Interactive Computer Support for Remote Design Teams: A New Approach

by

David John Perreault

B.S.E.E., Boston University (1989)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering at the Massachusetts Institute of Technology

September 1991

© Massachusetts Institute of Technology, 1991 All rights reserved.

Signature of Author ____________________________

Department of Electrical Engineering and Computer Science

August 9, 1991

Certified by ____________________________

John G. Kassakian

Professor of Electrical Engineering Thesis Supervisor

Accepted by ____________________________

Campbell L. Searle

Chairman, Departmental Committee on Graduate Students
Interactive Computer Support
for Remote Design Teams:
A New Approach

by

David John Perreault

Submitted to the Department of Electrical Engineering and Computer Science on August 9, 1991 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering

Abstract

The desire to facilitate interaction among geographically distributed workgroups has underscored the need for better computer-mediated communications tools. Development of effective collaboration tools which permit remote, synchronous interaction is a challenging task, especially in applications where real-time graphics displays are necessary. In this work, a shell technique is investigated which gives single-user applications a multi-user interface by linking together multiple workstations. When the shell is in operation, the linked workstations operate identically, thus providing a uniform computer interface for a group of collaborators.

To investigate the technical requirements of a shell-based system, and to characterize the resulting multi-user interface, a prototype system has been developed. One basic design requirement is that the system allow users to interact in the environment of existing application programs, while remaining independent of any particular program. A second requirement is that the resulting system be able to function over low bandwidth communications links, regardless of the amount of graphics output or database changes produced by the system.

Thesis Supervisor: Dr. John G. Kassakian
Title: Professor of Electrical Engineering
Acknowledgements

I want to thank Professor John G. Kassakian for giving me the opportunity to work on this project, and for providing help and encouragement throughout its execution. I would also like to thank the "10 - 082" crowd for burning the midnight oil with me: it shows that I'm not the only crazy person around.

Heartfelt thanks are due my parents, who have provided the support and background which made everything possible. I also wish to thank my wife, Hideko, whose company makes everything a pleasure. I feel lucky to have such a wonderful family.

Finally, the author wishes to acknowledge the Leaders for Manufacturing Program for its support of this work.
# Table of Contents

Abstract ................................................................. 3

Acknowledgements .................................................. 5

List of Figures ....................................................... 9

List of Tables ....................................................... 10

Chapter 1: Introduction ............................................. 11
  1.1 Background ................................................... 13
  1.2 Approach ...................................................... 15

Chapter 2: The Execution Shell Technique ....................... 19
  2.1 The Nature Of the Problem ................................... 19
  2.2 Program Flow Control by Interrupt Interception .......... 22
     2.2.1 The Interrupt Structure of the PC Compatible ....... 22
     2.2.2 Interception of Operating System Calls ............. 23
  2.3 The Star Configuration: Master and Slaves ............... 27
  2.4 Memory Resident Programming Under DOS ................. 30
     2.4.1 Keeping a Program Resident .......................... 30
     2.4.2 The Reentrance Issue ................................. 31
     2.4.3 Accessing DOS During Interrupt Interception ...... 32
     2.4.4 Accessing DOS during background processing ...... 33
     2.4.5 Additional Multitasking Issues ..................... 34

Chapter 3: Communications Issues ............................... 39
  3.1 Data Reception and Transmission .......................... 39
     3.1.1 Data Reception ....................................... 39
     3.1.2 Data Transmission .................................... 41
  3.2 Communications Control and Data Transparency ........... 44
     3.2.1 Data Transparency .................................... 44
     3.2.2 Assessing Data Integrity ............................. 46
  3.3 Exchange Protocols ......................................... 47
     3.3.1 Record Format ....................................... 47
     3.3.2 Send and Acknowledge Techniques .................... 50
     3.3.3 A Synchronized Exchange Protocol ................... 53
     3.3.4 A Sliding Window Protocol ........................... 56

Chapter 4: Capturing and Distributing User Input ............. 63
  4.1 Keyboard Input on the IBM PC ............................. 63
     4.1.1 Operation of the Keyboard Subsystem ................. 63
     4.1.2 Capturing Keyboard Input ............................ 69
  4.2 Buffering and Distribution of Keyboard Input ............ 73
     4.2.1 Buffering Keyboard Input for Redistribution ....... 73
4.2.2 Synchronized Distribution of Keyboard Data ...... 76
4.2.3 Buffered Distribution of Keyboard Input .......... 77

4.3 Pointing Device Input on the IBM PC ................. 79
4.3.1 Operation of the Mouse Subsystem ............... 79

4.4 Control and Distribution of Mouse Input .............. 82
4.4.1 Handling of Mouse Input by Synchronization ...... 83
4.4.2 Handling of Mouse Input by Keystroke Conversion .. 87

Chapter 5: Controlling Parallel Execution .................. 95
5.1 The Execution Shell Technique Revisited .......... 95
5.1.1 Operation of an Execution Shell ............... 95
5.1.2 Assumptions of the Execution Shell Technique ... 96

5.2 Issues in implementing an execution shell system .... 100
5.2.1 Hooking Interrupt 21H .................. 100
5.2.2 Error Handling .......................... 103
5.2.3 Managing the environment table ............... 106
5.2.4 Backdoor calling of INT 21h ................ 107

5.3 Limitations of the Execution Shell Technique ........ 108
5.3.1 Limitations imposed by information retrieval methods .......... 108
5.3.2 Limitations imposed by the device driver system . 109

Chapter 6: Evaluation and Conclusion ..................... 113
6.1 Preliminary evaluation of the RSColl approach ........ 113
6.1.1 Evaluation Methodology ...................... 113
6.1.2 General Observations ........................ 115

6.2 Conclusions .................................. 119
6.2.1 Brief statement of the problem .................. 119
6.2.2 Methodology ............................... 120
6.2.3 Results .................................. 121

6.3 Suggestions for future research ...................... 121

References .................................... 125
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Operation of remote takeover software. Transmission of video data is the limiting factor for graphics based applications.</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Operation of an RSColl system based on parallel execution. Since each workstation generates its own video and database changes, the system can function with graphics-based applications.</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Interaction of computer system components</td>
<td>21</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Depiction of an INT call to DOS or BIOS</td>
<td>26</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Possible RSColl communications topologies</td>
<td>29</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Algorithm for invoking background processing</td>
<td>35</td>
</tr>
<tr>
<td>Figure 7</td>
<td>The prototype system Receive Interrupt Service Routine</td>
<td>42</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Flowchart for the polled transmission approach</td>
<td>43</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Record format used in the prototype system</td>
<td>45</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Code used for handling buffered RX data in the prototype system</td>
<td>49</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Transmission algorithm for Send/Acknowledge protocol</td>
<td>51</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Reception Algorithm for Send/Acknowledge protocol</td>
<td>52</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Synchronized information exchange for controlling parallel execution among workstations.</td>
<td>54</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Algorithm for a send/ack cycle in a synchronized exchange</td>
<td>55</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Master side of synchronize and exchange algorithm</td>
<td>57</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Slave side of synchronize and exchange algorithm</td>
<td>58</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Background transmission and reception algorithm</td>
<td>60</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Layout of the keyboard buffer and pointers in the IBM PC</td>
<td>66</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Processing of keystrokes through the keyboard buffer</td>
<td>67</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Keyboard Status Bytes on the IBM PC</td>
<td>68</td>
</tr>
<tr>
<td>Figure 21</td>
<td>General approach for trapping keyboard input on the PC</td>
<td>70</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Algorithm for trapping keyboard input on the IBM PC</td>
<td>74</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Use of a meta-driver to control mouse I/O</td>
<td>85</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Operation of the keystroke conversion method by polling</td>
<td>89</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Keystroke conversion code used in the prototype system</td>
<td>92</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Processing arrangement to avoid reentrance of execution shell code</td>
<td>102</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Managing the INT 21H processing arrangement</td>
<td>103</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Interactions between a program, DOS, a device driver, and the device.</td>
<td>111</td>
</tr>
</tbody>
</table>
List of Tables

Table 1  Interrupt calls used to invoke operating system services in the PC ................................................. 24
Table 2  Reserved bytes used for control in the prototype system ............................................................... 46
Table 3  Some of the mouse support functions available through the Microsoft mouse interface ................. 81
Table 4  DOS Critical Errors and standard error codes ................................................................. 106
Chapter 1: Introduction

"Recently it has become clear that users are frustrated by the inability to collaborate with their co-workers through the computer tools they have become accustomed to using. And therein lies a major motivation for a new development in information technology" [1]

The recent explosion in the availability of computers has heightened interest in computer-based tools which support cooperative work. Computer support of group collaboration may have the most utility in geographically distributed workgroups, where face-to-face communications are impractical [2]. It is hoped that the development of computer tools which support remote interaction will lead to improved efficiency and better cooperation among distributed workgroups.

Recent research in the area of Computer-Supported Cooperative Work (CSCW) has examined the needs of teams of collaborators working together. A study of information exchange within engineering design teams indicated that graphics capabilities and fully synchronous communications would probably be necessary for an engineering collaboration tool to be effective [3]. Similarly, researchers studying the workspace activity of design teams "observed two key features that the designers utilized in developing ideas: a) being able to readily try out representations of ideas in the public workspace, and b) having those representations gradually evolve into distinct artifacts, often through other participants building on and modifying them." [4]. In the same vein, the
importance of shared drawing surfaces for developing and representing ideas was established in [5]. What may be learned from these studies is that the ability to interact synchronously in a shared workspace and the ability to easily represent ideas are desirable attributes of a remote collaboration tool.

Single-user workspaces for representing and developing ideas are a fundamental part of many applications programs. A rich supply of single-user programs for tasks such as Computer-Aided Design (CAD), financial analysis, and document editing has developed in recent years. These programs provide graphical workspace environments optimized for developing and representing ideas in task-specific formats. However, because of their single-user orientation, these existing tools have not been fully exploited as part of the collaborative process.

The main objective of this work is to develop the basis for a 'shell' program, which gives existing single-user application programs a multi-user interface by linking multiple workstations together. When the shell is running, every station should show the same data, and any action taken on one station should happen at all of them. This is a variation of the WYSIWIS\(^1\) interface described in [6],[7].

A software shell which allows users to interact in the environment of a task-specific program has several benefits. First, it provides the shared workspace and synchronous interaction desired in a remote collaboration tool, with the shared workspace inherently optimized for a specific task. Second, it

\(^1\)WYSIWIS, "What You See Is What I See" - pronounced whizzy-whiz.
allows the enormous base of existing single-user application software to be applied in a collaborative setting, without redeveloping each program for a multi-user environment. Furthermore, a shell-based system permits users to maintain the same software standard, eliminating problems inherent in adopting different data formats. Also, users can take advantage of their knowledge of established single-user programs, instead of having to learn a new tool exclusively for collaboration. Finally, it has been pointed out that "task-oriented tools designed to facilitate the completion and integration of specific work products" are one class of tools necessary to support the needs of distributed work teams [8]. A shell program designed to be used with existing task-oriented software may help fill this niche in the CSCW arena more rapidly than otherwise possible.

