a process from a list currently not implemented

will be displayed in a separate window. The actual process movement will be represented by showing its absolute position within the network graph currently displayed on the screen see figure 2. When the processes CU requests operations from other network components such as an ALU to perform an arithmetic operation that process will be shown migrating to the available ALU within the network. This type of process tracking will enable a user to see how processes are effected by having a different number of specific components on the network.

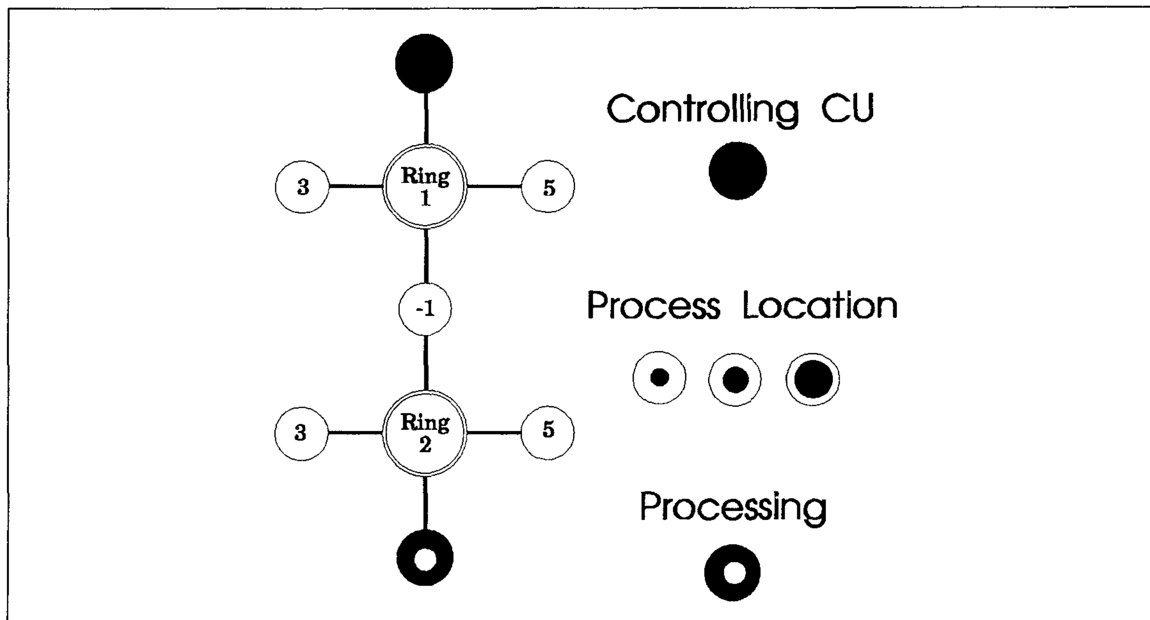

Figure 2

In this particular example if there are several ALUs available on the network, most likely the CU will find at least one idle ALU to who it can route its arithmetic operation. In another case there may only be one ALU and therefore that process will be idle and incur a delay due to the fact that no ALU was available. In this scenario the user may want or may be forced to add another ALU to the network. This would alleviate the problem of delays and unnecessary idling.

This is just one simple example of using the process tracking function that is available within the monitor. You can imagine how powerful this feature can be in tracking processes, mapping routing paths and detecting timing constraints.

5.4. System Files

When a user selects to record monitor activities the corresponding activities will be saved for later playback into a flat-file. The format of the file has yet not been determined and is open to whomever implements this feature.

A second file will be used to store relevant startup values that will be used to initialize the monitor when it is started. This will allow different users to start the monitor with different parameters and allow them to view and monitor different activities within a simulation.

Another file that is used is the NTT (described earlier). This file stores the network topology table and describes the network layout for use with the simulation and monitor. As described earlier, the information stored in the NTT is communicated to the monitor via the RTP/$L^{3's}$ Reporting Mechanism.

5.5. Current Implementation

Currently the monitor and simulation have the following properties:

- The software monitor can only monitor/display one (1) simulation per monitor instance.
- Multiple instances of a monitor can all monitor/display the same simulation. (This enables different users to view diverse portions of one simulation at the same time, possibly from two or more different machines)
- A monitor can connect to a simulation that is already in progress.
- The user of the monitor has full control of what, when, and who they monitor. (Ex. User A can monitor a process running on a simulated software/hardware machine from start to finish, user B can zoom in and monitor activities occurring on a node with address 00005, while user C can zoom in and watch activities occurring on token ring 00012)
- A monitor can run on the same machine as the simulation or can run on a remote machine. (The monitor and simulation can run on heterogeneous systems, however they must be able to connect via a network. Also the monitor must know the name of the machine the simulation is running on. The communication can occur within the *UNIX* internet domain.)

One of the main limitations that the system currently has is that only single instances of reporting windows are available. What this means is that if a user wants to track two or more activities of the same kind he will not be allowed. An example of this limitation is if a user wants to zoom in and monitor the activities of two or more rings. The current implementation allows only one such window to be opened at the same time. The basic underlying control mechanism and functional mechanisms have been built for each monitoring activity, therefore this extension should only be a matter of semantics.

A second system limitation is that the monitor can only track one process. The software/hardware machine being built currently allows only single process execution this monitor limitation can therefore be tolerable. The only mechanism needed to allow

Real-TV to allow tracking of multiple monitors would be to design the algorithms. Again, because the underlying control and functional mechanisms are available this extension should be able to be implemented with ease.

A third limitations is that the monitor can only connect to one simulations per monitor instance. Future extensions may allow one monitor instance to connect to several simulations at once and allow users to view each simulation in a window.

6. Real-TV's Design Overview

Real-TV is a distributed real-time monitor that provides an unobtrusive mechanism for obtaining pertinent data from an executing simulation. It is designed to be used as a front-end to the simulated software/hardware machine currently under development. While RTP/L^3's software/hardware machine is executing, Real-TV will not interfere with any outcomes and will not add any overhead to the processes being monitored or tracked. This guarantees that Real-TV will not jeopardize the predictability that the schedulability analyzer calculated. When Real-TV is actively monitoring a simulated system or if it is not, performance and deadlines will be unaltered and outcomes will be unchanged.

When Real-TV is transformed into a real-world system it will become another device on the network ring. Real-TV will then likely be an interactive nonreal-time component that may physically work like any other device that is currently connected to the network. It will request services and be serviced similarly. When the other components of the project, such as communications or the keyboard, are completed we will have better ideas on how to implement Real-TV's reporting mechanism into a real-world model in a unobtrusive manner.

In order for Real-TV to work properly, it needs to obtain relevant data from the simulation. To fulfill these requirements we needed to build a facility that would allow two or more processes, monitors and simulations, to communication regardless of there host machine location. The following sections give an overview of several of these underlying support mechanisms that were built. Each ensures reliable and efficient communication between the *monitor* and *simulation.*

We designed and built a *communication reporting mechanism.* This section is based on work done concurrently while Real-TV was being developed. The author of this paper and Richard Meyer designed and developed the following reporting mechanism. See [10] for a complete reference.

There are two main components that make up the reporting mechanism. The *reporter* and the *gatherer*. The former, is responsible for getting the data and reporting it to the gatherer. The monitor and simulation are both distributed systems, therefore the reporter is also responsible for sending the data to the correct gatherer.

The later element, the *gatherer*, is located on the other end of the communication mechanism and its primary duties are to gather the information sent from the *reporter*. It listens to the communication channel, when it receives data from the channel it will analyze the data and pass the request to the correct component. It is then responsible for returning the data to the correct requestor.

We needed to build a reporting mechanism which was:

- easily callable by both monitor and simulation
- portable
- capable of asynchronous communication
- capable of handling multiple connections

We chose to implement the reporting mechanism via *UNIX* internet stream sockets, and *UNIX* shared memory. The socket IPC mechanism is used to communicate between a monitor and a simulation via the TCP/IP protocol. While the shared memory is used to pass data between the listener-server entity (see below) and the monitor or simulation. The monitor and simulation both use this communication model to provide asynchronous communication in a predictable, reliable and efficient manner.

6.1. Simulation Name Server

The monitor and simulation can be executing on different host machines, this made it necessary to design a *name server*. The primary responsibility of this server is to assign unique keys to the simulation. These keys are later used by both the simulation and monitor to setup a communication path in order to exchange relevant information.

When the RTP/L^3 software/hardware simulation is started it is responsible for registering with a Simulation Name Server (*SNS*). The simulation will register its simulation name and its machine name. It will then be given a unique socket port number by the *SNS*. This procedure will ensure that communications, directed to a machine running multiple monitors or one running multiple simulations, are transported correctly and to the correct simulation or monitor. The socket port number will then be used to establish a communication socket between a simulation and a monitor that want to communicate.

When a monitor wants to connect to a running simulation it can request, from the *SNS*, a list of all the simulations currently active. The monitor will receive a list of simulation names, simulation host computer names, and a list of unique port ID's which correspond to each simulation. This socket port number will then be used to establish communications between a simulation and a monitor. This procedure is outlined in figure 3 .

The *SNS* will be running on a well-known machine with a well-known port number. These parameters will be agreed upon and publically known. The *SNS* is capable of running on any machine within the *UNIX internet domain* and may in fact be executing on different machines at different times. This feature allows for easy portability.

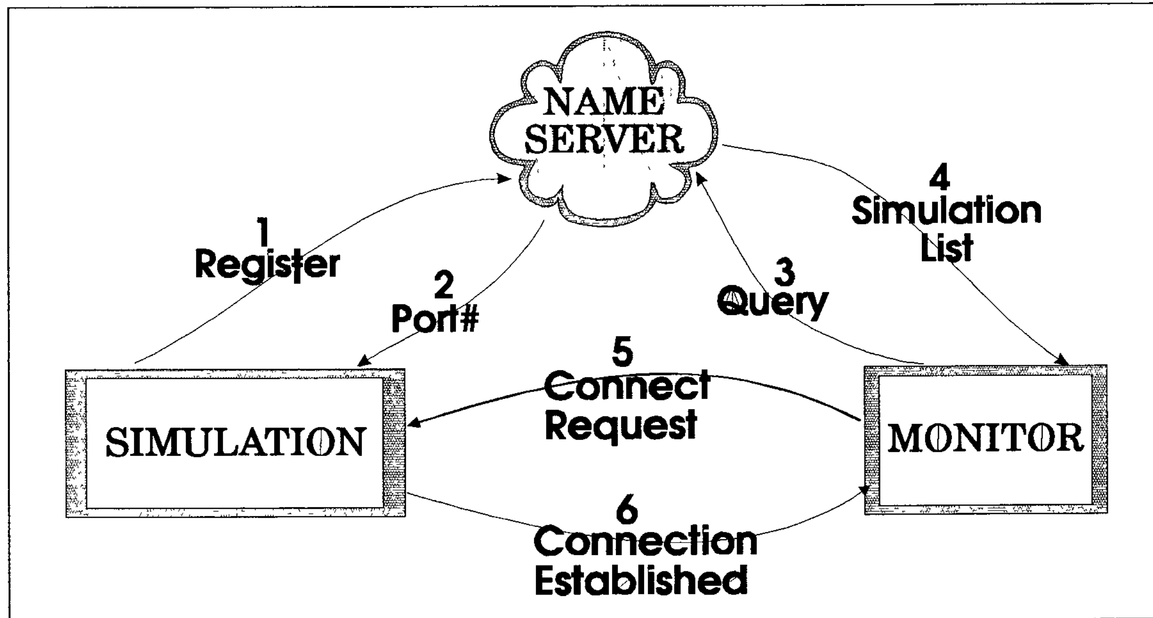

Figure 3

When the simulation is ready to register its services it will connect to the *SNS* via a UNIX socket. This connection is kept alive throughout the simulation and is again used to notify the *SNS* upon simulation termination. This ensures that the SNS is kept up to date with the simulations that are currently running and gives prospective monitors accurate active simulation lists. It is the simulations responsibility to register its services and to unregister its services upon terminating.

The *SNS* is capable of handling these three types of requests:

- Simulation Registry
- Simulation Unregistry
- Monitor Query

When a simulation registers its services it is known as *simulation registry*. When that same simulation is about to terminate, for what ever reason, it must unregister its services. This is known as *simulation unregistry*. Finally, *monitor query* is the procedure by which a monitor obtains a list of all simulations that are currently running including their unique port number identifier. The monitor can then select the simulation that it would like to monitor and connect to it correctly.

The main need for this *SNS* is to help alleviate the naming discrepancies that may be incurred due to multiple simulations or multiple monitors running on the same machine. It is also a nice feature which enables prospective monitors to see which simulations are currently running. One scenario may be that several simulations may be running and users may be assigned a problem. The user may want to run the problem on several different simulated software/hardware systems each with a different hardware configuration. The *SNS* gives each user a easy, flexible mechanism to locate the simulations to execute their problem set.

6.2. Token Ring Network Topology Table

RTP/L^3's hardware bus is based on a token-ring network, hence we needed to devise a way to represent it. Displaying the network topology of the underlying hardware system graphically without such a support feature would not provide a real world view. We needed to design a method that would allow users a quick and easy way to change network layouts and at the same time give a physical layout for Real-TV and the simulation to use.

6.2.1 Our Representation

We defined a format for a flat file called Network Topology Table (NTT). The NTT will be used for two specific areas during the simulation. First, it will be used to initialize the simulation with the layout of the hardware subsystems. Based on the network topology represented in this file we provide the users with an easy method for simulating different network configurations. Very easily they can create an NTT and execute a simulation passing the newly created NTT as a parameter. Changing the contents of this file or starting the simulation with a different NTT allows many combinations of topologies to be tested and analyzed for performance.

The second use for the NTT is for Real-TV. When the monitor connects to a simulation the first data they exchange is the NTT. The simulation will pass this information via the reporting mechanism described earlier. The monitor will then be able to build a graphical representation of the underlying network.

6.2.2 Real World Representation

In a real world token-ring network, obtaining a network topology is fairly straight forward. When the network comes up each node is responsible for announcing its presence and voicing its network address. Each node knows of his neighbors and their associated addressed. This information would be polled by a designated node on the network who would be responsible for gathering this information for the monitor. The exact protocol may differ for different vendors token ring networks.

6.2.3 NTT for the Simple Token Ring

Following is an example NTT for the network represented in figure 4. The node type column represents a symbolic identifier to describe one type of node (component) on the ring[4].

Node Type
1
3
-1
5
0 <---End of Network #1
-1
3
6
5
0 <---End of Network #2

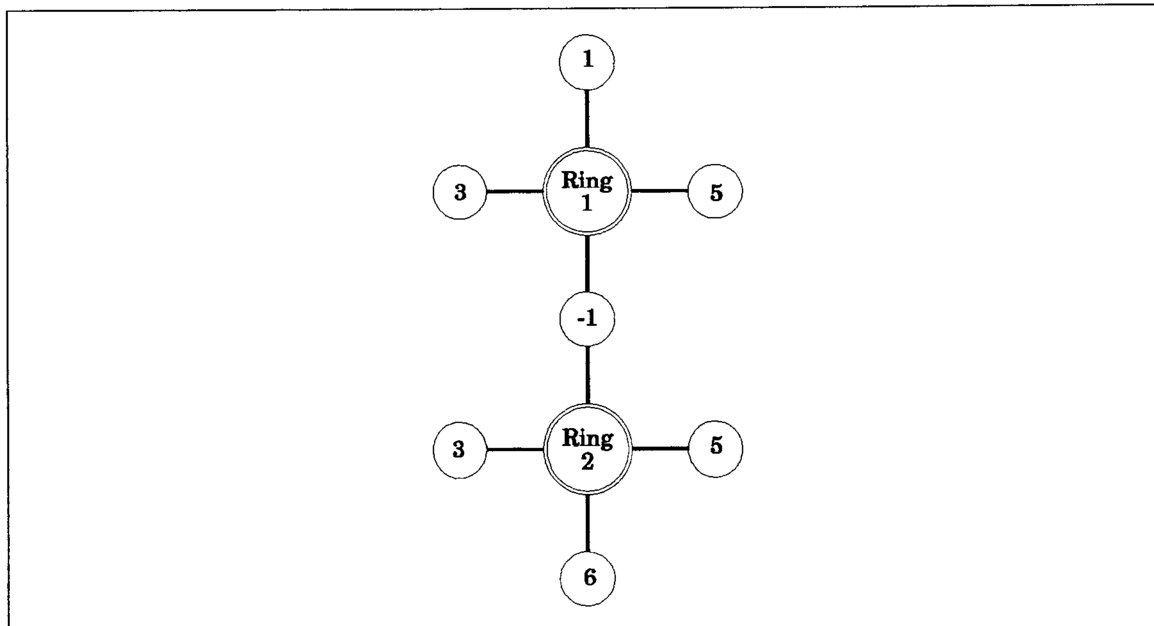

Figure 4

4 A zero (0) delimiter is used to separate disparate rings on the network

6.3. Component Legend

Component	Identifier
CU	1
ALU	2
Register	3
PSW	4
I/O Processor	5
Main Memory	6
Secondary Memory	7
Command Proc.	8
OS Kernel	9
End of Ring	0
Bridge[5]	< 0

CU - The Control Unit provides a means of controlling the processing of instructions.

ALU - The Arithmetic Logic Unit, executes mathematical instructions. These instructions either involve numerical operations (arithmetic) or non-numerical (logical) operations, such as program branching and symbolic processing.

Register - Registers are those components which provide a storage device for words.

PSW - A Program Status Word, stored in a special register, indicates program status, interrupts that the CPU may respond to, and address of the next instruction to be executed.

I/O Processor - An Input/Output Processor is a special-purpose processor used exclusively to control input-output operations.

Main Memory - Main Memory is a large fast memory used for program and data storage during computer operation.

Secondary Memory - Secondary Memory also called auxiliary memory, is generally much larger in capacity than Main Memory but slower. It is used for storing files

5 Bridges are represented in the SGT as negative integers

which are not continually accessed by the CPU. It also serves as an overflow memory when the capacity of main memory is exceeded.

Command Processor - A Command Processor reads an instruction and based on what is to be performed, distributes the work to the respective resources.

OS Kernel - The Operating System Kernel is a portion of the operating system that remains in main memory and consists of the most frequently used part of the operating system.

Bridge - A Bridge is used to connect a collection of separate network segments. Bridges examine each frame and only forward those that need to reach the other segments.

To identify bridges that connect two disparate rings the same negative integer is used in the NTT. *Example: In the above network layout Ring #1 and Ring #2 are connected via a bridge (-1). The bridge has separate addresses per network ring. Also note that the next set of bridges would be labeled -2. -3, -4 respectively.*

Ring #1 contains a CU, a Register, a Bridge, and a I/O processor. Ring #2 contains a Bridge, a Register, Main Memory, and a I/O processor.

Note that the NTT does not contain any notion of node addresses. When a network topology is modified it will make it easier for placing new components in between existing nodes without upsetting node addressing. It will be the responsibilities of the data communications portion of the simulated hardware system to build network topology and to assign each node per ring a unique network address.

6.4. Monitor Listener-Server Entity

This section describes the method used to setup the reporting mechanism in both the monitor and simulated software/hardware. Upon initialization of the graphical software monitor and the simulation each respective process is responsible for setting up and initializing a piece of shared memory. The shared memory is used to pass data between the controlling process[6] and the listener-server process. It also initializes a signal handler of the type *SIGUSR1*. The signal handler will be responsible for handling reading of the shared memory data, properly dissecting it and then finally servicing the request. Signals are used to interrupt the controlling process and notify it that data is ready to be read. This data can either be a monitor request, originating at the monitor, or can be simulation data, information pertaining to the requested information. After this information is retrieved only then may it resume its previous activities.

6 Controlling process is either the monitor or the simulation

When the preliminary initialization is finalized the listener-server (*LS*) (or server or socket server) process is forked off using the *UNIX* fork() system call. The *LS* process is then responsible for initializing its own end of the shared memory and for setting up the main communication socket. This is the socket which all communication is directed through. It is also the *LS's* responsibility to register and unregister with the *SNS*. Finally, the LS must establish and terminate socket connections between one through several communicants who may connect in an unpredictable manner throughout the life of a monitor or simulation. See figure 5 for a visual representation.

When a new connection is detected by the *LS* it is up to it to process these requests both efficiently and with a guarantee. This is done by polling all the connected sockets and servicing them in a timely fashion. Each connection that does have a pending read or write will be serviced until that read or write request is consummated. The *LS* protocol is structured so as not to interrupt the servicing of the current monitor or simulation requests.

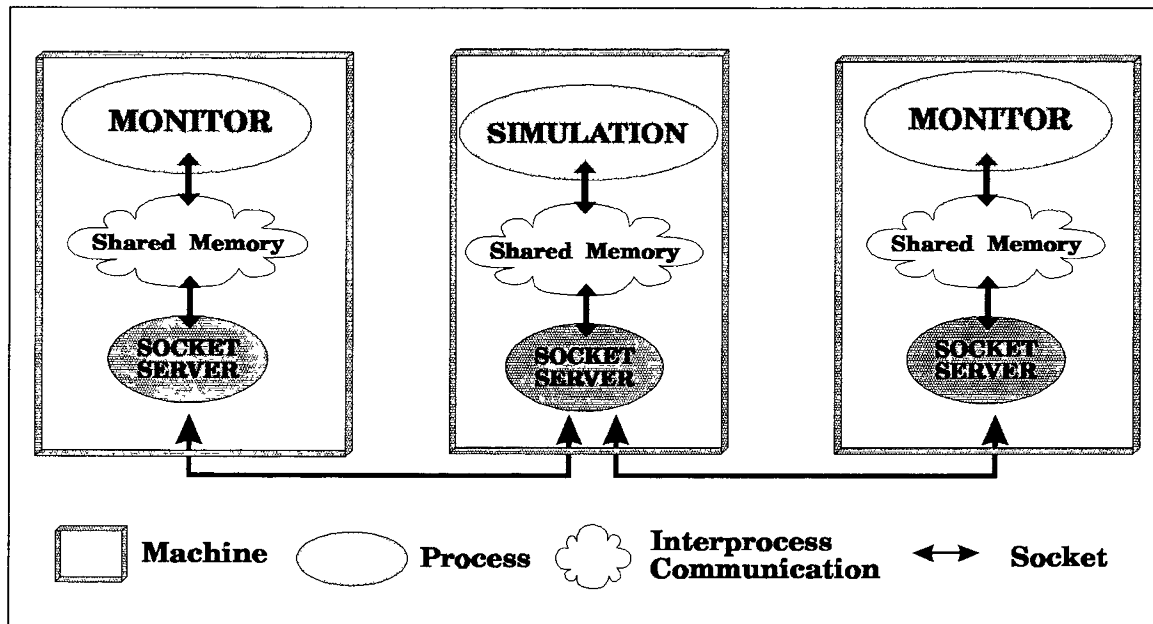

Figure 5

6.5. Monitor/Simulation Initialization & Setup

Both the monitor and simulation will call one function to initialize and setup the reporting communication mechanism. A call to the function init_rpt_mech() (initialize reporting mechanism) will (see figure 6):

- Setup the SIGUSR1 signal handler which signals that a request has arrived

- Setup the shared memory
- Fork() and exec() the LS

Once the LS is fork'ed and exec'ed the control returns back to the parent process, which in this case is the monitor or simulation. Then the child process, or the LS, is responsible for:

- Initializing and setup of the LS
- Setting up the LS's shared memory pointer
- Connecting and registering with the SNS (simulation only)
- Polling the sockets for new connections and new service requests
- Reading from the socket, passing the request to the simulation or monitor through the shared memory
- Finally, forwarding the serviced request data back to the originator

During the entire lifetime of the monitor and simulation the LS is accountable for establishing new connections to the simulation, reading requests initiated by the monitor, and sending back the serviced requests back to the monitor.