1.1 Background

CSCW software can be classified by whether it is designed for local or distant use\textsuperscript{2}, and also by whether it is designed to be used synchronously or asynchronously [1],[9]. As can be seen from [1] and [10], much of the work in computer-supported cooperative work has been devoted to either wide-area asynchronous communication tools, or local-area synchronous tools. Some wide-area synchronous tools do exist, including a PC-based graphics system produced by Optel Communications, Inc., and a conferencing/editing tool by Group Technologies, Inc. (both of which allow operation over telephone links).

\textsuperscript{2}For purposes of this work, \textit{local} means within the span of a local area network, as opposed to meaning within one room, as used in [9].
However, construction of such systems requires ground-up development of each interactive environment, implying that there will be no immediate proliferation of task-oriented tools of this nature. Thus, a system which provides remote, synchronous support for a variety of applications represents a new direction in CSCW research.

A background search has been conducted for experimental or commercial software which could be used to implement a shell system, and only one related type of software has been found. This class of software, available from a number of manufacturers, is used for remote control of a PC system. As shown in Fig. 1, it functions by running the application program on one system, and using the remote system(s) as a dummy. All video information is shipped from the application system to the dummy, and keyboard input from both systems is used by the application system. This attempts to give the remote user the illusion of working on the local machine in tandem with the local user. However, as discussed in [11] and [12], currently available software of this type is inadequate for graphics based applications such as CAD\textsuperscript{3}. Furthermore, this approach is not generally feasible for graphics-based applications. This is because the amount of graphics data involved requires excessive time lags with the bandwidth available (over standard modem connections). Thus, no software suitable for constructing a shell has been found, and the currently existing approach to this type of problem is inadequate.

\textsuperscript{3}e.g. remote takeover software such as Carbon Copy Plus from Meridian Technology, or Remote 2 from Crosstalk Communications.
Figure 1  Operation of remote takeover software. Transmission of video data is the limiting factor for graphics based applications.

1.2 Approach

The approach taken for developing a prototype system, the RSColl (Remote Synchronous Collaboration) shell program, is geared towards the low end PC/AT environment under the DOS operating system. The PC type system has been selected due to its widespread use and low cost. Furthermore, the RSColl shell system is designed to operate over low bandwidth communications links such as modems, allowing remote interaction at a minimum of expense and hardware requirements. It should also be noted that the shell program works best in combination with direct audio links (e.g. telephone). The results of [2] and [8], as well as personal experience with the prototype system suggest the benefits of these additional links.
To handle the large amounts of graphics data necessary, the RSColl system uses a parallel processing approach. As shown in Fig. 2, each workstation runs the application software independently, with user input at each system delivered globally. All graphics and database changes are thus generated locally, reducing the required data transmissions to an acceptable level for low bandwidth communications. Using this approach, the prototype system runs quite effectively over point-to-point modem connections at 2400 baud.

The prototype RSColl shell is a Terminate and Stay Resident (TSR) program which performs two major tasks. First, the software distributes the user input at each workstation over the entire RSColl network. This common input connection is what allows the users to interact in a WYSIWIS manner. Input from both keyboards and pointing devices (such as mice) are handled by the RSColl system. As a second task, the software monitors the workstations to ensure that they are actually working in parallel. Such a mechanism is necessary to prevent the data on the different systems from diverging. This task is accomplished by regulating the environment in which the application program executes. For example, one technique used for doing this is observing and controlling software access to external devices and files, to ensure that each program receives identical information.

Thus, the RSColl system allows people in different locations to work synchronously and interactively via low bandwidth communications links. The system allows multiple users to interact within the framework of an existing single user program. While the concept of a shell program is simple and useful, the
Figure 2  Operation of an RSColl system based on parallel execution. Since each workstation generates its own video and database changes, the system can function with graphics-based applications.

realization of such a system is very challenging. One implementational challenge is artificially controlling the interface between existing programs, the operating system, and the computer hardware. This can be difficult, since neither the programs nor the operating system were designed to allow for this. A second implementational challenge is to manage the capture and distribution of user input in a robust manner, without degrading the performance of the application programs. Addressing these issues while remaining sensitive to the needs of the user dominates the design of an RSColl system.

The main text of this thesis deals with the techniques necessary to construct an RSColl shell program in the PC environment, and discusses the
limitations and tradeoffs involved. Chapters 2 and 3 contain basic information about the programming and communications requirements involved. Chapter 4 deals with capturing and distributing user input from the keyboard and pointing devices. Chapters 2 and 5 deal with the task of regulating parallel execution among the systems. Finally, chapter 6 contains a preliminary evaluation of the prototype system and the shell technique, and discusses some potential areas for future research.
Chapter 2: The Execution Shell Technique

In order to construct a common user interface utilizing a parallel processing approach, it is necessary to constrain copies of a program running on different computers to behave identically. This can be achieved by artificially controlling the manner in which the programs execute. A program makes decisions based on data it obtains from the environment around it. For instance, a program may test whether a key has been pressed, and base its actions upon the result of that test. By regulating the flow of data between the environment and the program, the execution of the program can be controlled.

2.1 The Nature Of the Problem

A program running on a PC compatible computer interacts with its environment in a manner depicted in Fig. 3a. An application program may obtain data from the operating system (DOS), the BIOS (Basic Input / Output System) or by interacting directly with the PC hardware. Provided that the application satisfies certain requirements\(^4\), such as acquiring all of its input through DOS or the BIOS, program execution can be controlled using an execution shell\(^5\). As depicted in Fig. 3b, an execution shell is software which intercepts and controls

\(^4\)These requirements will be treated in more detail in Chapter 5.

\(^5\)Some limitations on controlling program execution will also be examined in Chapter 5.
interaction between the application program and the computer system.

In order to control the operation of an application program, an execution shell selectively filters calls to the operating system, supplanting the operating system when necessary\(^6\). For example, if an application program requested the current date, the execution shell could either pass the request on to the operating system, or return its own date information. Furthermore, the execution shell has full access to the operating system when intercepting function calls. This allows the bulk of the work to be carried out by the operating system, while the execution shell retains control over the information flow. For example, if an application program requested the date, the shell could set the date, and then pass the original call on to the operating system. Alternately, the execution shell could return a date to the application after first retrieving information from the operating system. Thus, the operating system handles the processing of data, while the execution shell maintains control over the information returned to the application program.

---

\(^6\)The term *operating system* will be used to refer to both DOS and the BIOS when no further distinction is required.
Figure 3  Interaction of computer system components
(A) Under normal operation
(B) Under control of an execution shell
2.2 Program Flow Control by Interrupt Interception

The ability to trap program calls to the operating system is essential for the shell technique to function. It would also be desirable if some interactions between the BIOS and the hardware could be trapped. Fortunately, these interactions take place via the interrupt structure of the PC, which allows them to be trapped by hooking (redirecting) the appropriate interrupts\(^7\).

2.2.1 The Interrupt Structure of the PC Compatible

The bottom 1K of memory on the PC compatible is dedicated to the interrupt vector table, which is composed of 256 double word pointers. Each pointer references a location to which control is transferred when processing an interrupt. Interrupts are numbered 00 - FF hex. The interrupt number is an index to the pointer in the interrupt vector table. For example, If an Interrupt 21h were issued, control would transfer to the address stored in entry 21h of the interrupt vector table.

In the PC system, interrupts are classed as being hardware or software generated. External hardware interrupts, such as the keyboard input interrupt (Int 09h), are generated by the various PC peripherals, and transferred to the CPU via a Programmable Interrupt Controller\(^8\). Software interrupts are usually generated

---

\(^7\)As discussed in [13] and [14], there are valid methods of calling DOS without using the interrupt structure. See Chapter 5 for a more complete discussion of this.

\(^8\)With the exception of the non maskable interrupt (NMI). See [15] and [16] for more detail.
by the INT instruction, and are normally used for supervisor calls to the operating system\(^9\). When the CPU detects an interrupt condition, it first checks the interrupt enable flag. If this flag is set, the pending interrupt is then processed as follows: First, the flag register is pushed onto the stack, and the interrupt and trap flags are cleared (preventing further maskable interrupts from being processed). Then the return address is pushed onto the stack. Finally, the interrupt number is used to index the interrupt vector table, and control is transferred to the location specified there. When interrupt processing is complete, an IRET instruction can be used to restore the flags and transfer control to the return address.

2.2.2 Interception of Operating System Calls

Many of the interrupts in the PC compatible computer are dedicated to the BIOS, with an additional amount reserved for use by DOS. A subset of these interrupts are used to invoke the operating system services, as shown in Table 1. To access these services, registers are usually set up in a specified manner. Then a software interrupt (INT) is issued to invoke the proper interrupt service routine (ISR). Information is usually returned to the calling program in the registers or in specified data areas. It is this standard interface which permits calls to the operating system to be trapped by an execution shell.

\(^9\)Software interrupts can also be generated by the INTO, DIV and IDIV instructions. Furthermore, if the Trap Flag is enabled, most instructions will generate a software interrupt. See [17] for details.
<table>
<thead>
<tr>
<th>Interrupt</th>
<th>System</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>05h</td>
<td>BIOS</td>
<td>Print Screen</td>
</tr>
<tr>
<td>10h</td>
<td>BIOS</td>
<td>Video I/O</td>
</tr>
<tr>
<td>11h</td>
<td>BIOS</td>
<td>Equipment Determination</td>
</tr>
<tr>
<td>12h</td>
<td>BIOS</td>
<td>Memory Size Determination</td>
</tr>
<tr>
<td>13h</td>
<td>BIOS</td>
<td>Disk I/O</td>
</tr>
<tr>
<td>14h</td>
<td>BIOS</td>
<td>RS-232 Serial I/O</td>
</tr>
<tr>
<td>15h</td>
<td>BIOS</td>
<td>Cassette Tape I/O (PC Only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AT Extended Services (AT Only)</td>
</tr>
<tr>
<td>16h</td>
<td>BIOS</td>
<td>Keyboard I/O</td>
</tr>
<tr>
<td>17h</td>
<td>BIOS</td>
<td>Printer I/O</td>
</tr>
<tr>
<td>18h</td>
<td>BIOS</td>
<td>ROM BASIC Caller</td>
</tr>
<tr>
<td>19h</td>
<td>BIOS</td>
<td>Bootstrap Loader</td>
</tr>
<tr>
<td>1Ah</td>
<td>BIOS</td>
<td>Time-of-Day Clock Handler</td>
</tr>
<tr>
<td>20h</td>
<td>DOS</td>
<td>Terminate a Program</td>
</tr>
<tr>
<td>21h</td>
<td>DOS</td>
<td>DOS Function Caller</td>
</tr>
<tr>
<td>25h</td>
<td>DOS</td>
<td>Absolute Disk Read</td>
</tr>
<tr>
<td>26h</td>
<td>DOS</td>
<td>Absolute Disk Write</td>
</tr>
<tr>
<td>27h</td>
<td>DOS</td>
<td>Terminate But Stay Resident</td>
</tr>
</tbody>
</table>

Table 1  Interrupt calls used to invoke operating system services in the PC.
Hooking (redirecting) interrupts is an operation fully supported under DOS\textsuperscript{10}. Hooking an interrupt consists of saving the original interrupt vector (for later restoration and for chaining to the old service routine), then replacing the vector with a pointer to a new service routine. Program flow control can be achieved by redirecting the interrupts used for operating system calls through an execution shell, as shown in figure 4.