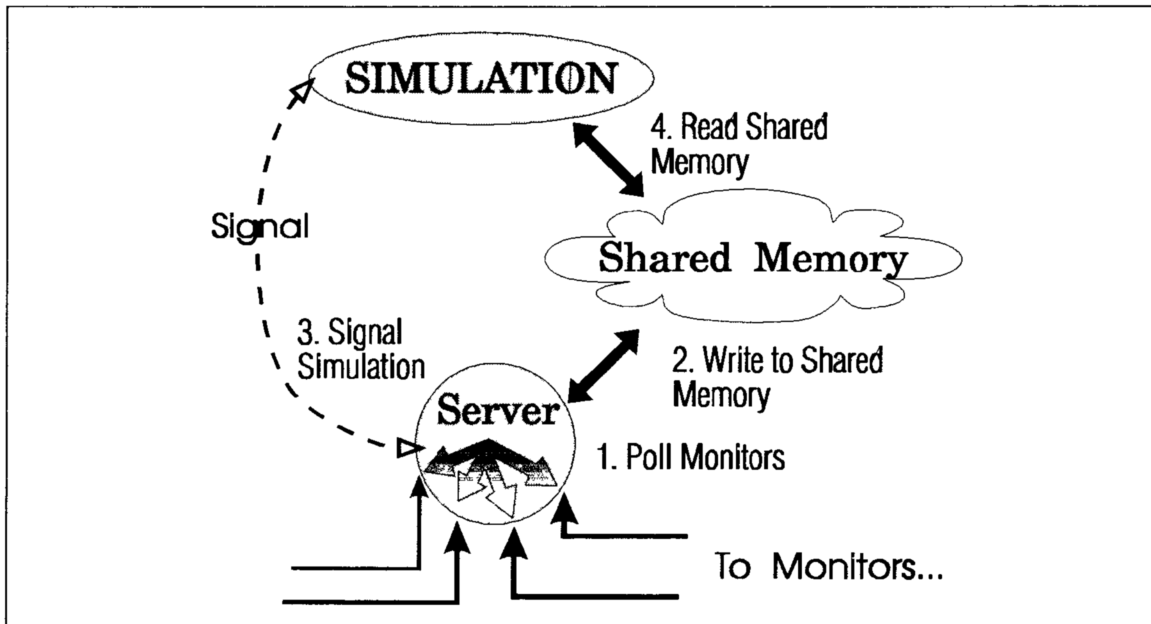

Figure 6

6.6. Reporting Protocol within the Software Simulation

The simulation only has to be concerned with the functional interface provided to it for accepting connections and sending messages to the monitor(s). There are four function calls provided:

- **init_rpt_mech** - Initialize Reporting Mechanism. Initializes data structures, fork()'s the listener process, and creates shared memory for IPC. This needs to be called only once by the simulation.
- **continuous monitor** - provides the mechanism to gather data and send messages to monitors viewing data in continuous monitor mode (explained below). This function is called once per simulation clock time.
- **msg_driver** - The interrupt driven reporting mechanism (IDRM) that is asynchronously invoked by a signal from the listener process. The IDRM determines which message was received then processes and sends an appropriate reply.
- **send_monitor** - Sends the requested information back to the monitor.

6.6.1 Types of Messages

There are two types of messages: Read/Write *(RW)* messages and Continuous Monitor *(CM)* messages. These two types of messages correspond to the two choices the monitor offers in viewing data: *Snapshot and Continuous mode.*

An *RW* message type is a request made by a monitor to a simulation to send current information about a single object (i.e. a node or bridge in the network).

The protocol is as follows:

- The monitor sends an RW type message; the monitor's listener process waits for the reply
- The request arrives at the simulation's listener
- The request is transferred to the simulation via shared memory and handled by the IDRM
- The listener blocks until the reply is placed in shared memory and is signaled by the simulation
- The listener, upon being signaled, fetches the reply from shared memory and sends the reply (message) to the monitor.

- The monitor's listener process continues upon receiving the message

By following the above protocol, when many RW type messages are queued at a simulation, they will be serviced one at a time. Thus synchronization has been achieved -- it is not possible for one monitor's request to interfere with another.

A *Continuous Monitor* message is a message that is sent to a monitor, at regular intervals, which describes the current state of an object. To establish the Continuous Monitor mode, the monitor sends an initial CM type message informing the simulation that it wants to register the CM mode of operation for a particular item.

The simulation, upon receiving a CM type message, registers the monitor and notes the item to be monitored. Whenever the continuous monitor function is called[7] by the simulation, the appropriate reply will be generated and sent.

The protocol is as follows:

- The monitor sends a CM type message describing the item to be continuously monitored
- The request is serviced by the simulation's listener
- The request is transferred to the simulation via shared memory and handled by the IDRM
- The listener, upon being signaled, fetches the reply from

shared memory and sends the reply (message) to the monitor

- The simulation, at pre-defined regular intervals, performs step 4 automatically until the monitor terminates this CM request

6.6.2 Transferring the Network Topology to a Monitor

After establishing communication with the simulation the monitor will request the NTT from the simulation. The *RW* protocol is altered slightly to accommodate the transfer of the *Network Topology Table* file since many messages must be sent to complete the request. When the simulation listener sees that the request was for the Network Topology Table (NTT), it cycles through the following protocol:

- Monitor sends a NTT request
- Simulation replies with NTTSTART and first NTT entry
- Monitor sends NTTNEXT request
- Simulation replies with NTTENTRY and next NTT entry

---

7 The simulation will call this function as needed--every clock tick, every tow clock ticks, or whatever is deemed appropriate

- Repeat step 3 and 4 until the last entry in the NTT is reached
- Simulation sends NTTEND and the last NTT entry

NTTSTART, NTTNEXT, and NTTEND are placed in the reportID field within the reporting mechanism data structure (described later).

NTTSTART -- Indicates the first entry of the NTT

NTTENTRY -- Indicates a NTT entry

NTTEND -- Indicates the last entry in the NTT

6.6.3 Resolving Reporting Functions

Since there are possibly several instantiations of one node type (i.e. CU, ALU, Main Memory, etc.) a cross-reference table is used, in the simulation, to resolve which reporting function should be called. The table contains the address of the node, the object, and a pointer to a function. The address corresponds to the 10 digit network address. The function pointer points to the actual function that will properly generate the message request packet to be sent to the monitor.

The reporting functions are provided by the programmers of the objects (node types). All instantiations of the same object will report the same "set" of information. A typical reporting function will gather the data for a particular instantiation, write the data to shared memory, and signal the listener process.

6.7. System Design - Low-level

All relevant data pertaining to the simulation and the current activities is kept in individual data structure. Each structure is defined below along with a brief description of each member.

The monitor, once started, will continuously loop processing user requests. These requests are made by moving the mouse pointer and pressing the mouse button on an appropriate control. These requests range from selecting a menu choice that monitors and tracks a process to clicking the mouse on a node to zoom in on the current information pertaining to that node.

The first function performed by the monitor is to connect to the simulation and receive the network topology description. This data is stored in an internal data structure that will be used each time to repaint the network graphically and when a users selects a component of the network, such as a CU. When the user selects a node, all the relevant data about the screen position and current status of the node is kept by this data structure (see Node Data Structure below).

Current monitor status, such as what windows are currently active, are kept in the monitor data structure. When the user requests an activity the program checks to see the current status of this activity and takes the appropriate actions. If the activity is currently active the window is updated to reflect to new information. If the activity is not currently active then the proper flags are set and a window is popped up to show the relevant information.

When the user clicks on a node, such as a CU, the monitor will fill the reporting mechanism data structure with the appropriate request data and pass it to the server socket. This requested data will then be returned by the simulation in the same data structure. Once the monitor receives this data it is up to it to dissect it and display the correct information to the user within the proper window.

The following section describes the most important data structures that are used to control Real-TV. Each member plays an important role in helping Real-TV to track current activities and to track the current status of user activity.

6.8. Data Structures

6.8.1 Monitor Data Structure

The main data structure used to control Real-TV is MonitorData. It primarily tracks the states of the different activities that the user selects to monitor and manipulate. The structure is as follows:

```
struct Monitor{
        int Nodes
        int Rings
        int Bridges
        int currentMenuSet
        int menuItems
        int buttonSelected
        int trafficWindow
        int trafficButIndex
        int zoomWindow
        int zoomButIndex
        int processWindow
        int processButIndex
        int tokenWindow
        int tokenButIndex
        int aboutButIndex
        int aboutWindow
        int legendWindow
        int legendButIndex
        int bridgeWindow
        int bridgeButIndex
        int ringSelected
        char addSelected[NODEADDRESS]
        int typeSelected
        int tokenNewView
        int quit
}
```

These three variables are used mainly when building the graphical representation of the underlying network.

Nodes -- Indicates how many elements are on the network including ring delimiters.

Rings -- Indicates how many rings are on the entire network.

Bridges -- Indicates how many bridges are on the entire network.

These variable describe the current menu set being displayed and are used when repainting the menu.

currentMenuSet -- Indicates the index of the menu set currently being displayed on the menu line. This variable is used to repaint the menu when the X Window is exposed.

menuItems -- Indicates the number of menu items in this menu set.

buttonSelected -- Indicates what button the user pressed from the main menu.

These variables track open windows, and their associated buttons.

xxxWindow -- This set of variables with the Window extension are flags that indicate if that window is already open.

xxxButIndex -- This set of variables with the ButIndex (button Index) extension are indexes into the structure that tracks what buttons have been registered per dialog box. Each registered button has an x,y coordinate, a string, and a function attached to it.

The last set of variables indicates the type of node the user selected when he pressed the mouse button for node, ring, or bridge statistics.

ringSelected -- Indicates what ring the node is a part of when a user presses the mouse button while the pointer is

within a network node.

typeSelected -- Indicates the type of node selected: CU, ALU, bridge, etc.

addSelected -- Indicates the address of the ring, node, or bridge selected.

Finally, **quit** indicates if the user selected quit from the main menu.

6.8.2 Node Data Structures

This data structure tracks all the nodes (CU, ALU, registers, etc.) on the network. The information contained gives exact coordinates of each node and other relevant data. This structure is used when the network is drawn on the screen as well as when the user selects a node from the graphical diagram.

```
struct NodePtr

{
        int x
        int y
        int gender
        int type
        char address[NODEADDRESS ]
        int ring
        int nodeMarked
        int processState
}
```

x -- Indicates the center X coordinate of the circle representing the node.

y -- Indicates the center Y coordinate of the circle representing the node.

gender -- Describes the nodes gender: Node, Ring, or Bridge.

type -- Indicates the nodes type: ALU, CU, Register, etc.

address -- Indicates the address of the node.

ring -- Indicates what ring number the node is connected to.

processState -- Indicates which process tracking state the current node is in. This is used when repainting the topology. The three states are Controlling CU, Processing, or Token.

The following data structure is made up of the data structure just described above. It is used to draw the network topology to the screen.

```
struct RegNodes

{
```

```
        int numberOfNodes
        int currentIndex
        struct NodePtr NodeInfo[ MAXNODES + MAXRINGS ]
}
```

numberOfNode -- Indicates the number of nodes on the network.

currentIndex -- Indicates the index into NodeInfo. This is the node currently being processed when the network is being drawn on the screen.

NodeInfo -- An array of NodePtr structures. Describes each node on the network as described earlier.

6.8.3 Ring Structures

```
struct ringData {
        int ringTop
        int processed
        int centerX
        int centerY
        double radians
        double currentRadians
        int lineLength
        int LineX
        int LineY
        int NodeX
        int NodeY
        int CircleX
        int CircleY
        int BridgeX
        int BridgeY
}
```

The variables in this structure describe the positions of all the rings on the network.

ringTop -- Index of the first ring drawn.

processed -- Indicates if this ring has been finished being processed. Because the network topology is drawn recursively we need to track which rings have been processed.

centerX -- Center X coordinate of this ring.

centerY -- Center Y coordinate of this ring.

radians -- Angle between node connectors on this ring. This angle is in radians. Because each ring may have a different number of nodes connected to it each node on the ring is spaced different from other rings. The number of nodes is divided by a full circle and an equal distant angle is calculated for each ring.

currentRadians -- The current angle to draw the node connector.

lineLength -- Length of the node connector line, from the ring to the node.

LineX -- Point X where the node connector ends, at the bridge.

LineY -- Point Y where the node connector ends, at the bridge.

NodeX -- Point X where the node connector connects on the node.

NodeY -- Point Y where the node connector connects on the node.

CircleX -- Point X where the node connector connects to the ring.

CircleY -- Point Y where the node connector connects to the ring.

BridgeX -- Point X where the node connector connects to the bridge.

BridgeY -- Point Y where the node connector connects to the bridge.

6.8.4 Reporting Mechanism Structure

```
struct RptMech {
        int reportType
        int reportID
        int value1
        int value2
        int value3
        int value4
        int value5
        char text1[ RPTTEXT ]
        char text2[ RPTTEXT ]
        char text3[ RPTTEXT ]
```

```
        char text4[ RPTTEXT ]
        char text5[ RPTTEXT ]
}
```

The reporting mechanism described earlier uses this structure to pass relevant data from the simulation to the monitor and back. The value and text fields below are used for different data depending on the report type requested.

reportType -- Indicates if the reporting mode is Read Write (RW) or Continuous.

reportID -- Indicates the type of report the monitor needs information about.

value1 - value5 -- Generic fields used for numeric return values.

text1 - text5 -- A generic field used for character return values.

7. Module Specification

This section describes several **selected** modules. It briefly describes what function each module performs and provides the interface to the module. Most of the functions described below have four common parameters defined. The first three are used when calling the X Window functions.

thisDisplay -- describes what display to write output to.

thisWindow -- indicates what window to write output to or take input from.

thisGC -- describes the Graphic Context for the window.

MonitorData -- described earlier.

7.1. File: Topology.c

7.1.1 Function Name: CheckNodePress

Function Interface

Display **thisDisplay**

Window **thisWindow**

GC **thisGC**

int **x, y** -- X, Y coordinates of the mouse pointer when the mouse was pressed.

struct Monitor **MonitorData**

Functions Performed

Scans the list of registered nodes (rings, nodes, or bridges) and returns whether one was pressed.

7.1.2 Function Name: RegisterNode

Function Interface

int **centerX, centerY** -- X, Y coordinate of the center point for the node.

int **gender** -- Node, Ring, or Bridge.

int **type** -- If a the gender is a node then it indicates what type of node (ALU,CU...).

int **ring** -- The address of the ring it is connected to.

char * **address** -- The address of the node being registered.

Functions Performed

Register the coordinates for the node and initializes all values.

7.1.3 Function Name: PaintTopology

Function Interface

Display **thisDisplay**

Window **thisWindow**

GC **thisDisplay**

struct TopologyData **Topology** -- Stores relevant data about the topology.

struct Monitor **MonitorData**

struct BridgeData **Bridges** -- Stores list of bridges already processed.

Function Performed

Gets called everytime that the graphical representation of the network needs to be redrawn. It paints the center Ring and then calls DrawTop() to recursively paint the rest of the topology. It also clears the previous data from the data structures topology and bridges, and clears the window.

7.1.4 Function Name: DrawTop

Function Interface

Display **thisDisplay**

Window **thisWindow**

GC **thisGC**

struct topologyData **Topology[]** - Stores relevant data about the topology.

struct Monitor **MonitorData**

struct ringData **RingData[]** -- Stores relevant data about the rings and their coordinates.

int **Current** -- Index of the current node we are inspecting.

int **Tmp** -- Index used to walk through the list of nodes.

int **RingTop** -- Index of the top of the current ring we are processing.

struct BridgeData **Bridges** -- Stores list of bridges already processed.

int **Ring** -- Ring currently processing.

int **BridgeMatched** -- ID of the bridge just matched.

Functions Performed

Recursive function that draws the network topology of the underlying hardware system.

7.1.5 Function Name: DrawChildRing

Function Interface

Display **thisDisplay**

Window **thisWindow**

GC **thisGC**

struct ringData **RingData[]** -- Stores relevant data about the rings and their coordinates.

int **Ring** -- Index of the ring to paint.

int **CurrentRing** -- Index of the parent ring.

struct parentData **ParentData** -- Relevant data of the parent Ring.

Function Performed

Given a parent ring and its relevant data this function will draw a child ring with the appropriate length connector lines and in the correct place.

7.1.6 Function Name: DrawBridge

Function Interface

Display **thisDisplay**

Window **thisWindow**

GC **thisGC**

struct ringData **RingData[]** -- Stores relevant data about the rings and their coordinates.

struct topologyData **Topology[]** -- Stores relevant data about the topology.

int **Ring** -- The index of the ring that the bridge will extend from.

int **currentNodePtr** -- Index of the node from where the bridge will extend.

Function Performed

Given a ring and its relevant information this function will draw a bridge that will connect this ring to another ring.

7.1.7 Function Name: DrawNode

Function Interface

Display **thisDisplay**

Window **thisWindow**

GC **thisGC**

struct ringData **RingData[]** -- Stores relevant data about the rings and their coordinates.

struct topologyData **Topology[]** -- Stores relevant data about the topology.

int **currentNodePtr** -- Index of the node from where the node will extend.

int **Ring** -- The index of the ring that the bridge will extend from.

Function Performed
Given a ring and its relevant information this function will draw a node that will extend from the ring.

7.2. File: MISC.C

7.2.1 Function Name: Unregisterbutton

Function Interface
int **buttonIndex** -- Index of the button to unregister.

Function Performed
This function will unregister (remove the entry from the data structure) of the button whose index is passed. The button is the button used in a dialog box to allow the user to select a function to be performed within the dialog box.

7.2.2 Function Name: RegisterButton

Function Interface
int **buttonX, buttonY** -- X,Y coordinates of the upper left corner.

int **Width** -- Width of the button.

int **Height** -- Height of the button.

char ***String** -- String to appear inside the button.

void **(*Function)**() -- Name of the function to call when the button is pressed.

Function Performed

This function will register the button (enter the relevant data into a data structure). This data is used when a user presses the mouse to detect if the pointer was within the button registered.

7.2.3 Function Name: DetectButtonPress

Function Interface

Display **thisDisplay**

Window **thisWindow**

GC **thisGC**

int **thisX, thisY** -- X, Y coordinate of the pointer when the mouse button was pressed.

struct Monitor **MonitorData**

Function Performed

This function will check the registered buttons and return the index of the registered button to the caller. The caller will then have access to the relevant data for this button.

7.2.4 Function Name: PaintZoomWindow

Function Interface

Display **thisDisplay**

Window **thisWindow**

GC **thisGC**

struct Monitor **MonitorData**

Function Performed

This function is called each time the node zoom window needs to be updated with the latest information as received from the simulation.

7.2.5 Function Name: PaintTokenWindow

Function Interface

Display **thisDisplay**

Window **thisWindow**

GC **thisGC**

struct Monitor **MonitorData**

Function Performed

This function is called each time the network statistics window needs to be updated with the latest information as received from the simulation.

7.3. Equipment Configuration

The monitor will run on any X Window based system. This include Sun Workstations, DEC Workstations, HP Workstations, and PC based systems running UNIX and the X Window System. With the support of the underlying X Window System it should run on any network supported by X. These systems must also have a mouse to manipulate the objects and to select monitoring activities.

7.4. Implementation Language & Window System

RTP/L^3s monitor was written entirely in the C programming language. The system runs under the X Window System environment. The X library, known as Xlib is the C language programming interface to Version 11of the X Window System. This library enables a programmer to write applications with an advanced user interface based on windows on the screen, with complete network transparency, that will run without changes on many types of workstations and personal computers [13].

The X window system was developed jointly by MIT's Project Athena and Digital Equipment Corporation. Currently, Version 11 is available and is used in this project.

8. System Evaluation

At the time this paper was written all modules were tested stand alone. The monitor and simulation have not yet been integrated, therefore each module that needed simulation information, has a special test driver written that will simulate data that would normally arrive from the simulation. There are several components of the monitor that required such a test driver: Network Topology Table, Node Zoom, Ring Statistics, and ProcessTracking.

8.1. Network Topology Table

To test the graphical display of the network topology I needed to simulate receiving the NTT from the simulation. The test driver reads a ascii flat file that describes the network topology. It is in the same format as would be the data that would be sent by the simulation. When the monitor and simulation are integrated the only changes that need to be made to accommodate the real data is to read from the reporting mechanism rather than the flat file. This change should be a minor one.

When the monitor starts up it would normally request from the simulation the network topology table, instead the monitor reads this information from the file described above.

8.2. NodeZoom

When the user wants to zoom in on a node within the network topology the simulation would respond with the appropriate data as requestedby the monitor. To facilitate the zoom option with data to be displayed a test driver just displays hard coded values. These values will normally be requested by the monitor and the simulation would respond with the current node statistic. This function will need to be rewritten slightly when the monitor and simulation are integrated.

8.3. Ring Statistics

The continuous data that this window needs is currently being read from a flat ascii file. If the simulation was feeding us with information to monitor the selected ring it would send us this CM type response every clock tick or some other periodic frequency. To simulate continuous monitoring of the ring, each time the mouse button is pressed the next input values are read from the file. When the simulation and monitor are integrated once the monitor requests this information from the simulation it will

receive the relevant data and it would appropriately update the screen with the latest information. This change, again, should be simple.

8.4. ProcessTracking

Process tracking requires relevant process information at every clock tick or some periodic frequency. Therefore, to simulate this without actually receiving this data from a simulation, the test driver reads the information from a flat ascii file, like above, whenever the user presses the mouse button. When the simulation and monitor are integrated this information will be feed to the simulation by the monitor at regular intervals. The monitor would then update the network graph to indicate the current status of the process being tracked.

8.5. Other modules

All other modules which did not require any test data were tested and debugged thoroughly, however, like with any large system bugs may have gone undetected. These modules were called with different parameters and tested in stress situations. All discovered bugs were noted and immediately resolved. The real test will be when the simulation and monitor are integrated and users begin to use all the features of the monitor and simulation.

8.6. Special Cases

Special cases such as:

- Rings without components (nodes)
- Rings with many components (nodes)
- Simulation process crashes
- Reporting mechanism

were tested by running the monitor with dozens of different NTT's. Each NTT was configured with a different number of components per ring, different number of rings and networks with different topologies. These special cases were tested to show how process tracking would be affected. Each ring was assumed to have at least one bridge connected to it. This bridge serves as the connector for disparate rings. Each case was successfully tested for functional correctness.

The other special cases of simulation crashes and reporting mechanism crashes are resolved through the alertness of the reporting mechanism. If the socket connec-

tion no longer exists, the monitor is alerted to this and is gracefully exited with out causing delayed reactions to user responses.

9. Future Extensions

Some of the most important future extensions should be: tracking multiple processes, monitoring processes through critical paths, transferring the monitor to a unobtrusive real-world component, allowing multiple instances of the same reporting window, and the capability to monitor multiple simulations with the same monitor.

One way to provide a real-world unobtrusive monitor may be to base the reporting mechanism gatherer on a separate token ring connected to each node on the network. This alleviates the contention that may be caused by sending monitoring data over the same communication wire as the bus of the system uses. Another way is to treat the monitor as just another component on the network ring. All servicing would be similar to the way current components are serviced.

10. Conclusion

This paper has described briefly the intent of the RTP/L^3 project and has provided the user with information about Real-TV. From the Previous Work section and the details given about Real-TV one can understand why such a tool is needed. The distributed nature of the project made it more difficult to design and develop. Secondly, because of the "predictability" the system is built upon, the design of the reporting mechanism was not as straight forward as first conceived. These issues added to the overall development time and because of them we have not been able to thoroughly integrate the monitor and the simulation as of this writing.