Thus, an execution shell can be implemented by taking advantage of the interrupt structure of the personal computer. The resulting control gained over program execution can be used to force copies of a program executing on multiple computers to behave identically. By controlling the individual workstations in a globally determined manner, the execution shells can be used to create a common user interface among remotely located computers.

\textsuperscript{10}Using Int 21 Fn 25 (Set Interrupt Vector) and Int 21 Fn 35 (Get Interrupt Vector).
Figure 4  Depiction of an INT call to DOS or BIOS
(a) Under normal operation
(b) Under control of an execution shell
2.3 The Star Configuration: Master and Slaves

Inducing copies of a program on remote computers to behave identically requires a rigorous control scheme. The execution shell operating at each workstation can control local program flow, but a global structure is needed to regulate parallel execution. The main functions of this structure are to coordinate the actions of the execution shells, and to uniformly distribute user input among the workstations.

The control scheme must be designed such that every computer receives consistent control and input information. For example, if keys are struck simultaneously on two different workstations, the keystroke data must be distributed to every workstation in the same order. To meet this requirement, the RSCol system is set up in a master / slave fashion. The execution shell at each workstation traps user input and transmits it to the master workstation. The master execution shell then serializes the input into one stream, and distributes it back to the slaves\(^{11}\). Program flow control is handled in a similar manner. The slave execution shells each send status information to the master. The master then determines the proper control actions to be taken, and returns this information to the slaves. The slave execution shells can then regulate program flow in a globally consistent manner\(^{12}\). Thus, use of a master / slave control scheme provides the consistency and structure necessary for a functional system.

\(^{11}\)The manner in which data distribution is handled greatly influences the functionality and efficiency of the system. See Chapter 4 for details.

\(^{12}\)See Chapter 5 for specifics on global execution control.
The organization of the communications links is another factor that influences the effectiveness of the system. Because the system is intended to be implemented using point to point modem connections, the alternatives are either a daisy chain (or ring) topology, or a star topology (Fig. 5). The star configuration is considered better suited for this application for two reasons. First, the star configuration allows direct communication between the master and any slave, whereas the daisy chain configuration requires messages to be buffered and retransmitted down the chain. Thus, the star configuration is inherently more efficient than the daisy chain configuration in terms of message propagation time and the amount of local buffering space needed at each computer. Second, failure of a single communications link will cause a daisy chained network to cease functioning entirely, whereas a star configured network will be unaffected except for the loss of one slave. Thus, a star topology is the best network configuration for this application in terms of fault tolerance and efficiency.

To implement the distributed system proposed, a master / slave control scheme has been selected. The execution shells operating at each slave workstation communicate with a master execution shell, which provides the control information needed for globally identical program operation. The star configuration is considered to be the most effective topology for implementing the communications links between workstations.
Figure 5  Possible RSColl communications topologies
  (A) Daisy chain topology
  (B) Star topology
2.4 Memory Resident Programming Under DOS

The ability to multitask the operating system is essential for realizing the proposed system. That is, the application program and the execution shell need to coexist in memory and use operating system resources in tandem. The MS-DOS operating system, however, was originally designed to be single-tasking. Fortunately, the designers of DOS made provisions for a limited form of multitasking called Terminate and Stay Resident (TSR) programming\textsuperscript{13}. These provisions were primarily for Microsoft's internal use, and many utilities needed to effectively multitask the operating system remain undocumented\textsuperscript{14}. Because MS-DOS is ill suited for multitasking, memory resident programming remains one of the most challenging areas of PC programming.

2.4.1 Keeping a Program Resident

The first requirement of multitasking is the ability to keep multiple programs (processes) in memory simultaneously. Normally, when a program has finished executing, it issues a termination call to DOS. DOS releases all memory allocated to the program, then returns control to the process which spawned the terminating program (usually the DOS shell, command.com.) To allow a program to terminate but remain resident in memory, DOS provides INT 21h function 31h\textsuperscript{15}. To use this service, a program first hooks a set of interrupts

\textsuperscript{13}For detailed information on TSR programming, see [18] and [19].

\textsuperscript{14}See [19] for a definitive treatment of undocumented DOS programming.

\textsuperscript{15}DOS also provides an older service, INT 27h, which performs a similar but more limited function.
which activate service routines belonging to the program. It then releases any memory it will not need. Finally, it issues the call to terminate but stay resident, upon which DOS returns control to the spawning process. Since the memory in which the terminated program resides is still allocated, it is secure from being overwritten by subsequent programs. Thereafter, the terminated program can be reactivated by the interrupts which it hooked before terminating.

2.4.2 The Reentrance Issue

When the MS-DOS operating system is in use, no other process may issue a DOS call, because operating system data may be overwritten by the second call. The condition in which a block of code is invoked by a process while it is in use by another process is called reentrance. The fundamental difficulty in multitasking under MS-DOS is that DOS is not designed to be reentrant.

To be precise, DOS is partially reentrant. When the operating system is invoked, it switches to one of its own stacks. As discussed in [19], DOS will use an I/O stack for function calls 00 - 0Ch, and a Disk stack for most other functions. Under certain conditions such as Critical Errors, when DOS needs to reenter itself, it may also switch to an Auxiliary stack. If DOS is using one of its internal stacks, and a call is made to a DOS function that uses the same internal stack, then the original data will be overwritten, possibly causing a system failure. Avoiding this situation is the heart of memory resident

\[\text{16} \text{A few DOS functions actually use the calling program's stack, and are thus fully reentrant. See [19] for details.}\]

\[\text{17} \text{However, DOS functions may be called which use a different stack than the one in current use, hence the term partially reentrant.}\]
programming.

When only one task is executing, there is no problem with reentrance, since the task can only make one DOS call at a time. However, if a memory resident program is activated by an interrupt while DOS is currently active, a reentrant condition will occur if the TSR issues any calls to DOS. Since any reasonably sophisticated memory resident program will need to access DOS, techniques need to be implemented which allow a TSR program to use DOS, while not causing a reentrant condition.

2.4.3 Accessing DOS During Interrupt Interception

One assumption that was made when developing the execution shell technique was that the execution shell would have access to the operating system when intercepting system calls. On the surface, it would seem that there are no reentry problems in doing this, since calls are being intercepted before the operating system is invoked. However, it must be remembered that DOS is partially reentrant, and can (and does!) reenter itself. Thus, during the processing of an INT 21h call, DOS could issue its own INT 21h, so long as it calls a function which uses a different internal stack. In order to prevent reentrance problems when the execution shell accesses DOS, the execution shell must be constrained to only access DOS functions which use the same stack as the call being intercepted\(^\text{18}\). With this restriction, DOS may be freely accessed during interrupt interception.

\(^{18}\)The exception to this condition is when it is known that DOS is not already in use.
2.4.4 Accessing DOS during background processing

While other possibilities exist [19], the primary technique for avoiding reentrance in memory resident programming is to defer accessing the operating system until it is not in use. The proposed system requires access to the operating system when handling background processing such as logging errors to the disk, or uploading files to another computer. To handle these requirements, interrupts which occur periodically can be set up to repeatedly test whether the operating system is busy. When the operating system is free, background processing can be carried out. To implement this scheme, a test is needed to determine whether or not DOS is busy. Fortunately, the designers of DOS provided a method for determining whether or not DOS is in use. As discussed in [18] and [19], undocumented DOS function 34h returns the address of a one byte flag known as the "InDOS" flag. When the InDOS flag is nonzero, code within DOS is currently in use, and it is unsafe to access DOS\(^9\). Thus, examination of the InDOS flag can be used to determine when it is safe to perform background processing.

In order to process data in the background, the execution shell needs to chain into a periodic interrupt. Interrupt 1Ch, the BIOS timer tick interrupt, is a hardware interrupt that occurs 18.2 times per second. To process data in the background, the execution shell can hook this interrupt. As shown in Fig. 6, the interrupt service routine should first execute the old ISR, to allow execution of

\(^9\)There is also a critical error flag which may be tested to determine if DOS has switched to the auxiliary stack due to a critical error. See [19] for details.
other programs that have chained into this interrupt. In order to prevent reentrance of the background processing code, the service routine should check if the background code is already executing. If not, and if DOS is not in use, the background code can be executed, after which control should be returned to the main process by use of an IRET. Using this technique, a background process can be multitasked with the main process, while avoiding reentrance problems.

Invoking background processing involves waiting until the InDOS flag indicates that DOS is not busy. However, because command.com uses INT 21 fn 0Ah (Buffered Keyboard Input) to retrieve user input, the InDOS flag will indicate that DOS is in constant use whenever the computer is sitting idle at the DOS prompt. To work around this problem, the designers of DOS included an undocumented "keyboard busy" interrupt. Interrupt 28h is continuously invoked by DOS when it is idling at the DOS prompt, and is set up to allow background tasks to reenter DOS while the computer is waiting for input. Background processing is achieved by chaining into this interrupt in the same manner as Int 1Ch, with the exception that processing should be invoked when the InDOS flag is equal to 1 (DOS Busy) instead of 0. Thus, by chaining into interrupts 1Ch and 28h, background processing requiring access to DOS can be carried out safely in the DOS environment.

2.4.5 Additional Multitasking Issues

An additional issue which arises when multitasking the DOS environment is the need for stack switching. When a background process is triggered by an
Figure 6  Algorithm for invoking background processing.
interrupt, the stack pointer references the stack of the foreground process. Since
the amount of space available on the foreground stack is generally unknown, it
is dangerous for another task to use that stack. To handle this, a stack switching
scheme must be implemented which saves the foreground stack pointer and
switches to a background stack during background processing. Since most higher
level languages assume that a program is not interfering with the stack, care must
be taken when dealing with data (such as local variables) which the higher level
language assumes is stored on the stack. Furthermore, if multiple background
tasks or ISRs use the same stack switching code, care must be taken to account
for reentrance of the stack switch [18]. Thus, stack switching is a delicate issue
which must be addressed when multitasking in the DOS environment.