The RTP/L^3 system described will undoubtedly be a very useful tool for providing students with a teaching platform and for verifying fundamental research in the field of real-time computing. Real-TV will provide users and researchers with the tool required to measure system performance, evaluate different hardware configurations and provide a level of system insight. Future extension to the project will migrate any knowledge gained from this experience and hopefully be able to transform the monitor into a real-world system; thus providing the optimal tool.

We hope this project will become a permanent fixture within the academic world for teaching and research and hope that some of the development tools, such as the schedulability analyzer, can be used within industry.

References

[1] Alexander D. Stoyenko, *Predictable Real-Time Systems: A Challenge for Computer Science and Engineering Curricula,* Department of Computer and Information Science, New Jersey Institute of Technology, 1991

[2] Hideyuki Tokuda, Makoto Kotera and Clifford W. Mercer, *A Real-Time Monitor for a Distributed Real-Time Operating System*, Computer, March 1990

[3] Liba, Svobodova, *Performance Monitoring in Computer Systems: A Structured Approach*, M.I.T. Laboratory for Computer Science, Cambridge, Mass., 1980

[4] Bernhard, Plattner, *Real-Time Execution Monitoring*, IEEE Trans. Software Eng., Vol. SE-10, No. 6, Nov 1984

[5] Jeffrey J.P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen, *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging,* IEEE, 1990

[6] Jeffrey J.P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen, *A Noninvasive Architecture to Monitor Real-Time Distributed Systems*, Computer, March 1990

[7] R.L. Glass, *Real-Time: The 'lost world' of software debugging and testing*, Communication Ass. Computing Machinery, vol. 23, May 1980

[8] Wolfgang A. Halang and Alexander D. Stoyenko, *Constructing Predictable Real-Time Systems,* 1991

[9] Alexander Stoyenko and Lonnie Welch, *A System for Teaching and Research in Predicatable Real-Time Systems*, Department of Computer and Information Science, New Jersey Institute of Technology, 1991

[10] Richard Czop and Richard Meyer, *RTP/L^3 Monitor Reporting Mechanism*, New Jersey Institute of Technology, Newark, NJ, Class Project for Predictable Real-Time Systems, 1991.

[11] Matt Harelicks class presentation on *RTP/L^3s Architecture*, 1991

[12] Lonnie R. Welch and Alexander D. Stoyenko, *A Multicomputer for Real-TIme Software Constructed from Reusable Components*

[13] Nye, Adrian, *Xlib Programming Manual*, O'Reilly & Associates, Inc.

[14] Haban, Dieter and Wybranietz, Dieter, *A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems*, IEEE Transactions on Software Engineering, Vol 16, No. 2, February 1990

[15] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger, *Monitoring Distributed Systems*, ACM Transactions on Computer Systems, Vol. 5, No. 2, May 1987, Pages 121-150

[16] Bernhard Plattner and Jurg Nievergelt, *Monitoring Program Execution: A Survey*, Computer, November 1981

[17] A. D. Stoyenko, *A Real-Time Language with A Schedulability Analyzer*, Ph.D. Thesis, Department of Computer Science, University of Toronto, 1987

[18] A. D. Stoyenko, V. C. Hamacher, R. C. Holt, *Analyzing Hard Real-Time Programs for Guaranteed Schedulability*, IEEE Transactions on Software Engineering, August 1991.

[19] L. R. Welch, *Architectual Support for, and Parallel Execution of, Programs Constructed from Reusable Software Components*, Ph. D. Thesis. Department of Computer Science, Ohio State University, December 1990.

[20] B. Bates and J.C. Wileden, *High-level debugging of distributed systems: The behavioral abstraction approach,* J. System Software, March 1983.

[21] H. Garcia-Molina et al., *Debugging a distributed system*, IEEE Trans. Software Eng., vol SE-10, No 2. March 1984.

[22] P.K. Harter, D.M. Heimbigner, and R. King, *IDD: An interactive distributed debugger*, in Proc. 5th Int. Conf. Distributed Computing Systems, May 1985.

[23] S. H. Jones, R.H. Barkan, and L.D. Wittie, *Bugnet: A real-time distributed debugging system*, Proc. 6th Symp. Reliability in Distributed Software and Database Systems, March 1987.

[24] D. Ferrari and V. Minetti, *A hybrid measurement tool for minicomputer*, Experimental Computer Performance and Evaluation, Amsterdam, The Netherlands: North Holland, 1981.

[25] L. Svobodova, *Online system performance measurements with software and hybrid monitors*, Operating Systems Review, Vol 7, No 4, October 1973.

[26] E. M. Herbst,N. Metropolis, and M.B. Wells, *Analysis of the Problem Codes on the MANIAC*, Math Tables and Other Aids to Computation, Vol. 9, No 49, January 1955.

[27] E.F. Miller, *Bibliography on Techniques of Computer Performance Analysis*, Computer, Vol. 5, No. 5, September/October 1974.

[28] Balzer, R. M. *EXDAMS-EXtendable debugging and monitoring systems*, Proceedings of AFOPS SPring Joint Computer Conference. AFIPS Press, Reston, VA. 1969.

Appendix A - Source Code Listing

```
/* FILE: MONITOR.C
   AUTHOR: RICHARD CZOP
   DATE: 12/13/91
   REAL-TV
*/

/* Include file for Monitor */
#include "monitor.h"
#include "menus.h"  /* defines all menus and menu function calls */
#include <math.h>

#include "process.h"

char AppName[] = { "Real-TV (Real-Time Visualizer)" } ;

Display   *mainDisplay ;
Window     mainWindow, aboutWindow, trafficWindow, zoomWindow, tokenWindow, legendWindow, bridgeWindow ;
Window     processWindow ;
GC         mainGC, aboutGC, trafficGC, zoomGC, tokenGC, legendGC, bridgeGC, processGC ;
XEvent     mainEvent ;
KeySym     mainKey ;
XSizeHints mainHint, aboutHint, trafficHint, zoomHint, tokenHint, legendHint, bridgeHint, processHint ;
int        mainScreen ;
XWindowAttributes windowAttributes ;


struct RegButtons RegisteredButtons ;
struct RegNodes RegisteredNodes ;
struct Process  ProcessData ;
struct P_Test   ProcessTest[ MAXPROCESSTEST] ;

unsigned long myforeground, mybackground ;
int i ;
char text[ 10 ] ;
int done ;

struct Monitor MonitorData ;
struct topologyData Topology[ MAXNODES ] ;
struct BridgeData Bridges[ MAXBRIDGES ] ;
struct BarData tokenBarData[ MAXBARS ] ;

/* DEBUG and TEST Struct */
struct BarData testBarData[ MAXBARS ] ;

main ( argc, argv )
int argc ;
char **argv ;
{
   int TotalNodes=0, TotalRings=0 ;
   int cnt ;
   int rt_value ;

   MonitorData.trafficWindow = FALSE ;
   MonitorData.zoomWindow    = FALSE ;
   MonitorData.processWindow = FALSE ;
   MonitorData.tokenWindow   = FALSE ;
   MonitorData.aboutWindow   = FALSE ;
   MonitorData.bridgeWindow  = FALSE ;
   MonitorData.legendWindow  = FALSE ;

   MonitorData.trafficButIndex = EMPTY ;
   MonitorData.zoomButIndex    = EMPTY ;
   MonitorData.processButIndex = EMPTY ;
   MonitorData.tokenButIndex   = EMPTY ;
   MonitorData.aboutButIndex   = EMPTY ;
   MonitorData.bridgeButIndex  = EMPTY ;
   MonitorData.legendButIndex  = EMPTY ;

   ProcessData.status = TRACKING_OFF ;
   ProcessData.started = FALSE ;
   ProcessData.TestIndex = 0 ;/* DEBUG and TEST ONLY */
   ProcessData.Processing = FALSE ;
   ProcessData.NODE_Index = EMPTY ;
   ProcessData.LASTOP_Index = EMPTY ;

   for ( cnt = 0 ; cnt < MAXNUMBEROFBUTTONS; cnt++ )
       RegisteredButtons.ButtonPtr[cnt].X = EMPTY ;

   /* initialize the bar data to empty */
   for ( cnt = 0 ; cnt < MAXBARS; cnt++ )
   {
       tokenBarData[ cnt ].percent = EMPTY ;
```

```
    testBarData[ cnt ].percent = EMPTY ;
}

if (argc == 1)
{
    fprintf( stderr, "Error usage: %s <topology file>\n", argv[0] );
    exit( -1 ) ;
}
RegisteredButtons.numberOfButtons = 0 ;

MonitorData.currentMenuSet = MAINMENUSET ;
MonitorData.buttonSelected = -99 ;/* Indicate no button selected */

MonitorData.quit         = FALSE ;
MonitorData.startTraffic = FALSE ;
       MonitorData.tokenNewView = FALSE;/* Only used for testing and debugging */


MenuSet[ MAINMENUSET  ]  = MainMenu ;
MenuSet[ LOADMENUSET  ]  = LoadMenu ;
MenuSet[ CONFIGMENUSET ] = ConfigMenu ;
MenuSet[ WINDOWMENUSET ] = WindowMenu ;

/* THIS IS FOR DEBUGGING AND TESTING THE STANDALONE SYSTEM ONLY! */
StuffTokenData();
StuffProcessData();


InitWindowEnv( argc, argv ) ;

/* Reads in the network topology information used to build graphical network map */
StuffTopology( &Topology, argv[1], &TotalNodes, &TotalRings ) ;
MonitorData.Nodes = TotalNodes ;
MonitorData.Rings = TotalRings ;


/*fprintf( stderr, "Total Nodes = %d   TotalRings = %d \n", TotalNodes, TotalRings ) ;
*/

/*fprintf( stderr, "Before Parse \n" ) ;
if ( argc > 1 )
    ParseCommandLine( argc, argv, &MonitorData ) ;

fprintf( stderr, "After Parse \n" ) ;
*/

/* Main event-reading loop */
done = 0 ;
while ( MonitorData.quit == FALSE )
{
    TestStuff(&MonitorData) ; /* This function only is necessary for showing test data */

    /* DEBUG and TESTING */
    if (ProcessData.status == TRACKING_ON)
       ProcessNext(mainDisplay, mainWindow, mainGC, ProcessTest[ ProcessData.TestIndex ].Address) ;

    /* read the next event */
    XNextEvent ( mainDisplay, &mainEvent ) ;
    switch ( mainEvent.type )
    {
       /* repaint window on expose event */
       case Expose :
          if ( mainEvent.xexpose.count == 0 )
          {
              /* Should Check what window this expose is */
              /* issued for */
              if ( mainEvent.xexpose.window == mainWindow )
              {
                 PaintTopology (
                    mainEvent.xexpose.display,
                    mainEvent.xexpose.window,
                    mainGC,
                    Topology,
                    &MonitorData,
                    &Bridges ) ;
                 PaintWindowBorders(
                    mainEvent.xexpose.display,
                    mainEvent.xexpose.window,
                    mainGC,
                    BUTTONTOP ) ;

                 /* If process tracking was happening when the repaint
```

```
      was called for */
   if (ProcessData.status == TRACKING_ON)
   {
      MarkCU( mainDisplay, mainWindow, mainGC,
          RegisteredNodes.NodeInfo[ProcessData.CU_Index].x,
          RegisteredNodes.NodeInfo[ProcessData.CU_Index].y,
          NODERADIUS );

      MarkToken( mainDisplay, mainWindow, mainGC,
          RegisteredNodes.NodeInfo[ProcessData.NODE_Index].x,
          RegisteredNodes.NodeInfo[ProcessData.NODE_Index].y,
          NODERADIUS );

      if ( ProcessData.Processing == TRUE )
          MarkBusy( mainDisplay, mainWindow, mainGC,
           ProcessData.NODE_Index ) ;
   }

   DisplayMenu( mainEvent.xexpose.display,
          mainEvent.xexpose.window,
          mainGC,
          MenuSet[ MonitorData.currentMenuSet ],
          &MonitorData) ;


   if (MonitorData.startTraffic == TRUE)
   {
      TrafficWindow( mainEvent.xexpose.display,
           trafficWindow,
           trafficGC,
           &MonitorData ) ;

   }
}
else if ( mainEvent.xexpose.window == aboutWindow )
{
   /* Repaint About Window */
   /* Must reset here */

   /* Paints buttons and text */
   PaintAboutBox(mainEvent.xexpose.display,
      mainEvent.xexpose.window,
      aboutGC,&MonitorData ) ;

}
else if ( mainEvent.xexpose.window == tokenWindow )
{
   /* Paints buttons and text */
   PaintTokenWindow(mainEvent.xexpose.display,
      mainEvent.xexpose.window,
      tokenGC ) ;
   AvgTokenPaint(mainEvent.xexpose.display,
      mainEvent.xexpose.window,
      tokenGC, testBarData[ 0 ].percent, !ADD_DATA, &MonitorData );
}
else if ( mainEvent.xexpose.window == legendWindow )
{
   /* Paints buttons and text */
   PaintLegendBox(mainEvent.xexpose.display,
      mainEvent.xexpose.window,
      legendGC, &MonitorData ) ;
}
else if ( mainEvent.xexpose.window == zoomWindow )
{
   /* Paints buttons and text */
   PaintZoomWindow( mainDisplay,
          zoomWindow,
          zoomGC,
          &MonitorData ) ;
}
else if ( mainEvent.xexpose.window == processWindow )
{
   /* Paints buttons and text */
   PaintProcessWindow( mainDisplay,
          processWindow,
          processGC,
          &MonitorData ) ;
}

else if ( mainEvent.xexpose.window == trafficWindow )
{
```

```
            PaintWindowBorders(
               mainEvent.xexpose.display,
               mainEvent.xexpose.window,
               trafficGC,
               BUTTONRIGHT ) ;

            /* Paints buttons and text */
            PaintTrafficWindow(mainEvent.xexpose.display,
               mainEvent.xexpose.window,
               trafficGC,
               &MonitorData ) ;
         }


      }

      break ;

   /* process keyboard mapping changes */
   case MappingNotify :
      XRefreshKeyboardMapping ( &mainEvent ) ;
      break ;

   /* process mouse-button presses */
   case ButtonPress :
      /* If a mainmenu button was depressed this function will automatically
          call the function mapped to that menu choice ONLY after
          that function is run to completion will it return here
      */
      if ( mainEvent.xbutton.window == mainWindow )
      {
          int ButtonPressed = EMPTY, NodeType = EMPTY;
          /* Call CheckButtonPress Only for menus NOT dialog boxes */
          CheckButtonPress( mainEvent.xbutton.display,
                mainEvent.xbutton.window,
                mainGC,
                mainEvent.xbutton.x,
                mainEvent.xbutton.y,
                &MonitorData,
                MenuSet[ MonitorData.currentMenuSet ] ) ;

          NodeType = CheckNodePress( mainEvent.xbutton.display,
                mainEvent.xbutton.window,
                mainGC,
                mainEvent.xbutton.x,
                mainEvent.xbutton.y,
                &MonitorData ) ;

          switch (mainEvent.xbutton.button)
          {
             case LEFTBUTTON:

                break ;
             case CENTERBUTTON:
                switch (NodeType)
                {
                    case RING:
                    break ;
                    /*case BRIDGE:
                    if (MonitorData.bridgeWindow == TRUE)
                    {
                    PaintBridgeWindow( mainDisplay,
                    bridgeWindow,
                    bridgeGC,
                    &MonitorData ) ;
                    }
                    else
                    {
                    BridgeWindow(mainDisplay,
                    bridgeWindow,
                    bridgeGC,
                    &MonitorData);
                    MonitorData.bridgeWindow = TRUE ;
                    }
                    break ;
                    */
                    case NODE:
                    case BRIDGE:
                    if (MonitorData.zoomWindow == TRUE)
                    {
                    /* Windows open just paint new stats */
                    PaintZoomWindow( mainDisplay,
```

```
                zoomWindow,
                zoomGC,
                &MonitorData) ;
                }
                else
                {
                ZoomWindow(mainDisplay,
                zoomWindow,
                zoomGC,
                &MonitorData);
                MonitorData.zoomWindow = TRUE ;
                }
                break ;
            }
            break ;

        case RIGHTBUTTON:
            switch (NodeType)
            {
                case RING:
                /* Only for testing and debugging */
                if (MonitorData.tokenWindow == TRUE)
                {
                /* Windows open just paint new stats */
                AvgTokenPaint(mainDisplay,
                tokenWindow,
                tokenGC,
                35,
                !ADD_DATA,
                &MonitorData ) ;
                }
                else
                {

                TokenWindow(mainDisplay,
                tokenWindow,
                tokenGC,
                &MonitorData);
                MonitorData.tokenWindow = TRUE ;
                }
                break ;
            }
            break ;
    }

}
/* If close button in about window button was pressed */
else if (mainEvent.xbutton.window == aboutWindow )
{
    /* Call DetectButtonPress Only for dialog boxes NOT menus */
    rt_value = DetectButtonPress(
            mainEvent.xbutton.display,
            mainEvent.xbutton.window,
            mainGC,
            mainEvent.xbutton.x,
            mainEvent.xbutton.y,
            &MonitorData
             ) ;
    if ( rt_value != -99 )
    {
       UnRegisterButton( MonitorData.aboutButIndex ) ;
       XUnmapWindow( mainEvent.xbutton.display, aboutWindow ) ;
       MonitorData.aboutWindow = FALSE ;
       MonitorData.aboutButIndex = EMPTY ;
    }

}
/* If close button in traffic window button was pressed */
else if (mainEvent.xbutton.window == trafficWindow )
{
    /* Call DetectButtonPress Only for dialog boxes NOT menus */
    rt_value = DetectButtonPress(
            mainEvent.xbutton.display,
            mainEvent.xbutton.window,
            mainGC,
            mainEvent.xbutton.x,
            mainEvent.xbutton.y,
            &MonitorData
             ) ;
    if ( rt_value != -99 )
    {
       UnRegisterButton( MonitorData.trafficButIndex ) ;
```

```
            XUnmapWindow( mainEvent.xbutton.display, trafficWindow ) ;
            MonitorData.trafficWindow = FALSE ;
            MonitorData.trafficButIndex = EMPTY ;
        }

    }
    /* If tokenWindow close button was pressed */
    else if (mainEvent.xbutton.window == tokenWindow )
    {
        /* Call DetectButtonPress Only for dialog boxes NOT menus */
        rt_value = DetectButtonPress(
                mainDisplay,
                tokenWindow,
                tokenGC,
                mainEvent.xbutton.x,
                mainEvent.xbutton.y,
                &MonitorData
                 ) ;
        if ( rt_value != -99 )
        {
            UnRegisterButton( MonitorData.tokenButIndex ) ;
            /*TokenWindow(mainDisplay,
                tokenWindow,
                tokenGC,
                &MonitorData);*/

            XUnmapWindow( mainEvent.xbutton.display, tokenWindow ) ;

            MonitorData.tokenWindow = FALSE ;
            MonitorData.tokenButIndex = EMPTY ;

        }

    }
    /* If close button in legend window button was pressed */
    else if (mainEvent.xbutton.window == legendWindow )
    {
        /* Call DetectButtonPress Only for dialog boxes NOT menus */
        rt_value = DetectButtonPress(
                mainEvent.xbutton.display,
                mainEvent.xbutton.window,
                mainGC,
                mainEvent.xbutton.x,
                mainEvent.xbutton.y,
                &MonitorData
                 ) ;
        if ( rt_value != -99 )
        {
            UnRegisterButton( MonitorData.legendButIndex ) ;
            XUnmapWindow( mainEvent.xbutton.display, legendWindow ) ;
            MonitorData.legendWindow = FALSE ;
            MonitorData.legendButIndex = EMPTY ;
        }

    }

    /* If processWindow close button was pressed */
    else if (mainEvent.xbutton.window == processWindow )
    {
        /* Call DetectButtonPress Only for dialog boxes NOT menus */
        rt_value = DetectButtonPress(
                mainDisplay,
                processWindow,
                processGC,
                mainEvent.xbutton.x,
                mainEvent.xbutton.y,
                &MonitorData
                 ) ;
        if ( rt_value != -99 )
        {
            UnRegisterButton( MonitorData.processButIndex ) ;
            XUnmapWindow( mainEvent.xbutton.display, processWindow ) ;
            MonitorData.processWindow = FALSE;
            MonitorData.processButIndex = EMPTY ;
            ProcessData.status = TRACKING_OFF ;
            ProcessData.started = FALSE ;
            ProcessData.TestIndex = 0 ;
        }
    }

    /* If zoomWindow close button was pressed */
    else if (mainEvent.xbutton.window == zoomWindow )
```

```
            {
                /* Call DetectButtonPress Only for dialog boxes NOT menus */
                rt_value = DetectButtonPress(
                      mainDisplay,
                      zoomWindow,
                      zoomGC,
                      mainEvent.xbutton.x,
                      mainEvent.xbutton.y,
                      &MonitorData
                       ) ;
                if ( rt_value != -99 )
                {
                   UnRegisterButton( MonitorData.zoomButIndex ) ;
                   XUnmapWindow( mainEvent.xbutton.display, zoomWindow ) ;
                   MonitorData.zoomWindow = FALSE;
                   MonitorData.zoomButIndex = EMPTY ;
                }
            }



            break ;


         /* process keyboard input */
         case KeyPress :
            i = XLookupString( &mainEvent, text, 10, &mainKey, 0 ) ;
            if ( i == 1 && text [ 0 ] == 'q' )
                done = 1 ;
            break ;
      }
   }   /* while */
   /* Termination and Cleanup */
   XFreeGC ( mainDisplay, mainGC ) ;
   XDestroyWindow ( mainDisplay, mainWindow ) ;
   XCloseDisplay ( mainDisplay ) ;
   exit ( 0 ) ;

}


ParseCommandLine( argc, argv, thisDisplay, MonitorData )
int argc ;
char **argv ;
Display *thisDisplay ;
struct Monitor MonitorData ;

{
   int ParamOkay = FALSE ;

   int loop ;
   for ( loop=0; loop < argc; loop++ )
   {
       if (strcmp(argv[loop], "-T") ==0)
       {
          ParamOkay = TRUE ;
          MonitorData.startTraffic = TRUE ;
       }
   }

   if (ParamOkay == FALSE)
   {
       fprintf(stderr, "USAGE: monitor {-T -Ssimulation name} \n" ) ;
       exit( -1 ) ;
   }
}
```

```
/* FILE: INIT.C
   AUTHOR: RICHARD CZOP
   DATE: 12/13/91
   REAL-TV
*/

#include "monitor.h"

void
InitWindowEnv( argc, argv )
int argc ;
char **argv ;
{
   char *display_name = NULL;    /* server to connect to; NULL means */
                                  /* connect to server specified in   */
                                 /* environment variable DISPLAY     */


   /* Initialize Window Environment */


   /* The display_name argument to XopenDisplay specifies which server */
   /* to connect to. When display_name is not specified by the user, it*/
   /* should be set to NULL, which causes XopenDisplay to connect to   */
   /* the server listed in the UNIX environment DISPLAY variable.      */
   if ((mainDisplay=XOpenDisplay(display_name)) == NULL)
   {
         (void)  fprintf( stderr,
                     "ERROR: MONITOR cannot connect to X server %s\n",
                      XDisplayName(display_name));
            exit(-1);
   }


   mainScreen  = DefaultScreen ( mainDisplay ) ;

   /* default pixel values */
   mybackground = WhitePixel ( mainDisplay, mainScreen ) ;
   myforeground = BlackPixel ( mainDisplay, mainScreen ) ;

   /* Before mapping the window(which display it on the screen), */
   /* an application must set the standard properties to tell the*/
   /* window manager a few essential things :                    */
   /*                                                            */
   /*     . Window name                                          */
   /*     . Icon   name                                          */
   /*     . Icon Pixmap                                          */
   /*     . Command name and arguments                           */
   /*     . Number of arguments                                  */
   /*     . preferred window sizes                               */

   /* default program-specified window postion and size */
   mainHint.x            = MAINWINDOWX ;
   mainHint.y            = MAINWINDOWY ;
   mainHint.width        = MAINWINDOWWIDTH ;
   mainHint.height       = MAINWINDOWHEIGHT ;
   mainHint.flags        = PPosition | PSize | PMinSize | PMaxSize;
   mainHint.min_width    = MAINWINDOWMINWIDTH ;
   mainHint.min_height = MAINWINDOWMINHEIGHT ;
   mainHint.max_width  = MAINWINDOWMAXWIDTH ;
   mainHint.max_height = MAINWINDOWMAXHEIGHT ;

   aboutHint.x           = ABOUTWINDOWX ;
   aboutHint.y           = ABOUTWINDOWY ;
   aboutHint.width       = ABOUTWINDOWWIDTH ;
   aboutHint.height      = ABOUTWINDOWHEIGHT ;
   aboutHint.flags       = PPosition | PSize | PMinSize ;
   aboutHint.min_width  = ABOUTWINDOWMINWIDTH ;
   aboutHint.min_height = ABOUTWINDOWMINHEIGHT ;

   trafficHint.x          = TRAFFICWINDOWX ;
   trafficHint.y          = TRAFFICWINDOWY ;
   trafficHint.width      = TRAFFICWINDOWWIDTH ;
   trafficHint.height     = TRAFFICWINDOWHEIGHT ;
   trafficHint.flags      = PPosition | PSize | PMinSize | PMaxSize;
   trafficHint.min_width  = TRAFFICWINDOWMINWIDTH ;
   trafficHint.min_height = TRAFFICWINDOWMINHEIGHT ;
   trafficHint.max_width  = TRAFFICWINDOWMAXWIDTH ;
   trafficHint.max_height = TRAFFICWINDOWMAXHEIGHT ;

   zoomHint.x            = ZOOMWINDOWX ;
   zoomHint.y            = ZOOMWINDOWY ;
```

```
zoomHint.width      = ZOOMWINDOWWIDTH ;
zoomHint.height     = ZOOMWINDOWHEIGHT ;
zoomHint.flags      = PPosition | PSize | PMinSize | PMaxSize;
zoomHint.min_width  = ZOOMWINDOWMINWIDTH ;
zoomHint.min_height = ZOOMWINDOWMINHEIGHT ;
zoomHint.max_width  = ZOOMWINDOWMAXWIDTH ;
zoomHint.max_height = ZOOMWINDOWMAXHEIGHT ;

processHint.x          = PROCESSWINDOWX ;
processHint.y          = PROCESSWINDOWY ;
processHint.width      = PROCESSWINDOWWIDTH ;
processHint.height     = PROCESSWINDOWHEIGHT ;
processHint.flags      = PPosition | PSize | PMinSize | PMaxSize;
processHint.min_width  = PROCESSWINDOWMINWIDTH ;
processHint.min_height = PROCESSWINDOWMINHEIGHT ;
processHint.max_width  = PROCESSWINDOWMAXWIDTH ;
processHint.max_height = PROCESSWINDOWMAXHEIGHT ;

tokenHint.x          = TOKENWINDOWX ;
tokenHint.y          = TOKENWINDOWY ;
tokenHint.width      = TOKENWINDOWWIDTH ;
tokenHint.height     = TOKENWINDOWHEIGHT ;
tokenHint.flags      = PPosition | PSize | PMinSize | PMaxSize;
tokenHint.min_width  = TOKENWINDOWMINWIDTH ;
tokenHint.min_height = TOKENWINDOWMINHEIGHT ;
tokenHint.max_width  = TOKENWINDOWMAXWIDTH ;
tokenHint.max_height = TOKENWINDOWMAXHEIGHT ;

bridgeHint.x          = BRIDGEWINDOWX ;
bridgeHint.y          = BRIDGEWINDOWY ;
bridgeHint.width      = BRIDGEWINDOWWIDTH ;
bridgeHint.height     = BRIDGEWINDOWHEIGHT ;
bridgeHint.flags      = PPosition | PSize | PMinSize | PMaxSize;
bridgeHint.min_width  = BRIDGEWINDOWMINWIDTH ;
bridgeHint.min_height = BRIDGEWINDOWMINHEIGHT ;
bridgeHint.max_width  = BRIDGEWINDOWMAXWIDTH ;
bridgeHint.max_height = BRIDGEWINDOWMAXHEIGHT ;

legendHint.x          = LEGENDWINDOWX ;
legendHint.y          = LEGENDWINDOWY ;
legendHint.width      = LEGENDWINDOWWIDTH ;
legendHint.height     = LEGENDWINDOWHEIGHT ;
legendHint.flags      = PPosition | PSize | PMinSize ;
legendHint.min_width  = LEGENDWINDOWMINWIDTH ;
legendHint.min_height = LEGENDWINDOWMINHEIGHT ;


/* window creation */
mainWindow = XCreateSimpleWindow ( mainDisplay,
                 DefaultRootWindow( mainDisplay ),
                 mainHint.x,
                 mainHint.y,
                 mainHint.width,
                 mainHint.height,
                 10,
                 myforeground,
                 mybackground ) ;
/* window creation */
aboutWindow = XCreateSimpleWindow ( mainDisplay,
                 DefaultRootWindow( mainDisplay ),
                 ABOUTWINDOWX,
                 ABOUTWINDOWY,
                 ABOUTWINDOWWIDTH,
                 ABOUTWINDOWHEIGHT,
                 5,
                 myforeground,
                 mybackground ) ;

/* window creation */
trafficWindow = XCreateSimpleWindow ( mainDisplay,
                 DefaultRootWindow( mainDisplay ),
                 TRAFFICWINDOWX,
                 TRAFFICWINDOWY,
                 TRAFFICWINDOWWIDTH,
                 TRAFFICWINDOWHEIGHT,
                 5,
                 myforeground,
                 mybackground ) ;

/* window creation */
zoomWindow = XCreateSimpleWindow ( mainDisplay,
                 DefaultRootWindow( mainDisplay ),
```

```
                    ZOOMWINDOWX,
                    ZOOMWINDOWY,
                    ZOOMWINDOWWIDTH,
                    ZOOMWINDOWHEIGHT,
                    5,
                    myforeground,
                    mybackground ) ;

    /* window creation */
    processWindow = XCreateSimpleWindow ( mainDisplay,
                    DefaultRootWindow( mainDisplay ),
                    PROCESSWINDOWX,
                    PROCESSWINDOWY,
                    PROCESSWINDOWWIDTH,
                    PROCESSWINDOWHEIGHT,
                    5,
                    myforeground,
                    mybackground ) ;

    /* window creation */
    tokenWindow = XCreateSimpleWindow ( mainDisplay,
                    DefaultRootWindow( mainDisplay ),
                    TOKENWINDOWX,
                    TOKENWINDOWY,
                    TOKENWINDOWWIDTH,
                    TOKENWINDOWHEIGHT,
                    5,
                    myforeground,
                    mybackground ) ;

    /* window creation */
    /*bridgeWindow = XCreateSimpleWindow ( mainDisplay,
                    DefaultRootWindow( mainDisplay ),
                    BRIDGEWINDOWX,
                    BRIDGEWINDOWY,
                    BRIDGEWINDOWWIDTH,
                    BRIDGEWINDOWHEIGHT,
                    5,
                    myforeground,
                    mybackground ) ;
    */

    /* window creation */
    legendWindow = XCreateSimpleWindow ( mainDisplay,
                    DefaultRootWindow( mainDisplay ),
                    LEGENDWINDOWX,
                    LEGENDWINDOWY,
                    LEGENDWINDOWWIDTH,
                    LEGENDWINDOWHEIGHT,
                    5,
                    myforeground,
                    mybackground ) ;

    /* set properties for window manager.                     */
    /* The UNIX shell command name and arguments are passed into   */
    /* MAIN in the standard fashion from the command line, as argv */
    /* and argc. these can be used directly as arguments in the    */
    /* call to set the standard properties. */



    /* Lets window manager and other windows know about us */
    XSetStandardProperties ( mainDisplay, mainWindow, AppName, AppName,
                None, argv, argc, &mainHint ) ;