Another issue which crops up when multitasking is the handling of the
Program Segment Prefix (PSP) and process ID. The PSP is a data structure that
accompanies each program loaded in memory. It is used both by the operating
system and by the program to store information about the program. DOS
identifies each loaded program with a process ID, which is actually the segment
address of the PSP for that program. Since DOS is single tasking, it only
recognizes one active process at a time. When handling tasks such as file I/O,
the operating system will use the PSP for the current active process. Thus, to
perform operations using its own PSP, a background process must install itself
as the process which DOS recognizes as active. To do this, two undocumented

---

20 See [18], [20] and [21] for details on the Program Segment Prefix. Information on the Process ID can be found in [19].
DOS functions are required\textsuperscript{21}. The first function, Get PSP Segment (Int 21h fn 51h), returns the process ID of the current active process. The second function, Set PSP Segment (Int 21h fn 50h), is used to change which process is active. Thus, to do operations using its own PSP, a background process must first save the current active process ID, and switch the active process ID. It can then perform the operations, and restore the original active process ID.

Another potential problem a background process must deal with is the handling of extended error information (handling of critical errors and user interrupts is deferred to Chapter 5). When a DOS function call is made, DOS sets up extended error information, which may be retrieved using DOS function 59h. Handling this information properly is important for system reliability. Consider the situation in which a background process is activated between the time a foreground process makes a DOS call and the time it retrieves the extended error data. If the background processing causes this information to change, the foreground process will retrieve erroneous data. To prevent this from happening, a background process should save the extended error information that is valid when it is activated, and restore it when background processing is complete. To do this (under DOS vers 3.1+), undocumented DOS function 5D0Ah (Set Extended Error Info.) can be used. Thus, proper handling of extended error information can increase system reliability in a multitasking scheme.

\textsuperscript{21}[18] actually uses a method which only requires one undocumented function, but it is far less efficient.
It can be seen that the requirements for multitasking in the DOS environment are stringent and somewhat esoteric. This is largely due to the fact that DOS was not originally designed for true multitasking. For a fuller treatment of the subject, the reader is referred to [18], [19].

The RSCol system operates within the constraints of memory resident programming under DOS. When the RSCol system program is executed at a workstation, it sets up the communications system, hooks a set of relevant interrupts, then terminates and stays resident. Once resident in memory, the RSCol program consists of an execution shell (which intercepts system calls), and a background process (which is chained into interrupts 28h and 1Ch, as previously described). Thereafter, the system follows the described rules for multitasking the DOS operating system. Thus, it can be seen that the structure of the operating system largely dictates the form of the RSCol system.
Chapter 3: Communications Issues

In order to implement the distributed system, a robust communications scheme is essential. The communications stream between two workstations consists largely of user input data (such as keystrokes), and control information (to regulate parallel execution). If mistransmission of a single piece of input data is not detected, the operation of the programs on the two computers could unrecoverably diverge. However, even a temporary communications blackout can be dealt with, as long as it is detected. Thus, for this application, robustness implies an immunity to undetected transmission errors, as opposed to a need for totally undisturbed communications. This chapter deals with the issues involved in constructing a robust communications interface suitable for a PC based RSCol system.

3.1 Data Reception and Transmission

The RSCol system must operate over point-to-point modem connections at relatively low bandwidths. Specifically, communications at each workstation should be handled through one (or more) serial ports, at speeds of approximately 1200 to 2400 baud. This implies that most error checking and all error correction must be handled in software.

3.1.1 Data Reception

Data reception through a serial port can be accomplished using either an
interrupt driven or a polled approach. In the interrupt driven approach, the serial port is configured to generate a hardware interrupt whenever new data arrives. An interrupt service routine (ISR) can then be used to retrieve and process the incoming data. In the polling approach, a program periodically tests whether data has been received at the serial port, and processes it as it arrives. Because the execution shell at each workstation is time-sharing the computer, data reception at each serial port must be interrupt driven instead of polled. Thus, the setup of the communications interface at each workstation is dictated by the nature of the execution shell technique.

To allow maximum flexibility, and to provide a mechanism for error correction, incoming data is coded into logical records. The receive ISR can be used to either decode the incoming data, or merely buffer it for later processing. Experimenting with the prototype system revealed that buffering the data for later processing was the better option for two reasons. First, because the serial receive interrupt may occur while DOS is in use, the serial receive ISR cannot access DOS safely. Thus, if decoding is performed in the ISR, it cannot contain any operations (such as error logging to the disk) which require access to the operating system. Second, processing time for a received byte can vary greatly, and may be long enough to cause overflow errors if performed in the ISR. For example, if an incoming byte is the end of a record, an entire block of information may need processing. While data is being processed by the ISR, additional received bytes cannot be handled. If the serial port cannot internally buffer enough incoming characters, data will be lost, causing a loss of
communication efficiency. Thus, an interrupt service routine is used to buffer the incoming data for later processing, as shown in Fig. 7.

Incoming data is buffered by the serial receive ISR. In cases where the execution shell is waiting for data, the routine for processing the serial receive data may be called directly. Otherwise, a background process periodically checks the buffer, decodes and then processes the incoming data for use. Because execution of the background process can be deferred until the operating system is not in use, the background process has full access to system resources (for error logging, etc.).

3.1.2 Data Transmission

Transmission of data may either be accomplished by polling or by use of a transmit ISR. To use a transmit ISR, the serial port is configured to generate an interrupt whenever it is ready to accept a new byte for transmission. The transmit interrupt service routine merely transfers a byte from a transmit buffer to the serial port. To send a block of data, a process need only place it in the transmit buffer, and allow the transmit ISR to handle it. A polling approach merely consists of repeatedly testing if the serial port is ready to accept a byte for transmission, and sending a byte every time it is. While the interrupt driven method is more efficient, a polled approach was used in the prototype for simplicity, as shown in Fig. 8.
Procedure MyCommISR; (Interrupt - params at top of unit)
Begin
CLI;
BeginInt; (* switch stack *)
STI;
LSR_value := $0E and port[COMselect or $2fd]; (* check for error *)
CLI; (* interrupts off to change buffer *)
RX_Buff[RX_tail_ptr] := port[COMSelect or $2f8];
(* on error, replace read char *)
(* with error marker char.*)
if LSR_value <> 0 then
begin
RX_Buff[RX_tail_ptr] := $EO;
if LSR_Value and $08 <> 0 then inc(Error_Count[13]); (* framing error *)
if LSR_Value and $04 <> 0 then inc(Error_Count[18]); (* parity error *)
if LSR_Value and $02 <> 0 then inc(Error_Count[15]); (* overrun error *)
New_Errs_Occurred := true;
end;
IF RX_Tail_ptr < Buff_end then inc(RX_Tail_Ptr)
else RX_Tail_Ptr := 0;
If RX_Tail_ptr = RX_Head_Ptr then
begin
inc(Error_Count[16]); (* RX buff ovflow *)
New_Errs_Occurred := true;
end;
EndInt; (* switch stack back *)
port[$20] := $20;  (* send End of Interrupt to 8259 *)
end;

Figure 7  The prototype system Receive Interrupt Service Routine
(a) Flowchart of Receive Interrupt Service Routine.
(b) Receive ISR used in the prototype system
Figure 8    Flowchart for the polled transmission approach

If two or more processes operating on a workstation need to transmit data over the same link, it is necessary to serialize access to the communications adapter. For instance, if both the background process and the execution shell need to send data to another workstation, it is critical that their transmissions do not interfere with each other. In an interrupt driven transmission it is sufficient
to disable the interrupts when placing a record in the transmit queue. This prevents any interference from another process when adding data to the queue. In a polled system, a flag can be employed to indicate whether the communications adapter is in use. To send data, a process sets the "in use" flag, transmits a record, then releases the flag. If no process attempts to transmit data when the "in use" flag is set by another process, interference cannot occur. The prototype system uses a variation of this approach, in which the background process will not execute if the execution shell is active. Because the background process is never active when the execution shell is transmitting data, no interference can occur.

3.2 Communications Control and Data Transparency

3.2.1 Data Transparency

In order to reliably exchange data, information is typically sent in formatted records, such as the kind shown in Fig. 9. Control characters are used to mark different portions of a record and identify its contents, thus permitting information decoding and error correction. To implement this structure, some bit patterns must be reserved for use as control characters. Given that any character may need to be sent as data, the problem of sending data that could match the control characters arises. That is, a mechanism is needed to make the data transparent to the control scheme.

The transparency method chosen for use in the prototype system is well known [22]. Any characters that fall within the band reserved for control
characters are coded into a pair of special control characters. The processing
software recognizes these special characters, and decodes them into the original
single character. In the case of the prototype system, byte values of 80-FF (hex)
are reserved for control information (Table 2). To send a data byte in this range,
say 89 (hex), the byte is recoded to the pair F8 F9. The decoding software
recognizes any character with a high nibble of F hex as part of a pair, and does
the proper decoding. This technique thus allows complete data transparency.

1: **Record Type:** This is a control code in [80h,AFh] for normal records, and
[B0h,DFh] for acknowledge records. This code acts as both a start of record
marker and indicator of record contents.

2: **Record Number:** This number is in the range [00h,7Fh], and is used to
identify the record for error correction and acknowledgement purposes (see
below).

3: **Data Field:** The contents of this field are record type dependent.
The field usually contains a length marker or end of field indicator in addition to
any data it carries.

4: **Checksum:** This is used for error detection within a record. It
consists of all bytes of the record (except the end marker) XORed together, and
ANDed with 7Fh (for transparency).

5: **End of Record:** This is a control code (EFh) which marks the end of the
record.

**Figure 9** Record format used in the prototype system.
### 3.2.2 Assessing Data Integrity

To ensure reliable communications, techniques are also needed to assess the integrity of received data. Because the system is intended to operate over serial links, the only error checking provided in hardware is for framing, parity and overrun errors. This error checking is provided by the asynchronous communications adapter [15], and will detect bytes lost due to faulty reception (framing, overrun), as well as single-bit errors in received bytes (parity). The receive ISR tests for these error conditions, and marks the data with control characters if errors are detected. This allows the decoding software to reject any erroneous data, and log error information to the disk.