    XSetStandardProperties ( mainDisplay, aboutWindow, "About Real-TV", AppName,
                None, argv, argc, &aboutHint ) ;

    XSetStandardProperties ( mainDisplay, trafficWindow, "Monitor Traffic", AppName,
                None, argv, argc, &trafficHint ) ;

    XSetStandardProperties ( mainDisplay, zoomWindow, "Node Statistics", AppName,
                None, argv, argc, &zoomHint ) ;

    XSetStandardProperties ( mainDisplay, processWindow, "Process Statistics", AppName,
                None, argv, argc, &processHint ) ;

    XSetStandardProperties ( mainDisplay, tokenWindow, "Ring Statistics", AppName,
                None, argv, argc, &tokenHint ) ;

 /* XSetStandardProperties ( mainDisplay, bridgeWindow, "Bridge Statistics", AppName,
                None, argv, argc, &bridgeHint ) ;
```

```
*/
   XSetStandardProperties ( mainDisplay, legendWindow, "Network Legend", AppName,
              None, argv, argc, &legendHint ) ;

   /* GC creation and initialization */
   mainGC = XCreateGC ( mainDisplay, mainWindow, 0, 0 ) ;
   XSetBackground( mainDisplay, mainGC, mybackground ) ;
   XSetForeground( mainDisplay, mainGC, myforeground ) ;

   aboutGC = XCreateGC ( mainDisplay, aboutWindow, 0, 0 ) ;
   XSetBackground( mainDisplay, aboutGC, mybackground ) ;
   XSetForeground( mainDisplay, aboutGC, myforeground ) ;

   trafficGC = XCreateGC ( mainDisplay, trafficWindow, 0, 0 ) ;
   XSetBackground( mainDisplay, trafficGC, mybackground ) ;
   XSetForeground( mainDisplay, trafficGC, myforeground ) ;

   zoomGC = XCreateGC ( mainDisplay, zoomWindow, 0, 0 ) ;
   XSetBackground( mainDisplay, zoomGC, mybackground ) ;
   XSetForeground( mainDisplay, zoomGC, myforeground ) ;

   processGC = XCreateGC ( mainDisplay, processWindow, 0, 0 ) ;
   XSetBackground( mainDisplay, processGC, mybackground ) ;
   XSetForeground( mainDisplay, processGC, myforeground ) ;

   tokenGC = XCreateGC ( mainDisplay, tokenWindow, 0, 0 ) ;
   XSetBackground( mainDisplay, tokenGC, mybackground ) ;
   XSetForeground( mainDisplay, tokenGC, myforeground ) ;

   /*bridgeGC = XCreateGC ( mainDisplay, bridgeWindow, 0, 0 ) ;
   XSetBackground( mainDisplay, bridgeGC, mybackground ) ;
   XSetForeground( mainDisplay, bridgeGC, myforeground ) ;
   */
   legendGC = XCreateGC ( mainDisplay, legendWindow, 0, 0 ) ;
   XSetBackground( mainDisplay, legendGC, mybackground ) ;
   XSetForeground( mainDisplay, legendGC, myforeground ) ;

   /* input event selection */
   XSelectInput ( mainDisplay, mainWindow, ButtonPressMask | KeyPressMask |
          ExposureMask ) ;

   XSelectInput ( mainDisplay, aboutWindow, ButtonPressMask | KeyPressMask |
          ExposureMask ) ;

   XSelectInput ( mainDisplay, trafficWindow, ButtonPressMask | KeyPressMask |
          ExposureMask ) ;

   XSelectInput ( mainDisplay, zoomWindow, ButtonPressMask | KeyPressMask |
          ExposureMask ) ;

   XSelectInput ( mainDisplay, processWindow, ButtonPressMask | KeyPressMask |
          ExposureMask ) ;

   XSelectInput ( mainDisplay, tokenWindow, ButtonPressMask | KeyPressMask |
          ExposureMask ) ;

   /*XSelectInput ( mainDisplay, bridgeWindow, ButtonPressMask | KeyPressMask |
          ExposureMask ) ;
   */
   XSelectInput ( mainDisplay, legendWindow, ButtonPressMask | KeyPressMask |
          ExposureMask ) ;

   /* window mapping */
   XMapRaised ( mainDisplay, mainWindow ) ;

}

void
PaintWindowBorders( thisDisplay, thisWindow, thisGC, buttonPosition )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int buttonPosition ;
{
   /* Get the current attributes of the window so we can repaint it properly */
   XGetWindowAttributes( thisDisplay, thisWindow, &windowAttributes ) ;

   /* Menu Delimiters */
   /* Top Menu Line   */
   if ( buttonPosition == BUTTONTOP )
   {
      XDrawLine ( thisDisplay,
```

```
        thisWindow,
        thisGC,
        TOPLINEINDENT,
        MENUTOPLINE,
        windowAttributes.width - TOPLINEINDENT,
        MENUTOPLINE ) ;
}
else if ( buttonPosition == BUTTONRIGHT )
{
    XDrawLine ( thisDisplay,
        thisWindow,
        thisGC,
        TOPLINEINDENT,
        MENUTOPLINE,
        windowAttributes.width - TOPLINEINDENT - 70,
        MENUTOPLINE ) ;

}
if ( buttonPosition == BUTTONTOP )
{
    /* Bottom Menu Line */
    XDrawLine ( thisDisplay,
        thisWindow,
        thisGC,
        TOPLINEINDENT,
        MENUBOTTOMLINE,
        windowAttributes.width - TOPLINEINDENT,
        MENUBOTTOMLINE ) ;
}

if ( buttonPosition == BUTTONLEFT)
{
    /* Left Side Line */
    XDrawLine ( thisDisplay,
        thisWindow,
        thisGC,
        TOPLINEINDENT + 70,
        MENUTOPLINE,
        TOPLINEINDENT + 70,
        MENUTOPLINE+((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) ;
}
else if ( buttonPosition == BUTTONRIGHT )
{
    /* Left Side Line */
    XDrawLine ( thisDisplay,
        thisWindow,
        thisGC,
        TOPLINEINDENT,
        MENUTOPLINE,
        TOPLINEINDENT,
        MENUBOTTOMLINE+
        ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT) ) ;

}
else if ( buttonPosition == BUTTONTOP )
{
    /* Left Side Line */
    XDrawLine ( thisDisplay,
        thisWindow,
        thisGC,
        TOPLINEINDENT,
        MENUBOTTOMLINE,
        TOPLINEINDENT,
        MENUBOTTOMLINE+((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) ;

}

if ( buttonPosition == BUTTONRIGHT )
{
    /* Right Side Line */
    XDrawLine ( thisDisplay,
        thisWindow,
        thisGC,
        windowAttributes.width - TOPLINEINDENT - 70,
        MENUTOPLINE,
        windowAttributes.width - TOPLINEINDENT - 70,
        MENUBOTTOMLINE+
        ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT) ) ;
        /*MENUTOPLINE+(windowAttributes.height-TOPLINEINDENT)) ; */
}
else
{
```

```
      /* Right Side Line */
      XDrawLine ( thisDisplay,
         thisWindow,
         thisGC,
         windowAttributes.width - TOPLINEINDENT,
         MENUBOTTOMLINE,
         windowAttributes.width - TOPLINEINDENT,
         MENUBOTTOMLINE+((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) ;

   }

   if ( buttonPosition == BUTTONBOTTOM )
   {
      /* Bottom Screeen Line */
      XDrawLine ( thisDisplay,
         thisWindow,
         thisGC,
         TOPLINEINDENT,
         MENUBOTTOMLINE+
         ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT) - 70,
         windowAttributes.width - TOPLINEINDENT,
         MENUBOTTOMLINE+
         ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT) - 70) ;
   }
   else if ( buttonPosition == BUTTONRIGHT )
   {
      /* Bottom Screeen Line */
      XDrawLine ( thisDisplay,
         thisWindow,
         thisGC,
         TOPLINEINDENT,
         MENUBOTTOMLINE+
         ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT),
         windowAttributes.width - TOPLINEINDENT - 70,
         MENUBOTTOMLINE+  ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) ;

   }
   else if ( buttonPosition == BUTTONTOP )
   {
      /* Bottom Screeen Line */
      XDrawLine ( thisDisplay,
         thisWindow,
         thisGC,
         TOPLINEINDENT,
         MENUBOTTOMLINE+
         ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT),
         windowAttributes.width - TOPLINEINDENT,
         MENUBOTTOMLINE+  ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) ;

   }

}
```

```
/* FILE: MISC.C
   AUTHOR: RICHARD CZOP
   DATE: 12/13/91
   REAL-TV
*/

#include "monitor.h"

Pause( length )
long length ;
{
   long count ;
   for( count=0; count < length; count++ )
       ;
}

UnRegisterButton( buttonIndex )
int buttonIndex ;
{
   extern struct RegButtons RegisteredButtons ;

   if ( buttonIndex >= 0 )
   {
       /* Unregisters the button index passed in */
       RegisteredButtons.ButtonPtr[buttonIndex].X = EMPTY ;
       RegisteredButtons.ButtonPtr[buttonIndex].Y = EMPTY ;
       RegisteredButtons.ButtonPtr[buttonIndex].string = NULL ;
       RegisteredButtons.ButtonPtr[buttonIndex].Func = NULL ;

   }

}

int
RegisterButton( thisDisplay, thisWindow, thisGC, buttonX, buttonY, Width, Height, String, Function )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int buttonX ;
int buttonY ;
int Width ;
int Height ;
char *String ;
void (*Function)() ;
{
   int cnt = 0 ;

   /* NOTE: Currently I am assuming that only one Button is registered per dialog box */

   extern struct RegButtons RegisteredButtons ;

   /*if ( ( RegisteredButtons.numberOfButtons + 1 ) > MAXNUMBEROFBUTTONS )
   {
       fprintf( stderr, "ERROR: function RegisterButton button overflow \n" ) ;
       exit( -1 ) ;
   }
   */

   /* Find the next available button place */
   for ( cnt = 0 ; cnt < MAXNUMBEROFBUTTONS; cnt++ )
       if (RegisteredButtons.ButtonPtr[cnt].X == EMPTY )
          break ;

   RegisteredButtons.ButtonPtr[cnt].X = buttonX ;
   RegisteredButtons.ButtonPtr[cnt].Y = buttonY ;
   RegisteredButtons.ButtonPtr[cnt].string = String ;
   RegisteredButtons.ButtonPtr[cnt].Func = Function ;

   /*RegisteredButtons.numberOfButtons++ ;*/
   fprintf(stderr, "Button registered index = %d \n", cnt ) ;

   return cnt ;  /* Return the index of the registered button */
}


/* Draws buttons for menus boxes NOT dialog boxes */

DrawMenuButton( thisDisplay, thisWindow, thisGC, buttonX, buttonY, Width, Height, String )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int buttonX ;
```

```
int buttonY ;
int Width ;
int Height ;
char *String ;
{
   /* Draw rectangular Button */
   XDrawRectangle( thisDisplay,
          thisWindow,
          thisGC,
          buttonX,     /* Upper x Left of rectangle */
          buttonY,     /* Upper y Left of rectangle */
          Width,           /* Width of button          */
          Height           /* Height of button         */
          ) ;

   /* Draw Bottom Shadow */
   XDrawLine ( thisDisplay,
          thisWindow,
          thisGC,
          buttonX+3,
          buttonY+Height+1,
          buttonX+Width+1,
          buttonY+Height+1 ) ;

   /* Draw Right Side Shadow */
   XDrawLine ( thisDisplay,
          thisWindow,
          thisGC,
          buttonX+Width+1,
          buttonY+3,
          buttonX+Width+1,
          buttonY+Height+1 ) ;

   XDrawImageString (
          thisDisplay,
          thisWindow,
          thisGC,
          buttonX + 10,
          buttonY + 20,
          String,
          strlen( String ) ) ;
}



/* Draws buttons for dialog boxes NOT menus */
int
DrawButton( thisDisplay, thisWindow, thisGC, buttonX, buttonY, Width, Height, String, Function, registerButton )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int buttonX ;
int buttonY ;
int Width ;
int Height ;
char *String ;
void (*Function)() ;
int registerButton ;
{
   /* Draw rectangular Button */
   XDrawRectangle( thisDisplay,
          thisWindow,
          thisGC,
          buttonX,     /* Upper x Left of rectangle */
          buttonY,     /* Upper y Left of rectangle */
          Width,           /* Width of button          */
          Height           /* Height of button         */
          ) ;

   /* Draw Bottom Shadow */
   XDrawLine ( thisDisplay,
          thisWindow,
          thisGC,
          buttonX+3,
          buttonY+Height+1,
          buttonX+Width+1,
          buttonY+Height+1 ) ;

   /* Draw Right Side Shadow */
   XDrawLine ( thisDisplay,
          thisWindow,
```

```
          thisGC,
          buttonX+Width+1,
          buttonY+3,
          buttonX+Width+1,
          buttonY+Height+1 ) ;

   XDrawImageString (
          thisDisplay,
          thisWindow,
          thisGC,
          buttonX + 10,
          buttonY + 20,
          String,
          strlen( String ) ) ;

   if (registerButton == TRUE)
       return RegisterButton( thisDisplay,
          thisWindow,
          thisGC,
          buttonX,
          buttonY,
          Width,
          Height,
          String,
          Function) ;
   else
       return EMPTY ;


}

/* Detects button presses for dialog boxes NOT menus */
int
DetectButtonPress( thisDisplay, thisWindow, thisGC, thisX, thisY, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int thisX ;
int thisY ;
struct Monitor MonitorData ;
{
   extern struct RegButtons RegisteredButtons ;

   int loopCnt = 0 ;
   int buttonPressed ;

   buttonPressed = -99 ;   /* Dummy Flag */

   for ( loopCnt = 0 ; loopCnt < MAXNUMBEROFBUTTONS ; loopCnt++ )
   {
       if (RegisteredButtons.ButtonPtr[ loopCnt ].Y == EMPTY)
          continue ;

       if ( (thisY >= RegisteredButtons.ButtonPtr[loopCnt].Y) &&
          (thisY <= (RegisteredButtons.ButtonPtr[loopCnt].Y+BUTTONHEIGHT) ) )
       {
          if ( (thisX >= RegisteredButtons.ButtonPtr[loopCnt].X )
             && (thisX <= (RegisteredButtons.ButtonPtr[loopCnt].X+BUTTONWIDTH+1)) )
          {

             /*RegisteredButtons.numberOfButtons = 0 ;*//* Reset buttons */
             UnRegisterButton( loopCnt ) ;
             buttonPressed = loopCnt ;

             /*ShowButtonPress( thisDisplay,
                    thisWindow,
                    thisGC,
                    RegisteredButtons.ButtonPtr[loopCnt].X ) ;
             */
             break ;
          }
       }
   }

   if ( (buttonPressed != -99) &&
       (RegisteredButtons.ButtonPtr[buttonPressed].Func != NULL) )
       /* Button was pressed */
       (*RegisteredButtons.ButtonPtr[buttonPressed].Func) (thisDisplay,
             thisWindow,
             thisGC,
             MonitorData
              ) ;
```

```
   return buttonPressed ;
}

void
Windows( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   MonitorData.currentMenuSet = WINDOWMENUSET ;
   MonitorData.buttonSelected = -99 ;/* Indicate no button selected */

   DisplayMenu( thisDisplay,
          thisWindow,
          thisGC,
          MenuSet[ MonitorData.currentMenuSet ],
          &MonitorData) ;

}


PaintTrafficWindow( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   int buttonIndex, registerButton = FALSE ;

   PaintWindowBorders( thisDisplay, thisWindow, thisGC, BUTTONRIGHT ) ;

   if (MonitorData.trafficButIndex == EMPTY)
       registerButton = TRUE ;

   buttonIndex = DrawButton( thisDisplay,
          thisWindow,
          thisGC,
          TRAFFICWINDOWWIDTH-70,
          (TRAFFICWINDOWHEIGHT-40-(BUTTONHEIGHT+15)),
          BUTTONWIDTH,
          BUTTONHEIGHT,
          "CLOSE",
          NULL,
          registerButton ) ;
   if (registerButton == TRUE)
       MonitorData.trafficButIndex = buttonIndex ;

}

char *
NodeTypeIs( nodeType )
int nodeType ;
{
   char nodeTypeIs[ NODETYPELEN ];

   switch ( nodeType )
   {
       case CU:
          strcpy( nodeTypeIs, "CU                  " );
          break ;
       case ALU:
          strcpy( nodeTypeIs, "ALU                 " );
          break ;
       case REGISTER:
          strcpy( nodeTypeIs, "REGISTER            " );
          break ;
       case PSW:
          strcpy( nodeTypeIs, "PSW                 " );
          break ;
       case IOPROCESSOR:
          strcpy( nodeTypeIs, "IO PROCESSOR        " );
          break ;
       case MAINMEMORY:
          strcpy( nodeTypeIs, "MAIN MEMORY         " );
          break ;
       case SECONDARYMEMORY:
          strcpy( nodeTypeIs, "SECONDARY MEMORY    " );
          break ;
       case COMMANDPROCESSOR:
          strcpy( nodeTypeIs, "COMMAND PROCESSOR   " );
          break ;
```

```
        case OSKERNEL:
            strcpy( nodeTypeIs, "OS KERNEL           " );
            break ;
        default:
            strcpy( nodeTypeIs, "BRIDGE              " );
            break ;

    }

    return nodeTypeIs;
}

void
PaintZoomWindow( thisDisplay, thisWindow, thisGC, MonitorData)
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
    char nodeType[ NODETYPELEN ] ;
    char dispString[ 60 ] ;
    int X = 30, label=0, cnt, buttonIndex, registerButton = FALSE;

    int percent = 50 ;
    buttonIndex = MonitorData.zoomButIndex ;

    PaintWindowBorders( thisDisplay, thisWindow, thisGC, BUTTONTOP ) ;

    sprintf( dispString, "Node Label:       %d", MonitorData.typeSelected ) ;
    XDrawImageString (
            thisDisplay,
            thisWindow,
            thisGC,
            X,
            70,
            dispString,
            strlen( dispString ) ) ;

    strcpy( nodeType, NodeTypeIs( MonitorData.typeSelected ) );
    sprintf( dispString, "Node Type:        %s", nodeType ) ;
    XDrawImageString (
            thisDisplay,
            thisWindow,
            thisGC,
            X,
            85,
            dispString,
            strlen( dispString ) ) ;

        sprintf( dispString, "Node Address:    %s", MonitorData.addSelected );
        XDrawImageString (
            thisDisplay,
            thisWindow,
            thisGC,
            X,
            100,
            dispString,
            strlen( dispString ) ) ;

        sprintf( dispString, "Member of ring:  %d", MonitorData.ringSelected );
        XDrawImageString (
            thisDisplay,
            thisWindow,
            thisGC,
            X,
            115,
            dispString,
            strlen( dispString ) ) ;

        sprintf( dispString, "Utilization :" );
        XDrawImageString (
            thisDisplay,
            thisWindow,
            thisGC,
            X,
            150,
            dispString,
            strlen( dispString ) ) ;

        X = 135 ;

        /* Draw X axis line */
```

```
XDrawLine ( thisDisplay, thisWindow, thisGC, X, 155, X+100, 155 );

/* Draw X axis delimiters */
for ( cnt = 0; cnt < 6; cnt ++ )
{
   XDrawLine ( thisDisplay, thisWindow, thisGC, X, 152, X, 158 );
   sprintf( dispString, "%d", label ) ;
   XDrawImageString ( thisDisplay,thisWindow,thisGC,X,170, dispString, strlen( dispString ) ) ;
   label = label + 20 ;
   X = X + 20 ;
}
X = 135 ;

XFillRectangle ( thisDisplay, thisWindow, thisGC, X, 145, percent, 5) ;

X = 30 ;

sprintf( dispString, "Max. idle time:  %f", 15.54 );
XDrawImageString (
   thisDisplay,
   thisWindow,
   thisGC,
   X,
   190,
   dispString,
   strlen( dispString ) ) ;


sprintf( dispString, "Avg. idle time:  %f", 8.32 );
XDrawImageString (
   thisDisplay,
   thisWindow,
   thisGC,
   X,
   205,
   dispString,
   strlen( dispString ) ) ;

sprintf( dispString, "Max busy time:   %f", 2.34 );
XDrawImageString (
   thisDisplay,
   thisWindow,
   thisGC,
   X,
   220,
   dispString,
   strlen( dispString ) ) ;

sprintf( dispString, "Avg. busy time:  %f", 2.01 );
XDrawImageString (
   thisDisplay,
   thisWindow,
   thisGC,
   X,
   235,
   dispString,
   strlen( dispString ) ) ;

sprintf( dispString, "Current Activity: %s", "IDLE" );
XDrawImageString (
   thisDisplay,
   thisWindow,
   thisGC,
   X,
   265,
   dispString,
   strlen( dispString ) ) ;

if (MonitorData.zoomButIndex == EMPTY)
   registerButton = TRUE ;

buttonIndex = DrawButton( thisDisplay,
   thisWindow,
   thisGC,
   20,
   10,
   BUTTONWIDTH,
   BUTTONHEIGHT,
   "CLOSE",
   NULL,
   registerButton ) ;
```

```
      if (registerButton == TRUE)
         MonitorData.zoomButIndex = buttonIndex ;/* Index of the button registered */


}

void
ZoomWindow( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   extern Window zoomWindow ;

   if (MonitorData.zoomWindow == TRUE )
   {
      MonitorData.zoomWindow = FALSE ;
      MonitorData.zoomButIndex = EMPTY ;/* Indicates that no button is registered for this window */
      XUnmapWindow( thisDisplay, zoomWindow ) ;
   }
   else
   {
      MonitorData.zoomButIndex = EMPTY ;/* Indicates that no button is registered for this window */

      /* window mapping */
      XMapRaised ( thisDisplay, zoomWindow ) ;

      /* Delay Painting of the button and text */
      while( XCheckTypedEvent( thisDisplay, Expose, &mainEvent ) ) ;
         sleep( 1 ) ;

      PaintZoomWindow( thisDisplay, zoomWindow, zoomGC, MonitorData ) ;
      MonitorData.zoomWindow = TRUE ;
   }
}


int
PaintTokenWindow( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   void TokenWindow();

   double index ;
   int cnt, xOffset = 8, yOffset, dispHeight, dispWidth, y, percent = 100, buttonIndex, registerButton = FALSE;
   char strPercent[ 5 ];

   /* Get the current attributes of the window so we can repaint it properly */
   XGetWindowAttributes( thisDisplay, thisWindow, &windowAttributes ) ;


   dispHeight = ((MENUBOTTOMLINE+((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) - MENUBOTTOMLINE ) ;
   dispWidth  = ((windowAttributes.width - TOPLINEINDENT) - TOPLINEINDENT ) ;

   yOffset = (dispHeight / 5) ;

   XClearArea( thisDisplay,
         thisWindow,
         0,
         0,
         windowAttributes.width,
         windowAttributes.height,
         FALSE ) ;


   /* Right Side Line */
   XDrawLine ( thisDisplay,
         thisWindow,
         thisGC,
         windowAttributes.width - TOPLINEINDENT - 30,
         MENUBOTTOMLINE,
         windowAttributes.width - TOPLINEINDENT - 30,
         MENUBOTTOMLINE+((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) ;

   /* Top Line */
   /*XDrawLine ( thisDisplay,
         thisWindow,
         thisGC,
```

```
        TOPLINEINDENT,
        MENUBOTTOMLINE,
        windowAttributes.width - TOPLINEINDENT - 30,
        MENUBOTTOMLINE ) ;
   */
   /* Bottom Screeen Line */
   XDrawLine ( thisDisplay,
        thisWindow,
        thisGC,
        TOPLINEINDENT,
        MENUBOTTOMLINE+
        ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT),
        windowAttributes.width - TOPLINEINDENT - 30,
        MENUBOTTOMLINE+  ((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) ;
   /* Left Side Line */
   XDrawLine ( thisDisplay,
        thisWindow,
        thisGC,
        TOPLINEINDENT,
        MENUBOTTOMLINE,
        TOPLINEINDENT,
        MENUBOTTOMLINE+((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) ;


   /* Right Side Line graph delimiters*/
   for ( cnt = 0; cnt < 6; cnt++ )
   {
      XDrawLine ( thisDisplay,
        thisWindow,
        thisGC,
        windowAttributes.width - TOPLINEINDENT - 30,
        MENUBOTTOMLINE+y,
        windowAttributes.width - TOPLINEINDENT - 30 + 5,
        MENUBOTTOMLINE+ y ) ;

      sprintf( strPercent, "%d", percent ) ;
      percent-=20 ;
      XDrawImageString (
        thisDisplay,
        thisWindow,
        thisGC,
        windowAttributes.width - TOPLINEINDENT + xOffset - 30 + 5,
        MENUBOTTOMLINE+ y,
        strPercent,
        strlen( strPercent ) ) ;
      y += yOffset ;
   }

   if ( MonitorData.tokenButIndex == EMPTY)
      registerButton = TRUE ;

   buttonIndex = DrawButton( thisDisplay,
        thisWindow,
        thisGC,
        TOKENWINDOWWIDTH-80,
        4,
        BUTTONWIDTH,
        BUTTONHEIGHT,
        "CLOSE",
        NULL,
        registerButton) ;

   if (registerButton == TRUE)
      MonitorData.tokenButIndex = buttonIndex ;
}



void
TokenWindow( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   extern Window tokenWindow ;
   int cnt;

   if (MonitorData.tokenWindow == TRUE )
   {
      MonitorData.tokenWindow = FALSE ;
      MonitorData.tokenButIndex = EMPTY ;/* Indicates that no button is registered */
```

```
        XUnmapWindow( mainDisplay, tokenWindow ) ;

        /* initialize the bar data to empty */
        /*
        for ( cnt = 0 ; cnt < MAXBARS; cnt++ )
        {
           tokenBarData[ cnt ].percent = EMPTY ;
           testBarData[ cnt ].percent = EMPTY ;
        }
        */
    }
    else
    {
        MonitorData.tokenButIndex = EMPTY ;/* Indicates that no button is registered */

        /* window mapping */
        XMapRaised ( mainDisplay, tokenWindow ) ;

        /* Delay Painting of the button and text */
        while( XCheckTypedEvent( mainDisplay, Expose, &mainEvent ) ) ;
           sleep( 1 ) ;

        /* Paints buttons and text */
        PaintTokenWindow( mainDisplay, tokenWindow, tokenGC, MonitorData ) ;

        MonitorData.tokenWindow = TRUE  ;
    }
}



void
BridgeWindow( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
    extern Window tokenWindow ;
    int cnt;


    if (MonitorData.bridgeWindow == TRUE )
    {
        MonitorData.bridgeWindow = FALSE ;
        MonitorData.bridgeButIndex = EMPTY ;/* Indicates that no button is registered */

        XUnmapWindow( mainDisplay, bridgeWindow ) ;
    }
    else
    {
        /* window mapping */
        XMapRaised ( mainDisplay, bridgeWindow ) ;

        /* Delay Painting of the button and text */
        while( XCheckTypedEvent( mainDisplay, Expose, &mainEvent ) ) ;
           sleep( 1 ) ;

        /* Paints buttons and text */
        PaintBridgeWindow( mainDisplay, bridgeWindow, bridgeGC, MonitorData ) ;

        MonitorData.bridgeWindow = TRUE  ;
    }
}


PaintBridgeWindow( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
    char nodeType[ NODETYPELEN ] ;
    char dispString[ 60 ] ;
    int X = 30, label=0, cnt, buttonIndex, registerButton = FALSE;

    int percent = 50 ;

    PaintWindowBorders( thisDisplay, thisWindow, thisGC, BUTTONTOP ) ;
```

```
sprintf( dispString, "Node Label:      %d", MonitorData.typeSelected ) ;
XDrawImageString (
       thisDisplay,
       thisWindow,
       thisGC,
       X,
       70,
       dispString,
       strlen( dispString ) ) ;
strcpy( nodeType, NodeTypeIs( MonitorData.typeSelected ) );
sprintf( dispString, "Node Type:      %s", nodeType ) ;
XDrawImageString (
       thisDisplay,
       thisWindow,
       thisGC,
       X,
       85,
       dispString,
       strlen( dispString ) ) ;

    sprintf( dispString, "Node Address:   %s", MonitorData.addSelected );
    XDrawImageString (
       thisDisplay,
       thisWindow,
       thisGC,
       X,
       100,
       dispString,
       strlen( dispString ) ) ;

    sprintf( dispString, "Member of ring:  %d", MonitorData.ringSelected );
    XDrawImageString (
       thisDisplay,
       thisWindow,
       thisGC,
       X,
       115,
       dispString,
       strlen( dispString ) ) ;

    sprintf( dispString, "Utilization :" );
    XDrawImageString (
       thisDisplay,
       thisWindow,
       thisGC,
       X,
       150,
       dispString,
       strlen( dispString ) ) ;

    X = 135 ;

    /* Draw X axis line */
    XDrawLine ( thisDisplay, thisWindow, thisGC, X, 155, X+100, 155 );

    /* Draw X axis delimiters */
    for ( cnt = 0; cnt < 6; cnt ++ )
    {
       XDrawLine ( thisDisplay, thisWindow, thisGC, X, 152, X, 158 );
       sprintf( dispString, "%d", label ) ;
       XDrawImageString ( thisDisplay,thisWindow,thisGC,X,170, dispString, strlen( dispString ) ) ;
       label = label + 20 ;
       X = X + 20 ;
    }
    X = 135 ;

    XFillRectangle ( thisDisplay, thisWindow, thisGC, X, 145, percent, 5) ;

    X = 30 ;

    sprintf( dispString, "Max. idle time:  %f", 15.54 );
    XDrawImageString (
       thisDisplay,
       thisWindow,
       thisGC,
       X,
       190,
       dispString,
       strlen( dispString ) ) ;


    sprintf( dispString, "Avg. idle time:  %f", 8.32 );
```

```
        XDrawImageString (
           thisDisplay,
           thisWindow,
           thisGC,
           X,
           205,
           dispString,
           strlen( dispString ) ) ;

        sprintf( dispString, "Max busy time:   %f", 2.34 );
        XDrawImageString (
           thisDisplay,
           thisWindow,
           thisGC,
           X,
           220,
           dispString,
           strlen( dispString ) ) ;

        sprintf( dispString, "Avg. busy time:  %f", 2.01 );
        XDrawImageString (
           thisDisplay,
           thisWindow,
           thisGC,
           X,
           235,
           dispString,
           strlen( dispString ) ) ;

        sprintf( dispString, "Current Activity: %s", "IDLE" );
        XDrawImageString (
           thisDisplay,
           thisWindow,
           thisGC,
           X,
           265,
           dispString,
           strlen( dispString ) ) ;

   if (MonitorData.bridgeButIndex == EMPTY)
        registerButton = TRUE ;

   buttonIndex = DrawButton( thisDisplay,
          thisWindow,
          thisGC,
          20,
          10,
          BUTTONWIDTH,
          BUTTONHEIGHT,
          "CLOSE",
          NULL,
          registerButton ) ;

   if (registerButton == TRUE)
        MonitorData.bridgeButIndex = buttonIndex ;/* Index of the button registered */

}

void
TrafficWindow( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   extern Window trafficWindow ;

   if (MonitorData.trafficWindow == TRUE )
   {
        MonitorData.trafficWindow = FALSE ;
        MonitorData.trafficButIndex = EMPTY ;/* Indicates that no button is registered */

        XUnmapWindow( thisDisplay, trafficWindow ) ;
   }
   else
   {
        MonitorData.trafficButIndex = EMPTY ;

        /* window mapping */
        XMapRaised ( thisDisplay, trafficWindow ) ;

        /* Delay Painting of the button and text */
```

```
      while( XCheckTypedEvent( thisDisplay, Expose, &mainEvent ) ) ;
         sleep( 1 ) ;

      /* Paints buttons and text */
      PaintTrafficWindow( thisDisplay, trafficWindow, trafficGC, MonitorData ) ;
      MonitorData.trafficWindow = TRUE ;
   }
}

void
Legend( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   extern Window legendWindow ;

   if (MonitorData.legendWindow == TRUE )
   {
      MonitorData.legendWindow = FALSE ;
      MonitorData.legendButIndex = EMPTY ;

      XUnmapWindow( thisDisplay, legendWindow ) ;
   }
   else
   {

      /* window mapping */
      XMapRaised ( thisDisplay, legendWindow ) ;

      /* Delay Painting of the button and text */
      while( XCheckTypedEvent( thisDisplay, Expose, &mainEvent ) ) ;
         sleep( 1 ) ;

      /* Paints buttons and text */
      PaintLegendBox( thisDisplay, legendWindow, legendGC, MonitorData ) ;
      MonitorData.legendWindow = TRUE ;
   }
}


PaintLegendBox( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   char description[21], id[4], dispString[ 26 ] ;
   int buttonIndex, registerButton = FALSE ;

   int X = 5 ;
   int Y = 60 ;

   FILE *fileptr ;          /* ptr to file                             */

   /* Attempt to open the file */
   if ( (fileptr=fopen( "legend.txt", "r" ) ) == NULL )
   {
      printf( "Can't open file: legend.txt\n" );
      exit( -1 );
   }

   XClearWindow( thisDisplay, legendWindow );

   while (fscanf( fileptr, "%20s %3s", description, id ) != EOF)
   {
      sprintf( dispString, "%-20s   %3s", description, id ) ;
      XDrawImageString (
         thisDisplay,
         thisWindow,
         thisGC,
         X,
         Y,
         dispString,
         strlen ( dispString ) ) ;
      Y = Y + 20 ;
   }

   fclose( fileptr ) ;
```

```
    if (MonitorData.legendButIndex == EMPTY)
        registerButton = TRUE ;

    buttonIndex = DrawButton( thisDisplay,
           thisWindow,
           thisGC,
           LEGENDWINDOWWIDTH-80,
           4,
           BUTTONWIDTH,
           BUTTONHEIGHT,
           "CLOSE",
           NULL,
           registerButton) ;

    if (registerButton == TRUE)
        MonitorData.legendButIndex = buttonIndex ;

}



void
AboutBox( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{

    extern Window aboutWindow ;

    if (MonitorData.aboutWindow == TRUE )
    {
        MonitorData.aboutWindow = FALSE ;
        MonitorData.aboutButIndex = EMPTY ;

        XUnmapWindow( thisDisplay, aboutWindow ) ;
    }
    else
    {

        /* window mapping */
        XMapRaised ( thisDisplay, aboutWindow ) ;

        /* Delay Painting of the button and text */
        while( XCheckTypedEvent( thisDisplay, Expose, &mainEvent ) ) ;
           sleep( 1 ) ;

        /* Paints buttons and text */
        PaintAboutBox( thisDisplay, aboutWindow, aboutGC, MonitorData ) ;
        MonitorData.aboutWindow = TRUE ;
    }

}


PaintAboutBox( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
    int buttonIndex, registerButton = FALSE ;
    int X = 55 ;
    int Y = 45 ;

    char *line1="              Real-TV" ;
    char *line2="       Real-Time Visualizer" ;
    char *line3="Monitor Real-Time System Simulation" ;
    char *line4="      Author: Richard Czop" ;
    char *line5="               1991" ;


    XDrawImageString (
           thisDisplay,
           aboutWindow,
           aboutGC,
           X,
           Y,
           line1,
           strlen ( line1 ) ) ;
    XDrawImageString (
```

```
          thisDisplay,
          aboutWindow,
          aboutGC,
          X,
          Y+(1*20),
          line2,
          strlen ( line2 ) ) ;
   XDrawImageString (
          thisDisplay,
          aboutWindow,
          aboutGC,
          X,
          Y+(2*20),
          line3,
          strlen ( line3 ) ) ;

   XDrawImageString (
          thisDisplay,
          aboutWindow,
          aboutGC,
          X,
          Y+(3*20),
          line4,
          strlen ( line4 ) ) ;

   XDrawImageString (
          thisDisplay,
          aboutWindow,
          aboutGC,
          X,
          Y+(4*20),
          line5,
          strlen ( line5 ) ) ;

   if (MonitorData.aboutButIndex == EMPTY)
       registerButton = TRUE ;
   else
       registerButton = FALSE ;

   fprintf(stderr,"About Box() registerButton = %d \n",registerButton ) ;

   buttonIndex = DrawButton( thisDisplay,
          aboutWindow,
          aboutGC,
          (ABOUTWINDOWWIDTH/2)- (BUTTONWIDTH/2),
          ABOUTWINDOWHEIGHT-75,
          BUTTONWIDTH,
          BUTTONHEIGHT,
          "Okay",
          NULL,
          registerButton ) ;

   if (registerButton == TRUE)
       MonitorData.aboutButIndex = buttonIndex ;
}


void
Load( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   MonitorData.currentMenuSet = LOADMENUSET ;
   MonitorData.buttonSelected = -99 ;/* Indicate no button selected */

   DisplayMenu( thisDisplay,
          thisWindow,
          thisGC,
          MenuSet[ MonitorData.currentMenuSet ],
          &MonitorData) ;


}

void
LoadMainMenu( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
```

```
{
   MonitorData.currentMenuSet = MAINMENUSET ;
   MonitorData.buttonSelected = -99 ;/* Indicate no button selected */

   DisplayMenu( thisDisplay,
          thisWindow,
          thisGC,
          MenuSet[ MonitorData.currentMenuSet ],
          &MonitorData) ;
}

void
Connect( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{


}

void
Stats( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{

   MonitorData.currentMenuSet = STATSMENUSET ;
   MonitorData.buttonSelected = -99 ;/* Indicate no button selected */

   DisplayMenu( thisDisplay,
          thisWindow,
          thisGC,
          MenuSet[ MonitorData.currentMenuSet ],
          &MonitorData) ;

}

void
Keyboard( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   /* fork() and exec() the keyboard interface */
   static char *nargv[] = {  (char *) 0};
   if (fork() > 0)
   {
       execvp( "ctty", nargv ) ;
          perror("ctty error\n");
          exit(1);
       }
}

void
RecordSession( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{


}

void
LoadReplay( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{

}


void
CleanUp( thisDisplay, thisWindow, thisGC, MonitorData )
```

```
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   MonitorData.quit = TRUE ;
}


void
Configure1 ( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{

}


void
Configure2 ( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{

}

void
Configure3 ( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{

}
```

```
/* FILE: STATS.C
   AUTHOR: RICHARD CZOP
   DATE: 12/13/91
   REAL-TV
*/

#include "monitor.h"


/* DEBUGGING AND TESTING OF STANDALONE SYSTEM ONLY */
StuffTokenData()
{
   char str_percent[ 3+1 ] ;
   int percent, index = 0 ;

   FILE *fileptr ;         /* ptr to file                          */

   /* Attempt to open the file */
   if ( (fileptr=fopen( "tokendata.test", "r" ) ) == NULL )
   {
       printf( "Can't open file: tokendata.test\n" );
       exit( -1 );
   }

   while (fscanf( fileptr, "%3s", str_percent ) != EOF)
       testBarData[ index++ ].percent = atoi( str_percent ) ;

   fclose( fileptr ) ;
}

/* DEBUGGING AND TESTING OF STANDALONE SYSTEM ONLY */
TestStuff(MonitorData)
struct Monitor MonitorData ;
{
   static int delay = 0;


   /* This prevents the test data to go to fast */
   if (delay == 0)
   {
       if (MonitorData.tokenWindow == TRUE)
          ShowNextTokenStat(MonitorData) ;
   }

   if (delay == 100)
       delay = 0 ;

}

/* DEBUGGING AND TESTING OF STANDALONE SYSTEM ONLY */
ShowNextTokenStat(MonitorData)
struct Monitor MonitorData ;
{
   static int index=1;
   AvgTokenPaint(mainDisplay, tokenWindow, tokenGC, testBarData[ index++ ].percent, ADD_DATA, MonitorData );

   if (index > 20)
       index = 0 ;

}

DrawBar(thisDisplay, thisWindow, thisGC, X, Y, maxHeight, maxWidth, percent )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int X ;
int Y ;
int maxHeight ;
int maxWidth ;
int percent ;
{
   int height, width ;

   width = maxWidth ;
   height = ((maxHeight * percent) / 100 );

   XClearArea( thisDisplay, thisWindow, X, Y-maxHeight, width, maxHeight, FALSE ) ;
   XFillRectangle ( thisDisplay, thisWindow, thisGC, X, Y-height, width, height) ;
}


/* Displays the average data inside of Packet */
```

```
AvgTokenPaint(thisDisplay, thisWindow, thisGC, percent, mode, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int percent ;
int mode ;
struct Monitor MonitorData ;
{
   extern struct BarData tokenBarData[] ;
   static int topIndex = 0 ;
   static double averageSize = 0, sizeTotal=0, numberOfTest=0;
   static char string[ 60 ] ;

   int X, Y, dispHeight, dispWidth, maxBars, barWidth = 10, xOffset = 8, cnt, index ;

   /* Get the current attributes of the window so we can repaint it properly */
   XGetWindowAttributes( thisDisplay, thisWindow, &windowAttributes ) ;

   dispHeight = ((MENUBOTTOMLINE+((windowAttributes.height-MENUBOTTOMLINE)-TOPLINEINDENT)) - MENUBOTTOMLINE ) ;
   dispWidth  = ((windowAttributes.width - TOPLINEINDENT) - TOPLINEINDENT ) ;

   X =  (windowAttributes.width - TOPLINEINDENT - xOffset - 30 - 7 ) ;
   Y =  (MENUBOTTOMLINE + dispHeight - 2) ;

   /* maximum number of bars that could fit in this window */
   maxBars = ( (dispWidth / (barWidth+2) ) - 4 ) ;

   if ( topIndex > maxBars )
       topIndex = 0 ;

   if (mode == ADD_DATA)
   {
       tokenBarData[ topIndex ].percent = percent ;
       numberOfTest++;
       sizeTotal += percent ;

       averageSize = (  sizeTotal / numberOfTest ) ;

       sprintf( string, "Ring #: %d    Address: %s",
       MonitorData.ringSelected, MonitorData.addSelected) ;
       XDrawImageString ( thisDisplay,thisWindow,thisGC,20,10,string,strlen( string ) ) ;

       sprintf( string, "Average Data Size: %f", averageSize ) ;
       XDrawImageString ( thisDisplay,thisWindow,thisGC,20,30,string,strlen( string ) ) ;

   }

   index = topIndex ;

   for (cnt = 0; cnt < maxBars; cnt++ )
   {
       if (tokenBarData[index].percent < 0)
       {
          if (cnt==0)
             topIndex = 0 ;
          break ;
       }
       else
       {
          X -= 12 ;
          DrawBar(thisDisplay, thisWindow, thisGC, X, Y, dispHeight, 10, tokenBarData[index].percent ) ;
       }
       index-- ;
       if (index == -1)
          index = maxBars-1 ;
   }


   if (mode==ADD_DATA)
       topIndex++ ;
   if (cnt==0)
       topIndex = 0 ;
}
```

```
/* FILE: TRACKING.C
   AUTHOR: RICHARD CZOP
   DATE: 12/13/91
   REAL-TV
*/
#include "monitor.h"
#include <math.h>
#include <X11/cursorfont.h>
#include "topology.h"
#include "process.h"

StuffProcessData()
{
   int index = 0 ;

   char Address[ NODEADDRESS+1 ] ;
   int  ProcessState ;
   char Source[ NODEADDRESS+1 ] ;
   char Desc[ NODEADDRESS+1 ] ;
   int Interval ;

   FILE *fileptr ;          /* ptr to file                          */

   /* Attempt to open the file */
   if ( (fileptr=fopen( "process.test", "r" ) ) == NULL )
   {
       printf( "Can't open file: process.test \n");
       exit( -1 );
   }
   /* FILE PROCESS.TEST
       node address, process type (CU=1,TOKEN=2,PROCESSING=3), src address, dest address, processing time
   */
   while (fscanf( fileptr, "%5s %1d %5s %5s %3d", Address, &ProcessState, Source, Desc, &Interval) != EOF)
   {
       strcpy( ProcessTest[ index ].Address, Address ) ;
       strcpy( ProcessTest[ index ].Source, Source ) ;
       strcpy( ProcessTest[ index ].Desc, Desc ) ;
       ProcessTest[ index ].ProcessState = ProcessState ;
       ProcessTest[ index ].Interval     = Interval ;

       index++ ;
       if (index > MAXPROCESSTEST)
       {
          fprintf( stderr, "OVERFLOW ERROR: ProcessTest[] \n " ) ;
          exit( -1 ) ;
       }
   }
   ProcessTest[ index ].ProcessState = EMPTY ;

   fclose( fileptr ) ;

}

/* Starts the process tracking -- sets up all the data structs and Marks controlling CLU and current place of
   process on the network graph

   Called only ONCE to start process tracking
*/
ProcessTrack( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   /* Put up a window for the process tracking statistics */
   ProcessWindow(thisDisplay, thisWindow, thisGC, MonitorData ) ;

   if (ProcessData.status == TRACKING_OFF)
       ProcessData.status = TRACKING_ON ;

   ProcessData.pid = 1345 ;
   ProcessData.activeTime = 0 ;
   strcpy( ProcessData.p_Name, "SimpleMath" ) ;

   /*ProcessNext( thisDisplay, thisWindow, thisGC, ProcessTest[ ProcessData.TestIndex ].Address ) ; */
}


/* Called x number of times after ProcessTrack() has been called to init process tracking
   This function will show the next move of the process migration path
   and update the correct data structures
*/
```

```
ProcessNext(thisDisplay, thisWindow, thisGC, nodeAddress )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
char *nodeAddress;
{
   extern struct topologyData Topology[] ;
   int index ;

   if (ProcessData.started == FALSE)
   {

       ProcessData.started = TRUE ;

       /* Initialize the data structure and mark CU */
       for(index=0; index <= RegisteredNodes.numberOfNodes; index ++ )
       {
          if (strcmp(nodeAddress,RegisteredNodes.NodeInfo[index].address) == 0)
          {
             MarkCU( thisDisplay, thisWindow, thisGC,
                 RegisteredNodes.NodeInfo[index].x,
                 RegisteredNodes.NodeInfo[index].y, NODERADIUS );

             RegisteredNodes.NodeInfo[index].nodeMarked = TRUE ;
             RegisteredNodes.NodeInfo[index].processState = CU ;
             ProcessData.CU_Index = index ;
             ProcessData.TestIndex++ ;/* TEST and DEBUG */

             break ;
          }
       }
   }
   else
   {
       /*fprintf( stderr, "Address '%s' State %d \n", nodeAddress,
                        ProcessTest[ProcessData.TestIndex].ProcessState ) ;
       */
       if (ProcessTest[ProcessData.TestIndex].ProcessState == EMPTY)
       {
          fprintf(stderr, "End of Process test data \n" ) ;
          RestoreNode( thisDisplay, thisWindow, thisGC ) ;

          ProcessData.TestIndex = 0 ;
       }
       else
       {

          /* Clear previously marked node and restore it to original state -- if not a CU, a node, and not
             currently processing
           */
          if (ProcessTest[ProcessData.TestIndex].ProcessState != CU
              && ProcessData.NODE_Index != EMPTY
              && RegisteredNodes.NodeInfo[index].processState != PROCESSING )
             RestoreNode( thisDisplay, thisWindow, thisGC ) ;

          /* Show next migration on network graph & update process window */
          for(index=0; index <= RegisteredNodes.numberOfNodes; index ++ )
          {
             if (strcmp(nodeAddress,RegisteredNodes.NodeInfo[index].address) == 0)
             {
                 if (ProcessTest[ProcessData.TestIndex].ProcessState == PROCESSING)
                 {
                    if ( ProcessData.Processing == TRUE )
                    {
                       ProcessData.Elapsed++ ;/* Increment Elapsed Time */

                       if( --ProcessData.Interval == 0 )
                       {
                           ProcessData.Processing = FALSE ;
                           ProcessData.TestIndex++ ;/* TEST and DEBUG */
                       }
                    }
                    else
                    {
                       ProcessData.Processing = TRUE ;
                       ProcessData.Interval = ProcessTest[ProcessData.TestIndex].Interval ;
                       ProcessData.Elapsed = 0 ;/* Reset elapsed time */
                       RegisteredNodes.NodeInfo[index].processState = PROCESSING ;
                       RegisteredNodes.NodeInfo[index].nodeMarked   = TRUE ;
                       ProcessData.NODE_Index = index ;
                       ProcessData.LASTOP_Index = index ;
```

```
                    }

                    MarkBusy( thisDisplay, thisWindow, thisGC, index ) ;
                 }
                 else
                 {
                    if (ProcessTest[ProcessData.TestIndex].ProcessState == TOKEN)
                    {
                       MarkToken( thisDisplay, thisWindow, thisGC,
                              RegisteredNodes.NodeInfo[index].x,
                              RegisteredNodes.NodeInfo[index].y, NODERADIUS );
                       RegisteredNodes.NodeInfo[index].processState = TOKEN ;
                       ProcessData.NODE_Index = index ;
                       RegisteredNodes.NodeInfo[index].nodeMarked   = TRUE ;

                    }
                    ProcessData.TestIndex++ ;/* TEST and DEBUG */

                 }

                 break ;
              }
           }
        }
   }

   PaintProcessStats() ;

}

PaintProcessStats ()
{
   static activeTime = 0;
   char char_activeTime[ 5 ], char_Elapsed[ 5 ], char_pid[ 5 ] ;
   char nodeTypeIs[ NODETYPELEN+1 ];

   sprintf( char_pid, "%d", ProcessData.pid ) ;
   /* Process ID */
   XDrawImageString (
          mainDisplay,
          processWindow,
          processGC,
          150,
          200,
          char_pid,
          strlen( char_pid ) );

   /* Process Name */
   XDrawImageString (
          mainDisplay,
          processWindow,
          processGC,
          150,
          220,
          ProcessData.p_Name,
          strlen( ProcessData.p_Name) ) ;

   sprintf( char_activeTime, "%d", ProcessData.activeTime++ ) ;
   XDrawImageString (
          mainDisplay,
          processWindow,
          processGC,
          150,
          240,
          char_activeTime,
          strlen( char_activeTime ) ) ;

   XDrawImageString (
          mainDisplay,
          processWindow,
          processGC,
          150,
          260,
          RegisteredNodes.NodeInfo[ProcessData.CU_Index].address,
          strlen(RegisteredNodes.NodeInfo[ProcessData.CU_Index].address) ) ;

   XDrawImageString (
          mainDisplay,
          processWindow,
          processGC,
          150,
          280,
```

```
            RegisteredNodes.NodeInfo[ProcessData.NODE_Index].address,
            strlen(RegisteredNodes.NodeInfo[ProcessData.NODE_Index].address) ) ;

    strcpy( nodeTypeIs, NodeTypeIs(RegisteredNodes.NodeInfo[ProcessData.NODE_Index].type) ) ;
    XDrawImageString (
            mainDisplay,
            processWindow,
            processGC,
            150,
            300,
            nodeTypeIs,
            strlen( nodeTypeIs ) );

    sprintf( char_Elapsed, "%d", ProcessData.Elapsed );
    XDrawImageString (
            mainDisplay,
            processWindow,
            processGC,
            150,
            320,
            char_Elapsed,
            strlen( char_Elapsed ) );

    if (ProcessData.LASTOP_Index != EMPTY )
    {
        strcpy( nodeTypeIs, NodeTypeIs(RegisteredNodes.NodeInfo[ProcessData.LASTOP_Index].type) ) ;
        XDrawImageString (
            mainDisplay,
            processWindow,
            processGC,
            150,
            353,
            nodeTypeIs,
            strlen( nodeTypeIs ) );
    }
}

RestoreNode( thisDisplay, thisWindow, thisGC )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
{
   int mainScreen;
   char nodeId[ 2 ] ;

   /* Create a GC for a blank line */
   GC localGC = XCreateGC ( thisDisplay, thisWindow, 0, 0 ) ;
   mainScreen  = DefaultScreen ( mainDisplay ) ;
   XSetBackground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;
   XSetForeground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;

   /* Clear Node & Repaint */
   XFillArc( thisDisplay, thisWindow, localGC,
          RegisteredNodes.NodeInfo[ProcessData.NODE_Index].x-NODERADIUS,
          RegisteredNodes.NodeInfo[ProcessData.NODE_Index].y-NODERADIUS,
          2*NODERADIUS, 2*NODERADIUS, 0, 360*64 ) ;

   DrawCircle( thisDisplay, thisWindow, thisGC,
          RegisteredNodes.NodeInfo[ProcessData.NODE_Index].x,
          RegisteredNodes.NodeInfo[ProcessData.NODE_Index].y,
          NODERADIUS ) ;

   sprintf( nodeId, "%d", RegisteredNodes.NodeInfo[ProcessData.NODE_Index].type ) ;

   /* Repaint node identifier */
   XDrawImageString ( thisDisplay, thisWindow, thisGC,
              RegisteredNodes.NodeInfo[ProcessData.NODE_Index].x,
              RegisteredNodes.NodeInfo[ProcessData.NODE_Index].y,
              nodeId,
              strlen( nodeId ) ) ;

}


/* Puts up a window for the process statistics */
ProcessWindow( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   extern Window processWindow ;
```

```
   if (MonitorData.processWindow == TRUE )
   {
       MonitorData.processWindow = FALSE ;
       MonitorData.processButIndex = EMPTY ;/* Indicates that no button is registered for this window */
       XUnmapWindow( thisDisplay, processWindow ) ;
   }
   else
   {
       MonitorData.processButIndex = EMPTY ;/* Indicates that no button is registered for this window */

       /* window mapping */
       XMapRaised ( thisDisplay, processWindow ) ;

       /* Delay Painting of the button and text */
       while( XCheckTypedEvent( thisDisplay, Expose, &mainEvent ) ) ;
          sleep( 1 ) ;

       PaintProcessWindow( thisDisplay, processWindow, processGC, MonitorData ) ;