<table>
<thead>
<tr>
<th>Byte Values</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$80 - $AF</td>
<td>record type / start of record indicator</td>
</tr>
<tr>
<td>$B0 - $DF</td>
<td>acknowledge record type / start of record</td>
</tr>
<tr>
<td>$E0</td>
<td>Framing / Parity / Overrun Error Marker</td>
</tr>
<tr>
<td>$E1 - $EF</td>
<td>reserved for future use</td>
</tr>
<tr>
<td>$F0 - $FF</td>
<td>coded byte values for data transparency</td>
</tr>
</tbody>
</table>

**Table 2**  
Reserved bytes used for control in the prototype system.
The byte-level error checking provided by the communications adapter is considered insufficient for implementing a robust communications interface, because it can only detect single-bit errors on a byte by byte level. Further error detection can be provided in software by including checksum information as part of the record format. Cyclic Redundancy Check (CRC) techniques for different types of error detection are well known [23]. For the prototype system, a simple XOR (exclusive OR) checksum is used. The bytes in a received record are XORed together, and compared with a checksum at the end of the record. If the values do not match, an error has occurred somewhere in the record. This technique provides record-level error detection which complements the byte-level error detection provided in hardware. The combination of hardware and software based error checks enables the decoding software to reliably identify transmission errors.

3.3 Exchange Protocols

3.3.1 Record Format

Structured records of the format shown in Fig. 9 are used to implement the communications interface. Control characters are used to mark different portions of the record, as well as to identify record contents. In the prototype system, byte values 80h-FFh are reserved for control information, the uses of which are shown in Table 2.

As previously discussed, incoming data is buffered by the serial receive ISR. Handling of the buffered data can either be initiated by a background
process, or directly by the execution shell. When the buffer handling software is invoked, it decodes all of the characters currently in the receive buffer and then returns. Decoding the data involves framing the characters into individual records, then logically processing the record contents. The buffer handling software interpretes the buffered characters sequentially as follows: If a start-of-record character is found, the software starts storing characters at the beginning of a record buffer, and resets the checksum count and error checking. Otherwise, characters are stored in the record buffer and the checksum information is updated. Decoding for the transparency scheme and recording of detected reception errors is also performed at this point. When an end of record character is encountered, the calculated checksum is compared to the expected value. If the checksum is correct, and no reception errors have been detected, the software to process logical records is invoked. Because the handling software returns immediately after processing all currently buffered characters, data can be processed either periodically or in a continuous manner. The decoding routine used by the prototype system is shown in Fig. 10 as an example.
Procedure Handle_RX_Buffer;
(* This routine scans the rx buffer, and creates records *)
(* in a buffer which HANDLE_RX_REC in the programs can *)
(* handle. If a good record (checksum & no f/p/o errs) *)
(* is found, it calls the record handler. *)
(* ******************************************************)
Begin
while not (rx_head_ptr = Rx_tail_ptr) do
  begin
    CLI; (* ints off to modify buffer ptr *)
    tmp_comm_byte := rx_buff[rx_head_ptr];
    if rx_head_ptr < buff_end then inc(rx_head_ptr)
    else rx_head_ptr := 0;
    STI; (* ints back on *)
  case tmp_comm_byte of
  $00..$7F : begin (* accept the character *)
    if RX_rec_pos > 254 then error(12);
    inc(RX_rec_pos);
    RX_record[RX_rec_pos] := tmp_comm_byte;
    RX_Checksum := RX_Checksum xor tmp_Comm_BYTE;
    end;
  $80..$DF : begin (* new record start *)
    if RX_rec_pos <> 0 then error(1);
    (* if not already 0, an error has occurred *)
    RX_rec_pos := 1;
    RX_record[RX_rec_pos] := tmp_comm_byte;
    if tmp_comm_flag then error(2);
    (* if not already F, an error has occurred *)
    tmp_Comm_flag := false; (* no 1/2 coded byte recvd *)
    RX_Checksum := tmp_Comm_BYTE; (* start checksum *)
    fpc_err_found := false; (* no fram/par/ovrun errs *)
    (* found in record yet. *)
    end;
  $E0 : begin (* fram/par/ovrun error marker *)
    fpo_err_found := true;
    Error(25);
    end;
  $EF : begin (* end of record... handle it *)
    RX_record[RX_rec_pos+1] := tmp_comm_byte; (* for test *)
    RX_Checksum := RX_Checksum and $7f; (* fix rx checksum *)
    if RX_Checksum <> 0 (* XOR of bytes and checksum must = 0 *)
      then error(3) (* checksum is bad *)
    else if fpo_err_Found then () (* fram/par/ovrun occurred *)
      else callrout(Rx_Rec_handler_ptr);
    RX_rec_pos := 0; (* set for next record *)
    (* checksum is good, no errs *)
    end;
  $F0..$FF : begin (* 1/2 of a coded byte *)
    if tmp_Comm_flag then begin
      RX_Checksum := RX_Checksum xor tmp_Comm_BYTE;
      inc(RX_rec_pos);
      RX_record[RX_rec_pos] := (old_half.Byte or byte(word(tmp_Comm_BYTE) shl 4) and $00f0));
      tmp_Comm_Flag := false;
      end
    else begin
      RX_Checksum := RX_Checksum xor tmp_Comm_BYTE;
      old_half.byte := tmp_Comm_byte and $0f;
      tmp_Comm_Flag := true;
      end;
    end;
  else error(4);
  end; (* case *)
end; (* while *)
end;

Figure 10  Code used for handling buffered RX data in the prototype system.
3.3.2 Send and Acknowledge Techniques

A mechanism for correcting transmission errors is necessary if communications are to be reliable. The prototype system uses a standard "send and acknowledge" protocol to achieve this, as shown in Figures 11, 12. If an error is detected in a received record, the information is discarded, and no acknowledge is returned. If a (non acknowledge) record is processed, and no errors are detected, an acknowledge record is returned to the workstation which sent the record. The acknowledge record indicates the record type and number which was received. Furthermore, the data from a correctly received record is only kept if the record number indicates that the data has not been previously processed. This prevents records which have been retransmitted from being processed twice. Thus, the data reception algorithm is designed to accept only data which has been correctly received and has not been previously accepted.

To complement this reception scheme, the transmission code is designed to periodically retransmit a record until it is acknowledged. After a record is transmitted, the process which sent the record must test whether the acknowledge has been received before sending another record of the same type. If not, it retries the transmission. After a suitable number of retries, the communications link is considered inoperative, and communications are shut down. Furthermore, because individual record types are treated separately, this communications scheme allows multiple processes to transmit data independently, as long as separate record types are used by each process. In the prototype system, the execution shell and the background process each transmit data independently.
Figure 11  Transmission algorithm for Send/Acknowledge protocol
Figure 12  Reception Algorithm for Send/Acknowledge protocol
across a single communications link, without interfering with the send and acknowledge handshake of the other. Structuring the handshaking in this manner has thus been found to be highly effective for this type of task.

3.3.3 A Synchronized Exchange Protocol

To regulate parallel execution among machines, the execution shells operating on different workstations must communicate. The communication between a master and a slave machine involves verifying that both computers are at the same state, and exchanging information for regulating parallel execution. For instance, if a master and a slave machine have reached the point of a file open request (by an application package), the two execution shells would need to communicate. After synchronizing with the master, the slave may send information to the master about the type of request it has received (open a file) along with information about the request (what file, a file checksum, etc.). The master would respond with information on how to proceed with the request (allow or reject the request). Thus, this form of communications involves synchronizing the two workstations (at the point of a specific system call), then exchanging data, as shown in Fig. 13. To perform this type of operation, a synchronized exchange protocol has been developed.

The synchronized protocol involves several send and acknowledge cycles. For this protocol, a single send and acknowledge cycle is handled in a timing loop, as shown in Fig. 14. Including receive buffer handling and acknowledge transmissions as part of the timing loop prevents a lockout condition from
Figure 13  Synchronized information exchange for controlling parallel execution among workstations.

occurring, in which both computers are waiting for an acknowledge from the other. A send and acknowledge handled in this manner will hereafter be called a record delivery.

The main issue that arises when performing this type of transaction is how to allow the machines to synchronize without locking the system if communications fail. Consider an interaction between the master and one slave. If the master reaches the synchronization point first, and requests data from the slave, it must then wait for the slave to send data. The protocol must be able to handle a situation in which the slave cannot respond because of a communications failure. To prevent the system from locking when communications fail, dummy
Figure 14  Algorithm for a send/ack cycle in a synchronized exchange.
records can be delivered by the computer which is waiting for a response, to ensure that the communications link is still active. The resulting protocol, shown in Figures 15 and 16, guarantees that a failed communications link will be detected during a wait period. In the event that the communications link is still functional, but the execution shell at one end is not, an absolute timeout can also be provided. This protocol has been found to be very effective for handling execution shell communications.

3.3.4 A Sliding Window Protocol

Background process communications are of a different nature than execution shell communications. First, background communications generally consist of unidirectional data transfers instead of synchronized data exchanges. Furthermore, a background process can generally time share the system with a foreground process, allowing a more efficient use of system resources than a synchronized protocol provides.

A sliding window protocol has been devised which is suited to the nature of background process communications. This protocol takes advantage of the periodic operation of a background process to remove the timing loops inherent in a synchronized protocol. It does this by transmitting a record at one timer tick, and testing for the appropriate acknowledge at a later tick. Data reception and acknowledgement can also be accomplished in the background, as previously discussed. This structure permits highly efficient data transfers, at the expense of not being able to exchange data in a synchronized manner.
Figure 15  Master side of synchronize and exchange algorithm
Figure 16  Slave side of synchronize and exchange algorithm
The background transmission and reception algorithm used in the prototype system is diagrammed in Fig. 17. This background process allows transmission and reception of data, as well as providing error logging services for the execution shell. As discussed in chapter 2, this process is invoked during timer ticks or keyboard busy loops, when DOS and the communications adapter are not in use.
Figure 17  Background transmission and reception algorithm
Chapter 4: Capturing and Distributing User Input

In order to cause copies of a program executing on multiple workstations to behave identically, it is necessary to supply them with the same user input. Consider the case of a CAD package running on two different workstations, in which each program manipulates data as the user enters keystrokes and moves the mouse cursor. In order for machines to behave identically, it must appear to the programs as if they are both receiving the same user input from the keyboard and mouse. Furthermore, the mouse and keyboard data must be accepted by each program in precisely the same order. For instance, a keystroke followed by a mouse movement can have quite a different effect than a mouse movement followed by a keystroke. Thus, in order to implement an RSColl system, it is necessary to provide a mechanism by which all user input at each workstation can be trapped, then delivered to every workstation in an identical manner.

This chapter deals with the construction of a mechanism for uniformly trapping and distributing user input. First, the PC keyboard system will be reviewed, and a method for trapping keyboard input will be discussed. Two methods for distributing keyboard input among remote workstations will then be presented. Second, the functioning of the mouse system will be discussed and two methods will be presented for trapping and distributing mouse input.

4.1 Keyboard Input on the IBM PC

4.1.1 Operation of the Keyboard Subsystem

In the time between when a key is struck and a program receives the key,
a whole chain of events occurs\textsuperscript{22}. Individual keypresses and releases are detected by a microcontroller embedded in the keyboard. When a legitimate key press or release is detected, or a key has been held down long enough for a typematic response to be generated, the microcontroller determines what key has been pressed, held down, or released, and generates a one byte \textit{scan code} to represent the key status change. A unique scan code is generated for each individual key on the keyboard, with the high bit turned on for a key make (new key press or typematic repeat), and turned off for a key break (release)\textsuperscript{23}. The generated scan code is transmitted to the PC motherboard, and ends up at PORT A (address 60H) on the programmable peripheral interface (PPI), where it can be accessed by the CPU. When the scan code is received at the motherboard, an interrupt request (IRQ1) is sent to the programmable interrupt controller. When the interrupt controller processes the request, it will generate a hardware interrupt 9H, which is the BIOS keyboard handler interrupt\textsuperscript{24}.

The function of the BIOS INT 9H handler is to interpret scan codes received from the keyboard, and buffer the resulting information in a format retrievable by an application program (through BIOS keyboard I/O interrupt 16H). Because the scan code information that INT 9h retrieves includes

\textsuperscript{22}For an detailed examination of the PC keyboard and its Interrupt Service Routines, see [24].

\textsuperscript{23}There are certain exceptions to this in the 101 key enhanced keyboard. See [24] for details.

\textsuperscript{24}For more detailed information on the operation of the programmable interrupt controller in the PC compatible computer, see [16].
information about both key presses and releases, it must convert this information into ASCII characters which may be used by an application program. Furthermore, it must store information such as which modification keys (Shift, Ctrl, Alt, etc.) are currently pressed, and which lock keys (Caps, Num, Scroll, etc.) are turned on. This information is used by INT 9h to determine what ASCII code to store in the keyboard buffer for a specific key (i.e. 'a', 'A', Ctrl-A, etc.), and can also be retrieved by an application program (through INT 16H).

Interrupt 9H stores the information it retrieves from the scan codes in a keyboard buffer and a pair of keyboard status bytes. The keyboard buffer is a 32 byte circular buffer located at absolute addresses $0040:001E$ to $0040:003D$ in the BIOS data area, as shown in Fig. 18. A two byte head pointer into the circular buffer is located at $0040:001A$, and a two byte tail pointer is located at $0040:001C$. INT 9h stores two bytes in the keyboard buffer for every character it processes. The first byte is the ASCII character for the key, and the second byte is the scan code for that key. (For some special keystroke combinations, a zero is stored for the ASCII code, and a special scan code is stored in place of the one received from the keyboard controller. See [24] for details). When the keyboard buffer is empty, the head and tail pointers point to the same location. When a character is added to the buffer by INT 9h, the ASCII code and scan

---

25[25] contains a good discussion of the layout of the keyboard buffer and keyboard status bytes. [24] contains an in depth description of the operation of INT 9H. A commented copy of the source code for the IBM PC BIOS INT 9H can be found in [15].

26The size and location of this buffer can be changed to a limited extent. See [24] for details.
code are stored in the word addressed by the tail pointer, and the tail pointer is advanced around the circular buffer. When a character is read by INT 16h, the character is taken from the locations addressed by the head pointer, and the head pointer is advanced around the circular buffer (Fig. 19).

The two keyboard status bytes are used by INT 9H to store information about the state of various modification (Shift, Alt, Ctrl) and lock (Num, Scroll, Caps, Insert) keys, as shown in Fig. 20. The keyboard status byte at $0040:0017 is used to store the current state of the modification keys, and the current status of the key locks. This information can be retrieved by application programs using the BIOS keyboard I/O interrupt (INT 16H Fn 02). The keyboard status

```
| 0040 001A | 2A 00 | 001B |
| 001C | 32 00 | 001D |
```

**Figure 18**  Layout of the keyboard buffer and pointers in the IBM PC.
THE CIRCULAR BUFFER IS EMPTY:
THE HEAD AND TAIL POINTERS
REFERENCE THE SAME LOCATION

INT 9H ADDS CHARS TO THE BUFFER:
THE TAIL POINTER ADVANCES AS
CHARACTERS ARE INSERTED

INT 16H READS CHARS FROM THE BUFFER:
THE HEAD POINTER ADVANCES AS
CHARACTERS ARE READ

ALL AVAILABLE CHARs ARE READ:
THE POINTERS AGAIN REFERENCE
THE SAME LOCATION. NEW ENTRIES
WILL ADVANCE THE TAIL STARTING HERE

Figure 19  Processing of keystrokes through the keyboard buffer
byte at $0040:0018$ is used by INT 9h to record which lock keys are currently depressed, as well as to record the state of some additional keys on the 101 key keyboard. This information is mainly for use by INT 9H itself, and is not generally of value to application programs. Thus, keystroke information is delivered to the motherboard by a microcontroller in the keyboard. This information is processed by INT 9H, then stored in a circular buffer and a pair of status bytes, where it can be retrieved by application programs through BIOS interrupt 16H.

### Keyboard Status Byte at Address 0040:0017 hex

<table>
<thead>
<tr>
<th>Insert</th>
<th>Caps</th>
<th>Num</th>
<th>Scroll</th>
<th>AltDn</th>
<th>CtrlDn</th>
<th>LShDn</th>
<th>RShDn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>Insert Toggle Status: 1 = On, 0 = Off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caps</td>
<td>Caps Lock Status: 1 = On, 0 = Off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num</td>
<td>Num Lock Status: 1 = On, 0 = Off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scroll</td>
<td>Scroll Lock Status: 1 = On, 0 = Off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AltDn</td>
<td>Alt Key Depressed: 1 = Yes, 0 = No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CtrlDn</td>
<td>Control Key Depressed: 1 = Yes, 0 = No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LShDn</td>
<td>Left Shift Key Depressed: 1 = Yes, 0 = No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RShDn</td>
<td>Right Shift Key Depressed: 1 = Yes, 0 = No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Keyboard Status Byte at Address 0040:0018 hex

<table>
<thead>
<tr>
<th>InsertDn</th>
<th>CapsDn</th>
<th>NumDn</th>
<th>ScrollDn</th>
<th>CtrlNum</th>
<th>101Key</th>
<th>101Key</th>
<th>101Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertDn</td>
<td>Insert Key Depressed 1 = Yes, 0 = No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CapsDn</td>
<td>CapsLock Key Depressed 1 = Yes, 0 = No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumDn</td>
<td>NumLock Key Depressed 1 = Yes, 0 = No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ScrollDn</td>
<td>Scroll Lock Key Depressed 1 = Yes, 0 = No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CtrlNum</td>
<td>Control-Num Lock Status 1 = On, 0 = Off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101Key</td>
<td>Defined only for 101 Key Keyboard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 20** Keyboard Status Bytes on the IBM PC.
4.1.2 Capturing Keyboard Input

One simple method of capturing keyboard input (for redistribution) is to trap the input at the INT 9H level. Replacing the BIOS INT 9H service routine with a new ISR (see chapter 2 for details on doing this) allows the keyboard data to be intercepted before it reaches the BIOS. However, as mentioned in [24], the BIOS INT 9h routine is hardware specific, so replacing it entirely can have undesirable side effects. The better solution is to have the replacement ISR call the original INT 9H routine to do the real scan code processing, and merely use the replacement to do pre and post processing of the keyboard data. Thus, a replacement INT 9H routine may first examine the incoming scan code (at port A), then call the original BIOS INT 9H to process it. Upon return from the BIOS interrupt, the replacement ISR would check if the BIOS routine placed a character in the keyboard buffer. If a character were added to the buffer, the routine could remove the character, and place it somewhere else, effectively capturing that character before any application program could read it. This method of capturing user input is depicted in Fig. 21.

One potential problem that can occur using this technique is reentrance of the replacement keyboard service routine. Normally, when INT 9H is generated, the Programmable Interrupt Controller will not let another INT 9H occur until the BIOS INT 9H service routine indicates that it is finished (by sending an End Of Interrupt (EOI) to the Interrupt Controller)\(^27\). However, when the replacement

\(^27\)See [16] for a discussion of the handling of hardware interrupts and the effects of reentrance.
Figure 21  General approach for trapping keyboard input on the PC
ISR calls the original BIOS routine, the End Of Interrupt is still sent by the BIOS routine, *before* the replacement ISR is finished. This means that a new INT 9H could occur while the replacement service routine in operation, possibly causing a system crash. One solution to this problem, examined in Kyle (1989), is to design the replacement ISR with an "in use" flag (semaphore). After calling the original interrupt service routine, the replacement routine will immediately exit if the flag indicates that the routine was previously in use (i.e. that nested interrupts have occurred). Thus, if another INT 9H occurs while the replacement ISR is still active, the new interrupt will exit before the reentrance can destroy any data, leaving the original interrupt to process the data. Another solution to this problem is to leave the interrupts turned off entirely after calling the original BIOS interrupt service routine. Because so little processing needs to be done after calling the original BIOS routine (just removing any new characters from the keyboard buffer), disabling the interrupts for this period will have no ill effects. This solution is used in the prototype system, and has been found quite effective for solving the reentrance problem.

Another issue that arises when handling user input is the treatment of certain key combinations (such as Ctrl-Alt-Del) which are specially processed by the BIOS INT 9H routine. These key combinations are treated within BIOS INT 9H instead of processed through to the keyboard buffer, and thus will not be captured for redistribution using the proposed technique. Worse still, these key combinations typically have drastic effects (such as rebooting the computer) which would be highly undesirable if not handled. As discussed in [24], the keys which
are processed internally by INT 9h are as follows: Ctrl-Numlock (or the dedicated Pause key on the 101-key keyboard) freezes the computer by placing INT 9H in a loop until another key is struck. Ctrl-Alt-Delete performs a soft reboot of the computer. Ctrl-Break clears the keyboard buffer, sets up break data in the BIOS data area, then calls the Ctrl-Break Interrupt (INT 1BH). Shift-PrtScr (or the dedicated PrintScreen key on the 101-key keyboard) calls the BIOS Print Screen interrupt (INT 05H). To handle this situation, the replacement INT 9h service routine can test for these key combinations before calling the original BIOS routine, and discard a special key combination if it occurs. Sensing the key combination consists of examining the scan code that is at port A (address 60H) and the status byte at $0040:0017, and checking if these values match any of the special key combinations. Discarding a key combinations consists of merely resetting the keyboard instead of allowing the BIOS INT 9H to process the scan code\(^{28}\). Resetting the keyboard consists of momentarily setting bit 7 of Port B (Address 61H) high, as discussed in [24]. Turbo Pascal code for sensing and discarding key combinations can be found in the KEYBOARD routine of the TSRU unit in [18]. Assembly code for doing this can be found in the BIOS source listing of [15]\(^{29}\). Thus, problems encountered when the BIOS processes certain key combinations can be solved by sensing these combinations and

\(^{28}\)Note that since the original BIOS routine is not called in this case, the Programmable Interrupt Controller must be reset before exiting the replacement INT 9H.