       MonitorData.processWindow = TRUE ;
   }
}

PaintProcessWindow( thisDisplay, thisWindow, thisGC, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   int registerButton = FALSE, buttonIndex, mainScreen ;

   /* Create a GC for a blank line */
   GC localGC = XCreateGC ( thisDisplay, thisWindow, 0, 0 ) ;
   mainScreen  = DefaultScreen ( mainDisplay ) ;
   XSetBackground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;
   XSetForeground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;


   /* Draw Legend */
   /* Draw Controlling CU */
   DrawCircle( thisDisplay, thisWindow, thisGC, 50, 75, NODERADIUS ) ;
   MarkCU( thisDisplay, thisWindow, thisGC, 50, 75, NODERADIUS ) ;
   XDrawImageString (
          thisDisplay,
          thisWindow,
          thisGC,
          100,
          75,
          "Controlling CU",
          strlen( "Controlling CU" ) ) ;

   /* Draw current Node with data */
   DrawCircle( thisDisplay, thisWindow, thisGC, 50, 115, NODERADIUS ) ;
   MarkToken( thisDisplay, thisWindow, thisGC, 50, 115, NODERADIUS ) ;
   XDrawImageString (
          thisDisplay,
          thisWindow,
          thisGC,
          100,
          115,
          "Current Process Location",
          strlen( "Current Process Location" ) ) ;

   DrawCircle( thisDisplay, thisWindow, thisGC, 50, 155, NODERADIUS ) ;
   XFillArc( thisDisplay, thisWindow, thisGC, 50-4, 155-4,
                 2*4, 2*4, 0, 360*64 ) ;
   XDrawImageString (
          thisDisplay,
          thisWindow,
          thisGC,
          100,
          155,
          "Processing Current Process",
          strlen( "Processing Current Process" ) ) ;

   /* Statistics */
   XDrawImageString (
          mainDisplay,
          processWindow,
          processGC,
          7,
          200,
                  "Process ID              : ",
```

```
        strlen( "Process ID              : " ) );
XDrawImageString (
        mainDisplay,
        processWindow,
        processGC,
        7,
        220,
                "Process Name           : ",
        strlen("Process Name           : " ) ) ;
XDrawImageString (
        mainDisplay,
        processWindow,
        processGC,
        7,
        240,
                "Process Active For     :            (units) ",
        strlen("Process Active For     :            (units) " ) ) ;

XDrawImageString (
        mainDisplay,
        processWindow,
        processGC,
        7,
        260,
                "Controlling CU         :            (address)",
        strlen("Controlling CU         :            (address)") ) ;

XDrawImageString (
        mainDisplay,
        processWindow,
        processGC,
        7,
        280,
                     "Process at component   :            (address)",
        strlen("Process at component   :            (address)") ) ;

XDrawImageString (
        mainDisplay,
        processWindow,
        processGC,
        7,
        300,
                "Processing at component: ",
        strlen("Processing at component: ") );

XDrawImageString (
        mainDisplay,
        processWindow,
        processGC,
        7,
        320,
                "Processing elapsed time: ",
        strlen("Processing elapsed time: ") );

XDrawImageString (
        mainDisplay,
        processWindow,
        processGC,
        7,
        340,
                "Last operation ",
        strlen("Last operation ") );

XDrawImageString (
        mainDisplay,
        processWindow,
        processGC,
        7,
        353,
                "performed at           : ",
        strlen("performed at           : ") );

XDrawLine( thisDisplay, thisWindow, thisGC, 5, 7+BUTTONHEIGHT+5, PROCESSWINDOWWIDTH-5, 7+BUTTONHEIGHT+5 ) ;
XDrawLine( thisDisplay, thisWindow, thisGC, 5, 180, PROCESSWINDOWWIDTH-5, 180 ) ;

if (MonitorData.processButIndex == EMPTY)
    registerButton = TRUE ;

buttonIndex = DrawButton( thisDisplay,
        thisWindow,
        thisGC,
        20,
```

```
        7,
        BUTTONWIDTH,
        BUTTONHEIGHT,
        "DONE",
        NULL,
        registerButton ) ;

   if (registerButton == TRUE)
       MonitorData.processButIndex = buttonIndex ;/* Index of the button registered */

}

MarkBusy( thisDisplay, thisWindow, thisGC, nodeIndex )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int nodeIndex ;
{
   static int radius = 4 ;

   int mainScreen;
   int centerX, centerY;

   /* Create a GC for a blank line */
   GC localGC = XCreateGC ( thisDisplay, thisWindow, 0, 0 ) ;
   mainScreen  = DefaultScreen ( mainDisplay ) ;
   XSetBackground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;
   XSetForeground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;

   centerX = RegisteredNodes.NodeInfo[nodeIndex].x ;
   centerY = RegisteredNodes.NodeInfo[nodeIndex].y ;

   /* Clear Node & Repaint */
   XFillArc( thisDisplay, thisWindow, localGC, centerX-NODERADIUS, centerY-NODERADIUS,
               2*NODERADIUS, 2*NODERADIUS, 0, 360*64 ) ;

   DrawCircle( thisDisplay, thisWindow, thisGC, centerX, centerY, NODERADIUS ) ;

   XFillArc( thisDisplay, thisWindow, thisGC, centerX-radius, centerY-radius,
               2*radius, 2*radius, 0, 360*64 ) ;

   radius = radius+1 ;
   if (radius >= NODERADIUS-4)
       radius = 4 ;

}

MarkCU( thisDisplay, thisWindow, thisGC, centerX, centerY, radius )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int centerX ;
int centerY ;
int radius ;
{
   int thisradius, mainScreen ;

   /* Create a GC for a blank line */
   GC localGC = XCreateGC ( thisDisplay, thisWindow, 0, 0 ) ;
   mainScreen  = DefaultScreen ( mainDisplay ) ;
   XSetBackground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;
   XSetForeground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;

   thisradius = NODERADIUS-2 ;
   XFillArc( thisDisplay, thisWindow, thisGC, centerX-thisradius, centerY-thisradius,
               2*thisradius, 2*thisradius, 0, 360*64 ) ;
}

MarkToken( thisDisplay, thisWindow, thisGC, centerX, centerY, radius )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int centerX ;
int centerY ;
int radius ;
{
   int mainScreen, thisradius ;

   /* Create a GC for a blank line */
   GC localGC = XCreateGC ( thisDisplay, thisWindow, 0, 0 ) ;
   mainScreen  = DefaultScreen ( mainDisplay ) ;
   XSetBackground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;
```

```
    XSetForeground( thisDisplay, localGC, WhitePixel(thisDisplay, mainScreen) ) ;

    thisradius = NODERADIUS-2 ;
    XFillArc( thisDisplay, thisWindow, thisGC, centerX-thisradius, centerY-thisradius,
                2*thisradius, 2*thisradius, 0, 360*64 ) ;

    thisradius = NODERADIUS-(NODERADIUS-6) ;
    XFillArc( thisDisplay, thisWindow, localGC, centerX-thisradius, centerY-thisradius,
                2*thisradius, 2*thisradius, 0, 360*64 ) ;

}
```

```
/* FILE: MENUS.C
   AUTHOR: RICHARD CZOP
   DATE: 12/13/91
   REAL-TV
*/

/* Menus handling routines */

#include "monitor.h"

/* Detects button presses for menus  NOT dialog boxes */

int
CheckButtonPress( thisDisplay, thisWindow, thisGC, thisX, thisY, MonitorData, MenuName )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int thisX ;
int thisY ;
struct Monitor MonitorData ;
struct Menu MenuName[] ;

{
   int loopCnt = 0 ;
   int currentX = MAINMENUBUTTONX ;

   MonitorData.buttonSelected = -99 ;/* Dummy Flag */


   /* Pointer coordinate is within the menu bar */
   if ( (thisY >= MENUTOPLINE+5) && (thisY <= (MENUTOPLINE+5+MENUBUTTONHEIGHT+1)))
   {
       for( loopCnt = 0; loopCnt < MonitorData.menuItems; loopCnt++ )
       {
          /* Pointer within a menu button */
          if ( (thisX >= currentX) && (thisX <= (currentX+MENUBUTTONWIDTH+1)) )
          {
             MonitorData.buttonSelected = loopCnt ;
             ShowButtonPress( thisDisplay,
                    thisWindow,
                    thisGC,
                    currentX ) ;
             break ;
          }
          else
             currentX = (MENUBUTTONXSPACE + (currentX + MENUBUTTONWIDTH));

       }
   }


   if ( (MonitorData.buttonSelected != -99) &&
       (MenuName[MonitorData.buttonSelected].Func != NULL) )
       /* Button was pressed */
       (*MenuName[MonitorData.buttonSelected].Func) (thisDisplay,
             thisWindow,
             thisGC,
             &MonitorData ) ;

}



ShowButtonPress( this_Display, this_Window, this_GC, Xcoordinate )
Display *this_Display ;
Window this_Window ;
GC this_GC ;
int Xcoordinate ;
{
   int buttonY = MENUTOPLINE + 5 ;
   int buttonX = Xcoordinate ;
   int count ;
   GC buttonGC ;

   static char gray_bits [] = {
       0x88, 0x88, 0x22, 0x22, 0x88, 0x88, 0x22, 0x22,
       0x88, 0x88, 0x22, 0x22, 0x88, 0x88, 0x22, 0x22,
       0x88, 0x88, 0x22, 0x22, 0x88, 0x88, 0x22, 0x22,
       0x88, 0x88, 0x22, 0x22, 0x88, 0x88, 0x22, 0x22 } ;


   for ( count=0; count<3; count++ )
```

```
   {
       XDrawRectangle( this_Display,
          this_Window,
          this_GC,
          ++buttonX,        /* Upper x Left of rectangle */
          ++buttonY,        /* Upper y Left of rectangle */
          MENUBUTTONWIDTH,    /* Width of button          */
          MENUBUTTONHEIGHT    /* Height of button         */
          ) ;
   }
}


void
DisplayMenu ( thisDisplay, thisWindow, thisGC, MenuName, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Menu MenuName[] ;
struct Monitor MonitorData ;
{
   int menuIndex = 0 ;
   int buttonX = MAINMENUBUTTONX ;
   int buttonY = MENUTOPLINE + 5 ;


   /* Get the current attributes of the window so we can repaint it properly */
   XGetWindowAttributes( thisDisplay, thisWindow, &windowAttributes ) ;

   XClearArea( thisDisplay,
          thisWindow,
          TOPLINEINDENT,
          MENUTOPLINE+2,
          windowAttributes.width - (2*TOPLINEINDENT),
          (MENUBOTTOMLINE - MENUTOPLINE) - 2,
          FALSE ) ;

   MonitorData.menuItems = 0 ;


   /* Given a Menu Struct This algorithm will draw a Menu */
   while ( MenuName[ menuIndex ].choiceName != NULL )
   {

       MonitorData.menuItems++ ; /* How many items in this menu */

       /*XDrawArc( thisDisplay,
          thisWindow,
          thisGC,
          buttonX,
          buttonY,
          MENUBUTTONWIDTH,
          MENUBUTTONHEIGHT,
          0,
          360 * 64 ) ;
       */

       DrawMenuButton( thisDisplay,
             thisWindow,
             thisGC,
             buttonX,
             buttonY,
             MENUBUTTONWIDTH,
             MENUBUTTONHEIGHT,
             MenuName[ menuIndex ].choiceName ) ;


       if ( MonitorData.buttonSelected == menuIndex )
          ShowButtonPress( thisDisplay, thisWindow, thisGC, buttonX ) ;

       buttonX = (MENUBUTTONXSPACE + (buttonX + MENUBUTTONWIDTH));

       menuIndex++ ;
   }



}
```

```
/* FILE: TOPOLOGY.C
   AUTHOR: RICHARD CZOP
   DATE: 12/13/91
   REAL-TV
*/
#include "monitor.h"
#include <math.h>
#include <X11/cursorfont.h>
#include "topology.h"

int
CheckNodePress( thisDisplay, thisWindow, thisGC, x, y, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct Monitor MonitorData ;
{
   int radius,
       cnt;
   double X, Y, power=2 ;
   int NodeType = EMPTY;

   for (cnt=0; cnt < RegisteredNodes.numberOfNodes; cnt++ )
   {
       switch (RegisteredNodes.NodeInfo[cnt].gender)
       {
          case NODE:        /* component is a Node */
             radius = NODERADIUS;
             break ;
          case RING:        /* component is a Ring */
             radius = (2*NODERADIUS) ;
             break ;
          case BRIDGE:      /* component is a Bridge */
             radius = BRIDGERADIUS;
             break ;
          default:
             fprintf(stderr, "Error: CheckNodePress() \n" ) ;
             exit( -1 ) ;
       }

       X =  pow(( (double)x-(double)RegisteredNodes.NodeInfo[cnt].x), power) ;
       Y =  pow(( (double)y-(double)RegisteredNodes.NodeInfo[cnt].y), power) ;
       if ( (X+Y) <= pow((double)radius,power) )
       {
          NodeType = RegisteredNodes.NodeInfo[cnt].gender;
          MonitorData.ringSelected = RegisteredNodes.NodeInfo[cnt].ring ;
          strcpy(MonitorData.addSelected, RegisteredNodes.NodeInfo[cnt].address ) ;
          MonitorData.typeSelected = RegisteredNodes.NodeInfo[cnt].type ;

          /*fprintf( stderr, "Node clicked: ring = %d  type = %d  gender = %d  address = %s \n",
                   RegisteredNodes.NodeInfo[cnt].ring,
                   RegisteredNodes.NodeInfo[cnt].type,
                   RegisteredNodes.NodeInfo[cnt].gender,
                   RegisteredNodes.NodeInfo[cnt].address ) ;
          */
          break ;
       }
   }
   return NodeType ;
}


/* Register the coordinates for the node */
RegisterNode( centerX, centerY, gender, type, ring, address )
int centerX ;
int centerY ;
int gender ;
int type ;
int ring ;
char* address ;
{
   int index ;

   index = RegisteredNodes.currentIndex ;
   RegisteredNodes.numberOfNodes++ ;

   RegisteredNodes.NodeInfo[index].x = centerX ;
   RegisteredNodes.NodeInfo[index].y = centerY ;
   RegisteredNodes.NodeInfo[index].gender = gender ;
   RegisteredNodes.NodeInfo[index].type   = type ;
   strcpy( RegisteredNodes.NodeInfo[index].address, address ) ;
   RegisteredNodes.NodeInfo[index].ring = ring ;
```

```
   RegisteredNodes.currentIndex++ ;



}


double
AngleInRadians( numberOfNodes )
int numberOfNodes ;
{
   return (double) ((2 * PI) / numberOfNodes) ;/* Calculate the angle between connectors in radians */
}


PaintTopology(  thisDisplay, thisWindow, thisGC, Topology, MonitorData, Bridges )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct topologyData Topology[] ;
struct Monitor MonitorData ;
struct BridgeData Bridges[] ;
{
   Cursor clockCursor, defaultCursor ;

   char string_RingId[ 3 ] ;
   int cnt,
       Nodes,
       LineLen,
       Ring,
       centerWindowWidth,
       centerWindowHeight,
       CenterX,
       CenterY,
       TokenRingRadius;

   struct ringData RingData[ MAXRINGS ];
   double Rads ;


   clockCursor = XCreateFontCursor( thisDisplay, XC_watch ) ;
   defaultCursor = XCreateFontCursor( thisDisplay, XC_gumby ) ;

   XDefineCursor( thisDisplay, thisWindow, clockCursor ) ;

   /* Clear all bridges from processed list */
   for( cnt = 0; cnt < MonitorData.Bridges; cnt++ )
       Bridges[cnt].BridgeID = 0 ;

   /* Clear all Ring Data */
   for( cnt = 1; cnt <= MonitorData.Rings; cnt++ )
       RingData[cnt].processed = FALSE ;

   /* Clear all Registered Nodes */
   RegisteredNodes.currentIndex  = 0 ;
   RegisteredNodes.numberOfNodes = 0 ;

   Nodes   = NumberOfNodesOnRing( 0, MonitorData.Nodes, Topology ) ;
   Rads    = (double) AngleInRadians( Nodes ) ;
   LineLen = ConnectLength( Rads ) ;
   Ring = 1 ;

   /* Get the current attributes of the window so we can repaint it properly */
   XGetWindowAttributes( thisDisplay, thisWindow, &windowAttributes ) ;

   centerWindowWidth  = (windowAttributes.width / 2 ) ;
   centerWindowHeight = (windowAttributes.height / 2 ) ;

   CenterX = centerWindowWidth ;
   CenterY = centerWindowHeight ;

   XClearWindow( thisDisplay, thisWindow ) ;

   /* Put up processing message */
   XDrawImageString (
          thisDisplay,
          thisWindow,
          thisGC,
          MAINWINDOWX+MAINWINDOWWIDTH - 200,
          30,
          "Building network topology...",
          strlen("Building network topology...") ) ;
```

```
   /* Draw Initial Token Ring */
   TokenRingRadius = NODERADIUS + NODERADIUS ;
   DrawCircle( thisDisplay, thisWindow, thisGC, CenterX, CenterY, TokenRingRadius ) ;
   DrawCircle( thisDisplay, thisWindow, thisGC, CenterX, CenterY, TokenRingRadius-5 ) ;
   RegisterNode( CenterX, CenterY, RING, 0, Ring, "00000" ) ;

   sprintf( string_RingId, "%d", Ring ) ;

   XDrawImageString (
          thisDisplay,
          thisWindow,
          thisGC,
          CenterX,
          CenterY,
          string_RingId,
          strlen(string_RingId) ) ;



   RingData[1].centerX     = CenterX ;
   RingData[1].centerY     = CenterY ;
   RingData[1].radians     = Rads ;
   RingData[1].currentRadians = 0 ;
   RingData[1].lineLength     = LineLen ;


   DrawTop( thisDisplay, thisWindow, thisGC,
       Topology, MonitorData, &RingData,
       0, 0, 0,
       Bridges, Ring,
       0 ) ;

   XDefineCursor( thisDisplay, thisWindow, defaultCursor ) ;

}



int
ConnectLength( Radians )
double Radians ;
{
   int tmpLine, NodeRadius, LineLength ;

   NodeRadius = NODERADIUS;
   tmpLine = ((2*NodeRadius) /2 ) ;

   LineLength = (int) (tmpLine / sin( (.5*Radians) )) ;
   if (LineLength < MINLINELENGTH)
       LineLength = MINLINELENGTH ;

   return  LineLength ;
}

int
NumberOfNodesOnRing( TopPtr, nodes, Topology )
int TopPtr ;
int nodes ;
struct topologyData Topology[] ;

{
   int currentNodePtr = TopPtr,
       numberOfNodes ;

   /* Calculate the number of nodes on the current ring */
   for ( numberOfNodes = 0; currentNodePtr <= nodes; currentNodePtr++ )
   {
       if ( atoi(Topology[ currentNodePtr ].nodeId) == ENDOFRING )
          break ;

       numberOfNodes++ ;
   }
   return numberOfNodes ;
}

int
DrawTop( thisDisplay, thisWindow, thisGC,
   Topology, MonitorData, RingData,
   Current, Tmp, RingTop,
   Bridges, Ring,
   BridgeMatched )
Display *thisDisplay ;
```

```
Window thisWindow ;
GC thisGC ;
struct topologyData Topology[] ;
struct Monitor MonitorData ;      /* Global monitor data-number of elements in topology file      */
struct ringData RingData[] ;      /* Relevant data to draw ring and nodes                          */
int Current ;              /* Pointer to the current node we are inspecting              */
int Tmp ;            /* Pointer used to walk down list of node in current ring        */
int RingTop ;              /* Pointer to the top of the current ring being processed        */
struct BridgeData Bridges[] ;     /* Struct holding bridges processed already                      */
int Ring ;              /* Ring currently Processing                                   */
int BridgeMatched ;        /* ID of bridge just matched                                     */
{
   char tempstring[ 10 ] ;

   int Bridge = 0 ;
   int Nodes = 0 ;
   int CurrentRing = Ring;
   struct parentData ParentData ;

   RingData[ Ring ].ringTop = RingTop ;
   while ( TRUE )
   {

       if ( (Current >= MonitorData.Nodes) || (Tmp >= MonitorData.Nodes) )
       {
          RingData[Ring].processed = TRUE;
          return 0 ;
       }
       /* Back to the bridge already matched-entire ring painted return */
       if ( atoi(Topology[Current].nodeId) == BridgeMatched )
       {
          /* Mark ring as processed */
          RingData[Ring].processed = TRUE;
          return 0 ;
       }

       if ( atoi(Topology[Current].nodeId) == ENDOFRING )
       {
          Current = RingData[Ring].ringTop ;
          Tmp     = Current ;
          continue ;
       }

       if ( atoi(Topology[Current].nodeId) > ENDOFRING )/* Node-process it */
       {
          DrawNode( thisDisplay, thisWindow, thisGC, RingData, Topology, Current, Ring ) ;
          Current++ ;  /* Move to the next node */
          Tmp++ ;
       }
       else if ( (atoi(Topology[Current].nodeId) < ENDOFRING) &&
             (BridgePainted( atoi(Topology[Current].nodeId), Bridges, MonitorData ) == FALSE) )
       {
          /* Finished processing all nodes */
          if ((Ring==1) && (atoi(Topology[Current].nodeId)==ENDOFRING) )
                   return 0 ;

          DrawBridge( thisDisplay, thisWindow, thisGC, RingData, Topology, Ring, Tmp ) ;
          ParentData.centerX    = RingData[Ring].centerX ;
          ParentData.centerY    = RingData[Ring].centerY ;
          ParentData.lineLength = RingData[Ring].lineLength ;
          ParentData.BridgeX    = RingData[Ring].BridgeX ;
          ParentData.BridgeY    = RingData[Ring].BridgeY ;

          Bridge = atoi(Topology[Current].nodeId) ;
          RingTop = Tmp ;

          Tmp++;
          RingTop++ ;
          while ( TRUE )
          {
             if ( atoi(Topology[Tmp].nodeId) == ENDOFRING )
             {
                 ParentData.currentRadians = RingData[Ring].currentRadians ;
                 if ( (Ring==1) && (atoi(Topology[Current].nodeId)==ENDOFRING) &&
                    (AllRingsProcessed( RingData, MonitorData.Rings ) == TRUE ) )
                       return 0 ;

                 CurrentRing++ ;
                 Tmp++;
                 RingTop = Tmp ;
             }
             if ( atoi(Topology[Tmp].nodeId) == Bridge )
```

```
            {
                SetMatchedBridge( Bridges, Bridge, MonitorData ) ;

                Nodes = NumberOfNodesOnRing( RingTop,
                          MonitorData.Nodes,
                          Topology ) ;

                RingData[CurrentRing].radians    = AngleInRadians( Nodes ) ;
                RingData[CurrentRing].lineLength = ConnectLength( RingData[CurrentRing].radians ) ;
                RingData[CurrentRing].currentRadians = ParentData.currentRadians+(180*ONERADIAN) ;

                DrawChildRing( thisDisplay, thisWindow, thisGC,
                      RingData, Ring, CurrentRing, ParentData ) ;

                RingData[Ring].currentRadians += RingData[Ring].radians ;

                DrawTop( thisDisplay, thisWindow, thisGC,
                      Topology, MonitorData, RingData,
                      Tmp+1, Tmp+1, RingTop,
                      Bridges, CurrentRing,
                      Bridge );

                CurrentRing = Ring ;/* Reset current ring counter */
                Current++ ;
                Tmp = Current ;

                if ((Ring==1) && (atoi(Topology[Current].nodeId)==ENDOFRING) )
                   return 0 ;

                break ;
            }
            else
            {
                Tmp++ ;
            }
         }
      }

   }

}

int
AllRingsProcessed( RingData, Rings )
struct ringData RingData [] ;
int Rings ;
{
   /* Returns true if all rings 2 thru n have been processed otherwise returns false */
   int cnt;
   for ( cnt=2; cnt <= Rings; cnt++ )
   {
       if ( RingData[cnt].processed == FALSE )
          return FALSE ;
   }
   return TRUE ;
}


DrawChildRing( thisDisplay, thisWindow, thisGC,RingData, Ring, CurrentRing, ParentData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct ringData RingData[] ;
int Ring ;
int CurrentRing ;
struct parentData ParentData ;
{
   char string_RingId[3] ;
   /* Calculate the length of the new line extending from the parent ring to the center of the
       child ring

       Add: Radius of token ring + Parents lineLength (extending connector) +
          The bridge symbol + childs line length + the radius of a ring
   */
   int ChildCenterX, ChildCenterY, ChildRingLine, LineX, LineY;
   int TokenRingRadius = (2*NODERADIUS) ;
   int newLineLength = TokenRingRadius+ParentData.lineLength + (2*NODERADIUS) + RingData[Ring].lineLength +
          TokenRingRadius ;

   /* Calculate the center points for the child token ring */
   ChildCenterY = ParentData.centerY-(newLineLength*sin(ParentData.currentRadians)) ;
   ChildCenterX = ParentData.centerX+(newLineLength*cos(ParentData.currentRadians)) ;
```

```
   /* Calculate the connecting line length & coordinates from the parent ring to the child ring */
   ChildRingLine = newLineLength - TokenRingRadius ;
   LineY   = ParentData.centerY-(ChildRingLine*sin(ParentData.currentRadians)) ;
   LineX   = ParentData.centerX+(ChildRingLine*cos(ParentData.currentRadians)) ;

   /* Draw Bridge Connector extending from child ring drawn above */
   XDrawLine( thisDisplay, thisWindow, thisGC, ParentData.BridgeX, ParentData.BridgeY, LineX, LineY ) ;

   /* Draw Child Token Ring */
   DrawCircle( thisDisplay, thisWindow, thisGC, ChildCenterX, ChildCenterY, TokenRingRadius ) ;
   DrawCircle( thisDisplay, thisWindow, thisGC, ChildCenterX, ChildCenterY, TokenRingRadius-5 ) ;

   sprintf( string_RingId, "%d", CurrentRing ) ;

   XDrawImageString (
          thisDisplay,
          thisWindow,
          thisGC,
          ChildCenterX,
          ChildCenterY,
          string_RingId,
          strlen(string_RingId) ) ;

   /* Save coordinates of the center point for the child ring */
   RingData[CurrentRing].centerX = ChildCenterX ;
   RingData[CurrentRing].centerY = ChildCenterY ;
   RingData[CurrentRing].currentRadians += RingData[CurrentRing].radians ;

   RegisterNode( ChildCenterX, ChildCenterY, RING, 0, CurrentRing, "00000" );

}



SetMatchedBridge( Bridges, id, MonitorData )
struct BridgeData Bridges[] ;
int id ;
struct Monitor MonitorData ;
{
   /* Flag the bridge (id) already processed */
   int cnt ;
   for( cnt = 0; cnt < MonitorData.Bridges; cnt++ )
   {
       if ( Bridges[cnt].BridgeID == 0 )
       {
          Bridges[cnt].BridgeID = id ;
          break ;
       }
   }
}


DrawBridge( thisDisplay, thisWindow, thisGC, RingData, Topology, Ring, currentNodePtr )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct ringData RingData[] ;
struct topologyData Topology[] ;
int Ring ;
int currentNodePtr ;
{