\(^{29}\)Note that the assembly code for resetting the keyboard presented in Fig. 2 of [24] is incorrect. Port B is at location 61H, not 20H as indicated.
discarding them, instead of invoking the BIOS to process them.

The trapping of keyboard input at the INT 9H level consists of using a replacement INT 9H service routine to supplement the BIOS keyboard handler. The replacement handler first checks if any special key combinations are represented by the incoming scan code, discarding the scan code if appropriate. Otherwise, the interrupts are disabled, and the original BIOS handler is invoked to process the scan code. When the BIOS handler returns (with the interrupts disabled), any character placed in the keyboard buffer is removed (and placed in another buffer), effectively trapping that keystroke before it can be read by an application program. This process is diagrammed in Fig. 22.

4.2 Buffering and Distribution of Keyboard Input

4.2.1 Buffering Keyboard Input for Redistribution

Once keystrokes are captured at individual workstations, they must be redistributed to every computer in an identical manner. To achieve this, keystrokes captured at each slave workstation are transmitted to the master workstation. The master workstation will order the keyboard input from the slave workstations, thus allowing this input to be redistributed to the slaves in an identical manner. The method chosen for distribution of buffered keyboard input from the master to the slaves significantly affects the performance of the system. Two methods for redistributing input will be considered.

In order to understand the advantages and disadvantages of a particular distribution method, it is necessary to examine the BIOS keyboard I/O interrupt
Figure 22  Algorithm for trapping keyboard input on the IBM PC
(INT 16H) functions in depth\textsuperscript{30,31}. Interrupt 16H function 0 is to used to read the next available keyboard character. When invoked, this function will either return the next character in the keyboard buffer, or wait until a character is entered in the keyboard buffer, and then return that character. This function is the only one needed in applications (such as editors, CAD packages, spreadsheets, etc.) where the computer merely waits for user input, acts on it, then waits for more user input.

Interrupt 16H function 1 is used to determine whether a keystroke is available to be read. The function returns a flag indicating whether there are any characters currently in the keyboard buffer, but does not remove any characters from the buffer. This function is used in conjunction with function 0 in interactive applications (such as games, simulators, etc.) in which the program is continuously active, and does not want to wait for keyboard data if it is not available. Programs which use this function call require careful treatment, because operation of the program depends not only on the order of the keyboard input, but also in the timing. To see this, consider the case of a game in which the action is advanced whether or not the player enters keyboard input. Typically, such a game operates in a loop where INT 16 Fn 1 is used to test for keystrokes periodically - an "If KeyPressed" loop. If a character arrives in the keyboard buffer one loop cycle later, the effects in the game may be completely

\textsuperscript{30}Because the DOS keyboard I/O functions ultimately invoke the BIOS functions, controlling the BIOS I/O is sufficient to control all keyboard I/O.

\textsuperscript{31}For a full description of the DOS and BIOS Keyboard I/O functions, see [20].
different, because the game has advanced farther. Thus, when operating such a program simultaneously on multiple workstations, keystrokes must be delivered to each application program on the same "If KeyPressed" cycle (INT 16H Fn 1 Call). This stringent requirement has strong implications when choosing a character distribution method.

4.2.2 Synchronized Distribution of Keyboard Data

Synchronized distribution of keyboard data is based on including INT 16H as part of the execution shell. That is, whenever an application makes a call to INT 16H, the execution shell at every slave machine communicates with the master via the synchronized exchange protocol of Chapter 3. The slave execution shells use the synchronized exchange to obtain information from the character buffer in the master computer (which contains the ordered characters from every workstation). Thus, characters are originally trapped at each workstation, and sent to the master. When the computers receive a request for keyboard data from an application program, data is returned from the buffer in the master computer.

The primary advantage of this distribution technique is that the programs operating on each computer are guaranteed to receive keystrokes in a precisely identical manner. In an "If KeyPressed" loop, for example, the individual computers synchronize with the master computer at each INT 16H call, and will each receive the same keyboard buffer information at each pass through the loop. The price of this advantage is a high communications overhead. Every time a program requests a keystroke, or tests whether a keystroke is available in the
buffer, a complete synchronized exchange must occur. The amount of data transferred in each exchange amounts to at most two bytes of character information (an ASCII code and a scan code), but the communications overhead to synchronize the computers and ensure data integrity amounts to much more than this. Because programs which utilize an "If KeyPressed" loop tend to call INT 16H quite often, this technique tends to lead to an excessive amount of time used to transfer data between the master and slaves. Furthermore, program execution is slowed even in the absence of user input, due to the need to synchronize the computers at every INT 16H call. Experimentation with this technique showed that the time lag between typing a character on a (slave) computer and seeing it appear at the screen was significant at modem-level communications bandwidths. Thus, distribution of keyboard data by absolute synchronization permits parallel execution of programs which use "If KeyPressed" (INT 16H Fn 1) loops, at the expense of operational inefficiency and a significant delay in character distribution among workstations.

4.2.3 Buffered Distribution of Keyboard Input

Buffered distribution of keyboard input is based on redistributing input to all of the workstations as soon as it is sequenced by the master workstation. Characters captured at the slave workstations are transferred to the master workstation, which organizes the data in a single buffer. A background process can then be used to distribute the ordered characters back to the slave workstations, where the data is placed in the local keyboard buffers.
The advantage of this distribution technique is its high efficiency. Because absolute synchronization between workstations is not necessary, the sliding window protocol of Chapter 2 can be used to transfer the data, leading to a higher efficiency than is possible with a synchronized protocol. Furthermore, because the workstations do not need to communicate at every INT 16H call, and because information can be transferred in blocks (instead of character by character), this method results in a lower communications overhead than obtained with the synchronized distribution technique. The primary disadvantage of this technique is that it may not function properly with applications that depend on the timing of keyboard input in addition to its order. As previously discussed, programs which utilize "If KeyPressed" (INT 16 Fn 1) loops may exhibit such timing dependent characteristics. Because data is placed in each local keyboard buffer as it arrives from the master workstation, different workstations may read the data from their buffer on different cycles through an "If KeyPressed" loop. If the application program is sensitive to this, the different workstations may cease to operate in parallel. This distribution method was used in the prototype version of the RSColl system, and was found to be highly effective with a broad group of application programs (editors, CAD packages, spreadsheets, etc.). The delays for character distribution were negligible, and program speed was not hindered by the character distribution system. However, there were some applications, including one CAD package, that would not function with the prototype system because of input timing dependencies. Thus, this technique has the advantage of high efficiency, at the expense of not being able to operate with certain types of
application programs.

4.3 Pointing Device Input on the IBM PC

In addition to keyboard input, many programs benefit from the use of pointing devices, such as mice, trackballs, and digitizing tablets. Because use of these devices, especially mice, has become widespread with graphics oriented programs (such as CAD and spreadsheet programs), handling of pointing device input is considered a necessity in a fully functional RSColl system. Because the I/O formats and requirements for different pointing devices (and even different mice) vary greatly, the task of capturing and distributing input from every type of device would be prohibitive. However, the vast majority of pointing devices are supplied with device drivers which permit them to be treated as a Microsoft mouse from an application’s point of view. The function of a device driver is to handle the hardware dependent control requirements of a device, while providing a standard interface to an application program. Thus, as long as pointing device input is handled at the interface between the application program and the device driver, many pointing devices can be treated as a mouse by the RSColl system.

4.3.1 Operation of the Mouse Subsystem

Because the mouse is not a standard device on the IBM PC, and because different mice may have different I/O requirements, mouse support has not been traditionally been included as part of the BIOS of most personal computers. However, operating system support for a mouse can be incorporated on a PC
through use of an installable device driver. The device driver handles the
(hardware dependent) information coming from the mouse, and provides a
(hardware independent) interface for an application program to retrieve mouse
data. The Microsoft mouse interface, described in [26], has become the defacto
standard for interfacing a mouse to the personal computer. Once a (Microsoft
compatible) mouse driver has been loaded into memory, mouse support services
are provided through interrupt 33H. These services, some of which are listed in
Table 3, are used just like standard DOS and BIOS services.

The mouse driver can be configured to operate in a variety of manners\[32\].
Normally, the mouse driver keeps tract of the screen mode, and displays a mouse
cursor which moves around the screen as the mouse is moved. This way, the
application program is not burdened with displaying and moving the mouse
cursor. The mouse cursor which appears may vary depending upon whether the
screen is in a text or graphics mode, and the cursor shape may be specified by
the application program when in graphics mode. Alternately, the mouse cursor
can be hidden, allowing a program to generate its own cursor symbol and control
it manually, while the mouse driver tracks the movement of the mouse. Thus,
the mouse driver tracks the mouse actions for the application program, and may
also be configured to handle the cursor transparently to the application.

\[32\]See [26] for a complete description of the mouse system.
<table>
<thead>
<tr>
<th>Service</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Mouse Reset and Status Test</td>
</tr>
<tr>
<td>1</td>
<td>Show Mouse Cursor</td>
</tr>
<tr>
<td>2</td>
<td>Hide Mouse Cursor</td>
</tr>
<tr>
<td>3</td>
<td>Get Mouse Position and Button Status</td>
</tr>
<tr>
<td>4</td>
<td>Set Mouse Cursor Position</td>
</tr>
<tr>
<td>5</td>
<td>Get Button Press Information</td>
</tr>
<tr>
<td>6</td>
<td>Get Button Release Information</td>
</tr>
<tr>
<td>11</td>
<td>Read Mouse Motion Counters</td>
</tr>
<tr>
<td>12</td>
<td>Set ISR Call Mask and Address</td>
</tr>
<tr>
<td>15</td>
<td>Set Mickey:Pixel Ratio</td>
</tr>
<tr>
<td>20</td>
<td>Swap Interrupt Service Routines</td>
</tr>
<tr>
<td>22</td>
<td>Save Mouse Driver State</td>
</tr>
<tr>
<td>23</td>
<td>Restore Mouse Driver State</td>
</tr>
</tbody>
</table>

Table 3  Some of the mouse support functions available through the Microsoft mouse interface.

Information about mouse movement and button presses can be retrieved from the mouse driver in either an interrupt driven or a polled manner. In the interrupt driven method, an application program can specify a set of interrupt routines which are executed by the mouse driver whenever the mouse is moved or a mouse button is pressed. This allows the program to respond immediately to the mouse, independent of other program action. Alternately, an application can periodically poll the mouse services interrupt and recover information such as how much the mouse has moved, and how many times the buttons have been pressed. This method is similar to periodically retrieving keyboard information from the keyboard buffer. The interrupt driven and polled techniques can also
be mixed, with some mouse actions causing program interrupts, while others are recorded for retrieval by the application program. As can be seen, there is a large degree of flexibility in the way an applications program can utilize the mouse driver.