   int CircleY, CircleX, LineY, LineX, NodeY, NodeX, BridgeY, BridgeX, BridgeOffset=0 ;

   int TokenRingRadius = (2*NODERADIUS) ;

   /* Calculate points on circle (start of line), points where line ends (on the node),
       and center of the nodes
   */
   CircleY = RingData[Ring].centerY-
       (TokenRingRadius*sin(RingData[Ring].currentRadians)) ;
   LineY   = RingData[Ring].centerY-
       ((TokenRingRadius+BridgeOffset+(RingData[Ring].lineLength-NODERADIUS))*
          sin(RingData[Ring].currentRadians)) ;
   NodeY   = RingData[Ring].centerY-
       ((TokenRingRadius+RingData[Ring].lineLength+BridgeOffset)*sin(RingData[Ring].currentRadians)) ;
   BridgeY = RingData[Ring].centerY-
       ((TokenRingRadius+BridgeOffset+RingData[Ring].lineLength+NODERADIUS)*
          sin(RingData[Ring].currentRadians)) ;

   CircleX = RingData[Ring].centerX+
       (TokenRingRadius*cos(RingData[Ring].currentRadians)) ;
   LineX   = RingData[Ring].centerX+
```

```
        ((TokenRingRadius+BridgeOffset+(RingData[Ring].lineLength-NODERADIUS))*
           cos(RingData[Ring].currentRadians)) ;

    NodeX   = RingData[Ring].centerX+
        ((TokenRingRadius+BridgeOffset+RingData[Ring].lineLength)*cos(RingData[Ring].currentRadians)) ;

    BridgeX  = RingData[Ring].centerX+
        ((TokenRingRadius+BridgeOffset+RingData[Ring].lineLength+NODERADIUS)*
           cos(RingData[Ring].currentRadians)) ;

    /* Draw Connector */
    XDrawLine( thisDisplay, thisWindow, thisGC, CircleX, CircleY, LineX, LineY ) ;

    /* Draw Node */
    DrawCircle( thisDisplay, thisWindow, thisGC, NodeX, NodeY, NODERADIUS ) ;


    XDrawImageString (
           thisDisplay,
           thisWindow,
           thisGC,
           NodeX,
           NodeY,
           Topology[currentNodePtr].nodeId,
           strlen(Topology[currentNodePtr].nodeId ) ) ;

    /* Save the points of the bridge */
    RingData[Ring].BridgeX = BridgeX ;
    RingData[Ring].BridgeY = BridgeY ;

    RegisterNode( NodeX, NodeY, BRIDGE, atoi(Topology[currentNodePtr].nodeId), Ring,
              Topology[currentNodePtr].nodeAddress ) ;

}

int
BridgePainted ( id, Bridges, MonitorData )
int id ;
struct BridgeData Bridges [] ;
struct Monitor MonitorData ;
{
    /* Returns true if the bridge was already painted otherwise returns false */
    int cnt;

    for ( cnt = 0 ; cnt < MonitorData.Bridges; cnt ++ )
    {
        if (Bridges[cnt].BridgeID == id)
           return TRUE ;
    }

    return FALSE ;
}

DrawNode( thisDisplay, thisWindow, thisGC, RingData, Topology, currentNodePtr, Ring )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct ringData RingData[] ;
struct topologyData Topology[] ;
int currentNodePtr ;
int Ring ;
{

    int CircleY, CircleX, LineY, LineX, NodeY, NodeX ;

    int TokenRingRadius = (2*NODERADIUS) ;

    /* Calculate points on circle (start of line), points where line ends (on the node),
        and center of the nodes
    */
    CircleY = RingData[Ring].centerY-
        (TokenRingRadius*sin(RingData[Ring].currentRadians)) ;
    LineY   = RingData[Ring].centerY-
        ((TokenRingRadius+(RingData[Ring].lineLength-NODERADIUS))*sin(RingData[Ring].currentRadians)) ;
    NodeY   = RingData[Ring].centerY-
        ((TokenRingRadius+RingData[Ring].lineLength)*sin(RingData[Ring].currentRadians)) ;

    CircleX = RingData[Ring].centerX+
        (TokenRingRadius*cos(RingData[Ring].currentRadians)) ;
    LineX   = RingData[Ring].centerX+
        ((TokenRingRadius+(RingData[Ring].lineLength-NODERADIUS))*cos(RingData[Ring].currentRadians)) ;
    NodeX   = RingData[Ring].centerX+
```

```
        ((TokenRingRadius+RingData[Ring].lineLength)*cos(RingData[Ring].currentRadians)) ;

    /* Draw Connector */
    XDrawLine( thisDisplay, thisWindow, thisGC, LineX, LineY, CircleX, CircleY ) ;

    /* Draw Node */
    DrawCircle( thisDisplay, thisWindow, thisGC, NodeX, NodeY, NODERADIUS ) ;


    XDrawImageString (
           thisDisplay,
           thisWindow,
           thisGC,
           NodeX,
           NodeY,
           Topology[currentNodePtr].nodeId,
           strlen(Topology[currentNodePtr].nodeId ) ) ;

    /* Debug */
    /*MarkCLU( thisDisplay, thisWindow, thisGC, NodeX, NodeY, NODERADIUS ) ;*/
    /*MarkToken( thisDisplay, thisWindow, thisGC, NodeX, NodeY, NODERADIUS ) ;*/

    RingData[Ring].currentRadians += RingData[Ring].radians ;

    RegisterNode( NodeX, NodeY, NODE, atoi(Topology[currentNodePtr].nodeId), Ring,
                  Topology[currentNodePtr].nodeAddress ) ;

}

DrawTopology ( thisDisplay, thisWindow, thisGC, Topology, MonitorData )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
struct topologyData Topology[] ;
struct Monitor MonitorData ;
{
    int atoi() ;

    int    X             = 100,
        Y             = 125,
        Width         = TOKENRINGWIDTH,
        Height        = TOKENRINGHEIGHT,
        CenterX       = (X+(Width/2)),
        CenterY       = (Y+(Height/2)),
        Radius        = (TOKENRINGWIDTH/2),
        CircleX       = 0,
        CircleY       = 0,
        LineX         = 0,
        LineY         = 0;

    int    lineLength    = 40,
        prevLineLen,
        n_Type, n_Address,
        ringCnt       = 0,
        nodeCnt       = 0,
        numberOfNodes = 0,
        numberOfRings = 0,
        currentAngle  = 0,
        currentNodePtr = 0,
        centerWindowWidth = 0,
        centerWindowHeight = 0;

    double  Radians       = 0,
        currentRadians = 0 ;

    int NodeRadius, NodeX, NodeY ;
    int TokenRingRadius, tmpLine ;

    /* Get the current attributes of the window so we can repaint it properly */
    XGetWindowAttributes( thisDisplay, thisWindow, &windowAttributes ) ;

    centerWindowWidth  = (windowAttributes.width / 2 ) ;
    centerWindowHeight = (windowAttributes.height / 2 ) ;

    CenterX = centerWindowWidth ;
    CenterY = centerWindowHeight ;

    numberOfRings = MonitorData.Rings ;/* Figure out the number of rings to display   */

    currentNodePtr = 0 ;

    /* For each ring on the network */
```

```
   for ( ringCnt=0; ringCnt < 1; ringCnt++ )
   {
      /*DrawToken( thisDisplay, thisWindow, thisGC, CenterX, CenterY, Radius-7 ) ;*/

      /* Calculate the number of nodes on the current ring */
      for ( numberOfNodes = 0; currentNodePtr <= MonitorData.Nodes; currentNodePtr++ )
      {
         if ( atoi(Topology[ currentNodePtr ].nodeId) == ENDOFRING )
            break ;

         numberOfNodes++ ;
      }

      /*fprintf( stderr, "Number of Nodes = %s \n", numberOfNodes ) ;
*/
      /* Reset pointer to start of token ring description */
      currentNodePtr = currentNodePtr - numberOfNodes ;

      Radians = ( (2 * PI) / numberOfNodes) ;/* Calculate the angle in radians */
      currentRadians = 0 ;

      /* calculate the length of the line from the angle between line and
         known radius of a node */
      NodeRadius = ( (( 2 * (lineLength+Radius) ) * sin(Radians/2) ) / 2 ) - 5 ;

      NodeRadius = 15;
      tmpLine = ((2*NodeRadius)) /2;
      lineLength = tmpLine / sin( (.5*Radians) ) ;

      /* Draw Token Ring */
      TokenRingRadius = NodeRadius + 15 ;
      DrawCircle( thisDisplay, thisWindow, thisGC, CenterX, CenterY, TokenRingRadius ) ;
      DrawCircle( thisDisplay, thisWindow, thisGC, CenterX, CenterY, TokenRingRadius-5 ) ;

      /* Draw nodes and connectors */
      for (nodeCnt=0 ; nodeCnt < numberOfNodes; nodeCnt++ )
      {
         /* Calculate what type of node and address of the node */
         n_Type    = atoi( Topology[ currentNodePtr ].nodeId );
         n_Address = atoi( Topology[ currentNodePtr ].nodeAddress );

         if ( n_Type < 0 )   /* Node is a Bridge */
         {
            prevLineLen = lineLength ;
            lineLength =  (2*lineLength) ;
         }

         CircleY = CenterY-(TokenRingRadius*sin(currentRadians)) ;
         LineY   = CenterY-((TokenRingRadius+(lineLength-NodeRadius))*sin(currentRadians)) ;
         NodeY   = CenterY-((TokenRingRadius+lineLength)*sin(currentRadians)) ;

         CircleX = CenterX+(TokenRingRadius*cos(currentRadians)) ;
         LineX   = CenterX+((TokenRingRadius+(lineLength-NodeRadius))*cos(currentRadians)) ;
         NodeX   = CenterX+((TokenRingRadius+lineLength)*cos(currentRadians)) ;


         /* Draw Connector */
         XDrawLine( thisDisplay, thisWindow, thisGC, CircleX, CircleY, LineX, LineY ) ;

         /* Draw Node */
         if ( n_Type > 0 )
            DrawCircle( thisDisplay, thisWindow, thisGC, NodeX, NodeY, NodeRadius ) ;
         else
            lineLength = prevLineLen ;/* Reset lineLength to original */

         XDrawImageString (
            thisDisplay,
            thisWindow,
            thisGC,
            NodeX,
            NodeY,
            Topology[currentNodePtr].nodeId,
            strlen(Topology[currentNodePtr].nodeId ) ) ;

         currentNodePtr++ ;

         currentRadians += Radians ;
      }
   }
}

DrawToken( thisDisplay, thisWindow, thisGC, CenterX, CenterY, Radius )
```

```
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int CenterX ;
int CenterY ;
int Radius ;
{
   double OneRadian    = .017453292,
       circleRadians = 0 ;
   int radCnt = 0 ;

   int CircleY, CircleX,
       Percision = 32 ;

   circleRadians = OneRadian/Percision ;

   /* Draw Circle *WARNING-DON NOT USE XDrawArc() */
   for ( radCnt = 0; radCnt < 10*Percision; radCnt++ )
   {
       CircleY = CenterY - ( Radius * sin( circleRadians ) ) ;
       CircleX = CenterX + ( Radius * cos( circleRadians ) ) ;
       XDrawPoint( thisDisplay, thisWindow, thisGC, CircleX, CircleY ) ;

       CircleY = CenterY - ( Radius-2 * sin( circleRadians ) ) ;
       CircleX = CenterX + ( Radius-2 * cos( circleRadians ) ) ;
       XDrawPoint( thisDisplay, thisWindow, thisGC, CircleX, CircleY ) ;

       CircleY = CenterY - ( Radius-4 * sin( circleRadians ) ) ;
       CircleX = CenterX + ( Radius-4 * cos( circleRadians ) ) ;
       XDrawPoint( thisDisplay, thisWindow, thisGC, CircleX, CircleY ) ;

       circleRadians += OneRadian/Percision ;
   }
}



DrawCircle( thisDisplay, thisWindow, thisGC, CenterX, CenterY, Radius )
Display *thisDisplay ;
Window thisWindow ;
GC thisGC ;
int CenterX ;
int CenterY ;
int Radius ;
{
   double OneRadian    = .017453292,
       circleRadians = 0 ;
   int radCnt = 0 ;

   int CircleY, CircleX,
       Percision = 1 ;     /* Define the quality of the outputted circle drawn--the higher the number
                   the better the quality and the slower the drawing process */

   circleRadians = OneRadian/Percision ;

   /* Draw Circle *WARNING-DON NOT USE XDrawArc() */
   for ( radCnt = 0; radCnt < 360*Percision; radCnt++ )
   {
       CircleY = CenterY - ( Radius *  sin( circleRadians ) ) ;
       CircleX = CenterX + ( Radius * cos( circleRadians ) ) ;

       XDrawPoint( thisDisplay, thisWindow, thisGC, CircleX, CircleY ) ;
       circleRadians += OneRadian/Percision ;
   }

   XFlush( thisDisplay ) ;
}


StuffTopology( Top, fileName, Nodes, Rings, Bridges )
struct topologyData Top[] ;
char *fileName ;
int *Nodes ;
int *Rings ;
int *Bridges ;
{
   int index = 0, ringCnt = 0, numberOfBridges ;
   char ID[ NODEID ] ;
   char Address[ NODEADDRESS ] ;
   FILE *fileptr ;        /* ptr to file                          */

   *Nodes   = 0 ;
```

```
    *Rings   = 0 ;
    *Bridges = 0 ;


    /* Attempt to open the file */
    if ( (fileptr=fopen( fileName, "r" ) ) == NULL )
    {
        printf( "Can't open file: %s\n", fileName );
        exit( -1 );
    }

    while (fscanf( fileptr, "%3s %5s", ID, Address ) != EOF)
    {
        strcpy( Top[ index ].nodeId, ID ) ;
        strcpy( Top[ index ].nodeAddress, Address ) ;

        index++ ;
        if ( index > MAXNODES )
        {
           fprintf( stderr,"Memory overflow error loading topology! \n" );
           fclose( fileptr ) ;
           exit( -1 );
        }

        /* End of a ring description */
        if ( atoi( ID ) == ENDOFRING )
           ringCnt++ ;

        /* Bridge */
        if ( atoi( ID ) < 0 )
           numberOfBridges++;
    }
    fclose( fileptr ) ;

    *Nodes   = index ;         /* Number of nodes & rings on entire connected network */
    *Rings   = ringCnt ;          /* Number of rings connected to network       */
    *Bridges = (numberOfBridges/2) ;  /* Number of bridges on the network             */

}
```

```
void Configure1() ;
void Configure2() ;
void Configure3() ;

void AboutBox() ;
void Load() ;
void Connect() ;
void Stats() ;
void Keyboard() ;
void Windows() ;

void RecordSession() ;
void LoadReplay() ;
void CleanUp() ;
void TrafficWindow() ;
void ZoomWindow() ;
void TokenWindow() ;

void LoadMainMenu() ;
void Legend();
void ProcessTrack() ;

static struct Menu MainMenu[] = {
            {"About",    AboutBox },
            {"File...",  Load     },
            {"Connect",  Connect  },
            {"Window...",Windows  },
            {"Process",  ProcessTrack },
            {"Legend",   Legend   },
            {NULL,NULL}
            } ;


static struct Menu LoadMenu[] = {
            {"Record", RecordSession},
            {"Replay", LoadReplay},
            {"Quit",   CleanUp },
            {"Main",   LoadMainMenu },
            {NULL, NULL }
            } ;

static struct Menu WindowMenu[] = {
            {"Keyboard", Keyboard    },
            {"Traffic", TrafficWindow},
            {"Main",   LoadMainMenu  },
            {NULL, NULL }
            } ;


static struct Menu ConfigMenu[] = {
            {"Config1", Configure1 },
            {"Config2", Configure2 },
            {"Config3", Configure3 },
            {"Main",   LoadMainMenu },
            {NULL,NULL}
            } ;

struct Menu *MenuSet[ NUMBEROFMENUS ] ; /*= { MainMenu, LoadMenu, ConfigMenu }; */

struct RegButtons RegisteredButtons ;
```

```
/* FILE: PROCESS.H */

#define TRACKING_ON     1
#define TRACKING_OFF    0

struct Process
{
   int status;          /* Is tracking on or off? */
   int started;         /* Has tracking started yet? */
   int activeTime;         /* Length of process activation */
   int pid;         /* Process ID */
   char p_Name[ 15 ];      /* Process name */
   int LASTOP_Index ;      /* Index of the node the last operation was performed at */
   int CU_Index;           /* CU's index into the topology graph data */
   int NODE_Index;         /* Index of the node that the process is currently at */
   int Processing;         /* Are we processing something? */
   int Interval;           /* Time left to process */
   int Elapsed;         /* Time already processed */
   int TestIndex;          /* TEST and DEBUG ONLY index of the next process tracking activity */

} ;

struct P_Test
{
   char Address[ NODEADDRESS + 1 ];
   char Source[ NODEADDRESS + 1 ];
   char Desc[ NODEADDRESS + 1 ];
   int ProcessState;
   int Interval ;
} ;
```

```
/* FILE: MONITOR.H */

/* Include file for Monitor */
/* X include files */
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <X11/keysym.h>
#include <X11/Xresource.h>
#include <X11/X10.h>

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <ctype.h>

#define  TRUE           1
#define  FALSE          0

/* Mouse buttons */
#define  LEFTBUTTON     1
#define  CENTERBUTTON      2
#define  RIGHTBUTTON       3


#define  ADD_DATA       1
#define  MAXBARS        75

#define  TOPLINEINDENT     10
#define  MENUTOPLINE       5
#define  MENUBOTTOMLINE       45

#define  BUTTONTOP      0
#define  BUTTONRIGHT       1
#define  BUTTONLEFT     2
#define  BUTTONBOTTOM      3

#define  BUTTONPRESSED          0
#define  MENUBUTTONWIDTH   70
#define  MENUBUTTONHEIGHT  30
#define  MENUBUTTONXSPACE  15
#define  MAINMENUBUTTONX   40
#define  MAINMENUBUTTONTEXTY  28
#define  MAINMENUITEMS     4

#define  NUMBEROFMENUS     4
#define  MAXNUMBEROFBUTTONS     20      /* Maximum number of buttons per dialog box */
#define  BUTTONHEIGHT      30
#define  BUTTONWIDTH       60

#define  MAINWINDOWX       5
#define  MAINWINDOWY       5
#define  MAINWINDOWWIDTH   850
#define  MAINWINDOWHEIGHT  700
#define  MAINWINDOWMINWIDTH   MAINWINDOWWIDTH
#define  MAINWINDOWMINHEIGHT  MAINWINDOWHEIGHT
#define  MAINWINDOWMAXWIDTH   MAINWINDOWWIDTH
#define  MAINWINDOWMAXHEIGHT  MAINWINDOWHEIGHT


#define  ABOUTWINDOWX      250
#define  ABOUTWINDOWY      300
#define  ABOUTWINDOWWIDTH  300
#define  ABOUTWINDOWHEIGHT    300
#define  ABOUTWINDOWMINWIDTH  300
#define  ABOUTWINDOWMINHEIGHT 300

#define  TRAFFICWINDOWX       MAINWINDOWX
#define  TRAFFICWINDOWY       (MAINWINDOWY + MAINWINDOWHEIGHT + 30)
#define  TRAFFICWINDOWWIDTH   MAINWINDOWWIDTH
#define  TRAFFICWINDOWHEIGHT  125
#define  TRAFFICWINDOWMINWIDTH   MAINWINDOWWIDTH
#define  TRAFFICWINDOWMINHEIGHT  TRAFFICWINDOWHEIGHT
#define  TRAFFICWINDOWMAXWIDTH   MAINWINDOWWIDTH
#define  TRAFFICWINDOWMAXHEIGHT  TRAFFICWINDOWHEIGHT

#define  ZOOMWINDOWX       MAINWINDOWX+MAINWINDOWWIDTH+10
#define  ZOOMWINDOWY       MAINWINDOWY
#define  ZOOMWINDOWWIDTH   275
#define  ZOOMWINDOWHEIGHT  350
#define  ZOOMWINDOWMINWIDTH   ZOOMWINDOWWIDTH
```

```
#define  ZOOMWINDOWMINHEIGHT  ZOOMWINDOWHEIGHT
#define  ZOOMWINDOWMAXWIDTH   ZOOMWINDOWWIDTH
#define  ZOOMWINDOWMAXHEIGHT  ZOOMWINDOWHEIGHT

#define  PROCESSWINDOWX       MAINWINDOWX+MAINWINDOWWIDTH-15
#define  PROCESSWINDOWY       MAINWINDOWY+100
#define  PROCESSWINDOWWIDTH   300
#define  PROCESSWINDOWHEIGHT  450
#define  PROCESSWINDOWMINWIDTH   PROCESSWINDOWWIDTH
#define  PROCESSWINDOWMINHEIGHT  PROCESSWINDOWHEIGHT
#define  PROCESSWINDOWMAXWIDTH   PROCESSWINDOWWIDTH
#define  PROCESSWINDOWMAXHEIGHT  PROCESSWINDOWHEIGHT

#define  TOKENWINDOWX      MAINWINDOWX+MAINWINDOWWIDTH-135
#define  TOKENWINDOWY      MAINWINDOWY+ZOOMWINDOWHEIGHT+30
#define  TOKENWINDOWWIDTH  400
#define  TOKENWINDOWHEIGHT    350
#define  TOKENWINDOWMINWIDTH  TOKENWINDOWWIDTH
#define  TOKENWINDOWMINHEIGHT TOKENWINDOWHEIGHT
#define  TOKENWINDOWMAXWIDTH  TOKENWINDOWWIDTH
#define  TOKENWINDOWMAXHEIGHT TOKENWINDOWHEIGHT

#define  BRIDGEWINDOWX     MAINWINDOWX+MAINWINDOWWIDTH-135
#define  BRIDGEWINDOWY     MAINWINDOWY+ZOOMWINDOWHEIGHT+60
#define  BRIDGEWINDOWWIDTH    400
#define  BRIDGEWINDOWHEIGHT   350
#define  BRIDGEWINDOWMINWIDTH BRIDGEWINDOWWIDTH
#define  BRIDGEWINDOWMINHEIGHT   BRIDGEWINDOWHEIGHT
#define  BRIDGEWINDOWMAXWIDTH BRIDGEWINDOWWIDTH
#define  BRIDGEWINDOWMAXHEIGHT   BRIDGEWINDOWHEIGHT

#define  LEGENDWINDOWX     0
#define  LEGENDWINDOWY     100
#define  LEGENDWINDOWWIDTH    200
#define  LEGENDWINDOWHEIGHT   350
#define  LEGENDWINDOWMINWIDTH LEGENDWINDOWWIDTH
#define  LEGENDWINDOWMINHEIGHT   LEGENDWINDOWHEIGHT
#define  LEGENDWINDOWMAXWIDTH LEGENDWINDOWWIDTH
#define  LEGENDWINDOWMAXHEIGHT   LEGENDWINDOWHEIGHT


/* Where are we now-Are we at a Different Menu Set or Are we at a different Function */
#define  MAINMENUSET      0
#define    LOADMENUSET    1
#define  CONFIGMENUSET    2
#define  WINDOWMENUSET    3
#define  STATSMENUSET     4


#define  SHELL         15
#define  CONNECT       16
#define  LOADPARAMS    17
#define    LOADREPLAY     18
#define  CLEANUP       19


extern Display   *mainDisplay ;
extern Window     mainWindow, aboutWindow, trafficWindow, zoomWindow, tokenWindow, legendWindow, bridgeWindow ;
extern Window     processWindow ;
extern GC         mainGC, aboutGC, trafficGC, zoomGC, tokenGC, legendGC, bridgeGC, processGC ;
extern XWindowAttributes windowAttributes ;
extern XEvent     mainEvent ;
extern KeySym     mainKey ;
extern XSizeHints mainHint, aboutHint, trafficHint, zoomHint, tokenHint, legendHint, bridgeHint, processHint ;
extern int        mainScreen ;
extern unsigned long myforeground, mybackground ;
extern char AppName [] ;
extern struct Menu *MenuSet[ ] ;
extern struct RegNodes RegisteredNodes ;
extern struct BarData tokenBarData[] ;
extern struct BarData testBarData[] ;
extern struct Process ProcessData ;

#define MAXPROCESSTEST    300

extern struct P_Test ProcessTest[] ;


/*---------------------------------------------------*/
/* Network Topology Data Structs */
/* Network Topology Defines */
#define TOKENRINGWIDTH     50
```

```
#define TOKENRINGHEIGHT       TOKENRINGWIDTH
#define TOKENRINGRADIUS       (TOKENRINGWIDTH / 2)

#define ONERADIAN   .017453292
#define MAXNODES       1000
#define MAXRINGS    100
#define MAXBRIDGES  200
#define NODEID         4           /* Includes NULL char */
#define NODEADDRESS 6
#define ENDOFRING   0
#define TOKENFORMAT    "%2s %6s"
#define NODERADIUS  15
#define BRIDGERADIUS    15

#define MINLINELENGTH   15        /* Connector Line Minimum Length */

#define NODE     0
#define RING     1
#define BRIDGE      2

#define NODETYPELEN 25

#define CU                      1
#define ALU                     2
#define REGISTERS               3
#define PSW                     4
#define IOPROCESSOR             5
#define MAINMEMORY              6
#define SECONDARYMEMORY         7
#define COMMANDPROCESSOR        8
#define OSKERNEL                9

#define EMPTY              -1

struct topologyData
{
   char nodeId[ NODEID ] ;
   char nodeAddress[ NODEADDRESS ] ;
} ;
extern struct topologyData Topology[] ;

struct ringData
{
   int ringTop ;          /* index of the top of the ring definition */
   int processed ;        /* Is this ring finished being processed */
   int centerX ;          /* Center of token ring point X */
   int centerY ;          /* Center of token ring point Y */
   double radians ;       /* Current Angle to use when drawing connectors */
   double currentRadians ;
   int lineLength ;       /* Length of the connector line */
   int LineX ;        /* Point X where connector ends */
   int LineY ;        /* Point Y where connector ends */
   int NodeX ;
   int NodeY ;
   int CircleX ;          /* Point X where connector starts */
   int CircleY ;          /* Point Y where connector starts */
   int BridgeX ;
   int BridgeY ;          /* Points where the bridge line extended to */
} ;

struct parentData
{
   double currentRadians ;
   int centerX ;
   int centerY ;
   int lineLength ;
   int BridgeX ;
   int BridgeY ;          /* Points where the bridge line extended to */

};


/*------------------------------------------------*/

/* Generic Menu Structure */
struct Menu
{
   char *choiceName ;  /* Name of menus items */
   void (*Func)() ;    /* Pointer to a function that will be called */
} ;

struct Buttons        /* Button defined on a screen */
```

```
{
   int X ;
   int Y ;
   char *string ;
   void (*Func)() ;
} ;

struct RegButtons      /* Registered Buttons      */
{
   int numberOfButtons ;
   struct Buttons ButtonPtr[ MAXNUMBEROFBUTTONS ] ;
} ;

#define PROCESSING 3
#define TOKEN      2
/* CU is already defined above */
struct NodePtr
{
   int x ;          /* Center X point      */
   int y ;          /* Center Y point      */
   int gender ;     /* Node, Ring, or Bridge   */
   int type ;       /* alu, cu, register, etc     */
   char address[NODEADDRESS ] ;      /* network address      */
   int ring ;       /* what ring is it a member of  */
   int nodeMarked ;    /* TRUE or FALSE indicating if the node is marked during a process tracking */
   int processState ;  /* CU, TOKEN, PROCESSING which state is the marked node in */
} ;

struct RegNodes        /* Registered Nodes        */
{
   int numberOfNodes ; /* Number of registered nodes   */
   int currentIndex ;  /* Index of the last nodes    */
   struct NodePtr NodeInfo[ MAXNODES+MAXRINGS ] ;
} ;

/* All Variables that keep track of the Monitors state should be wrapped into
   this structure and this structure should be passed to any function
   needing this information
*/
struct Monitor
{
   int startTraffic;      /* Flag indicating command line arg to start Trafic Monitor*/
   int Nodes ;         /* How many elements on the network includes delimiter */
   int Rings ;         /* How many total rings make up the network            */
   int Bridges ;          /* Number of bridges in current topology              */
   int currentMenuSet ;
   int menuItems ;
   int buttonSelected ;
   int trafficWindow ;    /* Flag indicating if window is already open */
   int trafficButIndex ;     /* Index of the registered dialog button*/
   int zoomWindow ;       /* Flag indicating if window is already open */
   int zoomButIndex ;     /* Index of the registered dialog button */
   int processWindow ;
   int processButIndex ;
   int tokenWindow ;      /* Flag indicating if window is already open */
   int tokenButIndex ;    /* Index of the registered dialog button*/
   int aboutButIndex ;    /* Index of the registered dialog button*/
   int aboutWindow ;
   int legendWindow ;
   int legendButIndex ;
   int bridgeWindow ;
   int bridgeButIndex ;
   int ringSelected ;     /* Ring selected by mouse press */
   char addSelected[NODEADDRESS] ;  /* Address of the node selected */
   int typeSelected ;     /* Type of node selected */

   int tokenNewView ;     /* Only for testing and debugging--used to repaint the token view window */

   int quit ;
} ;

struct BridgeData
{
   int BridgeID ;
} ;


struct BarData
{
   int percent ;
} ;
```