4.4 Control and Distribution of Mouse Input

Because different mice (and other pointing devices) have different I/O requirements, handling of mouse input is constrained to occur at the INT 33H level, where the application program retrieves data from the mouse driver. If mouse input were to be controlled below this level, the hardware differences between various pointing devices would require a new device driver to be constructed for every pointing device made, which is clearly impractical.

The requirement that mouse input be handled at the INT 33H level has significant implications when trying to cause multiple workstations to operate identically. This is because the mouse driver itself is capable of handling the display and updating of the mouse cursor, as previously discussed. Since the RSColl system cannot intercept mouse data before it reaches the mouse drivers, the mouse drivers must not be permitted to handle the mouse cursors, because each workstation would display the mouse cursor in a different location. Any method chosen for handling mouse input must be designed to account for this phenomenon.

Another consideration when trapping and distributing mouse input is coordinating the mouse input with the keyboard input on each workstation. In order for applications programs running on separate workstations to behave
identically, the programs must each retrieve data from the mouse and keyboard in precisely the same order. For example, the effect of an application receiving a keystroke and then a mouse movement could be completely different than if the mouse movement were received first. Thus, the mouse data distribution must be coordinated such that each workstation receives the mouse data synchronized to the keyboard data in the same manner.

A final consideration when designing a scheme for mouse data distribution is the possibility of timing dependence on mouse data reception. Many programs will take user input (such as mouse data), act on that input, then wait for more user input. For programs in this category, only the order of the mouse input (and how it is coordinated with the keyboard input) is important. However, some programs (especially games) process data continuously, and the result of a mouse action in the program may be dependent on which call to INT 33H the mouse data was retrieved. This case is similar to the case of timing dependence on keyboard input discussed previously, and tends to occur in the same types of programs. To properly control a program that has this type of timing dependency, an RSColl system must ensure that the application program at each workstation receives the same mouse data on every call to INT 33H.

4.4.1 Handling of Mouse Input by Synchronization

One possible method of distributing mouse input is analogous to the synchronized method for distributing keyboard input. This method is designed to allow the mouse to be used independently of the keyboard: a characteristic that

83
is sacrificed by other mouse handling methods. While this technique is theoretically sound, it has not been implemented to date, due to the extreme operational deficiencies it is expected to exhibit. The technique is mainly presented to expose the difficulties in handling the mouse in the most general manner, i.e. as a source of input completely independent of the keyboard.

To implement a synchronized mouse distribution method, it is necessary to first trap the mouse input at each workstation, while not allowing the mouse drivers to display the mouse cursors. This task can be accomplished by extending the execution shell technique to INT 33H. The original mouse I/O interrupt (INT 33H) routine is replaced with an interrupt routine that acts as a meta-driver. The meta-driver uses the original driver to retrieve information from the mouse, but takes over the rest of the driver duties, such as displaying the mouse cursor and returning information to application programs. This way, the meta-driver effectively controls all information flow from the mouse to the application program (and the screen), yet remains independent of the mouse hardware, as shown in Fig. 23. In this capacity, the meta-driver can be used as part of the execution shell to ensure that every workstation displays the same mouse cursor position, and returns the same information to the application programs.

The synchronized method of handling mouse input is based on combining the mouse data from each workstation at the master workstation, then returning the same data and mouse cursor display information to each workstation. Trapping and redistribution of data can be handled through the meta-drivers at each workstation. Consider the operation of the technique if the application
Figure 23  Use of a meta-driver to control mouse I/O

program polls the mouse driver for data\textsuperscript{33}. Whenever the application programs at each workstation request data from the mouse, the meta-driver at each workstation sends the latest information about mouse movements to the master. The master evaluates this information, and returns identical mouse movement information to the meta-drivers at each workstation. The meta-drivers then update the display of the mouse cursors, and return the information to the requesting application programs.

The main benefit of this technique is the fact that it permits the mouse to be used independently of the keyboard. That is, both a mouse cursor and a

\textsuperscript{33}Operation of the technique for application programs which use the interrupt driven technique is similar (but slightly more complicated), and will not be discussed.
separate text cursor can be on the screen at the same time, and data can be entered using both cursors simultaneously. Independent mouse operation is necessary to handle programs which require the use of a mouse, and is desirable for any program which provides mouse support. Furthermore, as long as a synchronized distribution technique is used to handle keyboard data, this mouse handling technique will correctly function with programs that exhibit timing dependencies with mouse data reception. This is because identical data is distributed to each meta-driver on each call to INT 33H by an application program. Thus, this mouse handling technique is extremely flexible in terms of the types of application programs it will function with.

The flexibility that is gained through use of the synchronized mouse handling technique has an extremely large cost in terms of communications efficiency and overall system performance. First, consider the need to trap mouse input and synchronize it among multiple computers. Normally, when a mouse is moved, the number of interrupts to the mouse driver is quite large, and the mouse cursor position on the screen is updated quite often. However, to keep a group of computers operating identically, the meta-drivers need to synchronize and communicate with the master every time the mouse cursors are updated on the screen. This either leads to an extremely high communications overhead to manage the mouse, or an extremely slow and erratic updating of the mouse cursor, or both. Furthermore, in order for the mouse data to be coordinated properly with the keyboard data as previously discussed, both the mouse and the keyboard data must be distributed via synchronized techniques. Otherwise, there
ould be variations in the timing of keyboard data reception with respect to mouse data reception. The need to use a synchronized technique for the keyboard further exacerbates the communications overhead problem, and introduces the performance deficiencies associated with synchronized keyboard data distribution.

Thus, handling the mouse synchronously among computers allows the mouse to be used as a source of input which is independent of the keyboard, at the expense of poor overall system performance. The reasons for the resulting poor performance are the fact that the mouse cursor must be updated synchronously among computers, and the fact that the mouse input must be synchronized to the keyboard data among workstations.

4.4.2 Handling of Mouse Input by Keystroke Conversion

The previous section illustrated the difficulties in handling the mouse as an independent source of input on the PC. To avoid these difficulties, a method has been devised which permits a mouse to be used effectively with many types of programs, at the expense of not providing full mouse support. In many programs, including many major CAD programs, a mouse facilitates moving the program cursor around, but is not actually required for program operation. For programs of this type, the mouse may be effectively handled by converting the mouse movements and button presses into keystrokes. The so-called keystroke conversion technique is based on converting mouse motion into streams of arrow keys, and mouse button presses into appropriate keystrokes or key sequences.

The keystroke conversion method operates by having the execution shell
transfer data from the mouse to the application program through the keyboard buffer. To make such a system work properly, the application program must not be permitted to access the pointing device itself. Hiding the existence of the mouse from an application program can be achieved by redirecting the mouse services interrupt (INT 33H) to point to an IRET instruction. When a program calls the mouse reset and status function (INT 33H Fn. 0) to check for the existence of the mouse, the call will immediately return, indicating that the mouse driver has not been found\textsuperscript{34}. To access the mouse driver, the execution shell at a workstation can temporarily restore the original mouse driver interrupt, and call it just as an application program would. In addition to hiding the mouse driver from application programs, it is necessary to ensure that the mouse cursor is hidden from the screen. This can be accomplished with a call to the mouse driver (INT 33H, Fn. 2).

Mouse data can be converted to keyboard data as part of the background process, as shown in Fig. 24\textsuperscript{35}. The background process first reads the mouse motion counters, to find out how much the mouse has moved since the mouse was last checked\textsuperscript{36}. The motion counters contain horizontal and vertical movement counts in units of *mickeys*, which are approximately 1/200th of an inch. To

\textsuperscript{34}Some Programs require the user to specify what pointing device is installed in the system. In this case, the application program should be configured for no pointing device.

\textsuperscript{35}An interrupt driven method could also be used, with similar results.

\textsuperscript{36}[26] contains code for calling the various mouse functions in BASIC and Assembler. [18] contains Turbo Pascal code which can be used to call the various mouse driver routines.
Figure 24  Operation of the keystroke conversion method by polling
convert this motion to keystrokes, the background process adds a remainder (from the previous keystroke conversion) to the horizontal motion count, then integer divides this value by a sensitivity parameter. This operation yields the number of (left or right) arrow keys to enter in the buffer. The sensitivity parameter provides calibration between the motion of the mouse and the effect in the program, with values in the range of 15 to 150 being normal. A modulo division of the mouse counter value by the sensitivity parameter provides the remainder to be used in the next conversion. The calculated number of keystrokes are entered in the (keyboard capture) buffer, to be transmitted to the master workstation. The entire process is then repeated for vertical mouse motion. The mouse buttons are handled in a similar manner: First, the number of Left button presses since the last check is retrieved (using INT 33H, Fn. 5)\textsuperscript{37}. For each press of the button, a character (or sequence of characters) is inserted in the keyboard capture buffer, to be transmitted to the master workstation. Once the left button is handled, the process is repeated for the right mouse button\textsuperscript{38}. The character (or sequence) inserted in the buffer should be a useful command within a given application, and can be changed for different applications. For example, in the OrCAD Computer Aided Design package, a Carriage Return is used to mark points on the screen, and would therefore be a logical character to represent.

\textsuperscript{37}Note that slightly more precise ordering of mouse motion and button presses could be achieved using an interrupt driven approach. However, this approach proved to be quite satisfactory when tested in the prototype system.

\textsuperscript{38}The reset function returns the number of buttons actually present on a pointing device. Thus, a different number of buttons could be handled as an extension of the method shown.
a button press. Thus, information is retrieved from the mouse by the background process, and converted to an appropriate sequence of characters. The characters are inserted in a buffer along with any captured keyboard input, to be sent to the master workstation. The conversion code used in the prototype system is shown in Fig. 25.

There are two major advantages in using the keystroke conversion method instead of a synchronized method. First, because no separate mouse cursor exists, this method eliminates the burden of synchronizing mouse cursor motion among computers. Second, because mouse data is converted to keystrokes, there are no problems with synchronizing the mouse to the keyboard, and timing dependency issues (with data distribution) only apply to the keyboard subsystem. Thus, the major deficiencies of the synchronized method of mouse handling are avoided. However, because this method does not support a separate mouse cursor, or allow an application to directly use mouse support services, this technique may function poorly (or not at all) with some application programs. How well an application program will perform depends on the extent to which it relies on mouse services being available, and how it utilizes the mouse services. Thus, the keystroke conversion method provides very efficient mouse support, at the expense of not permitting application programs to use a mouse in the most general manner.
Procedure ReadMouseData;
(* *****************************)
(* This procedure queues the mouse driver for movement and *)
(* key data. Mouse movement is converted to Arrow keys in *)
(* the proper directions with sensitivities determined by *)
(* the MOUSESENSIT variable. Remainder of mouse movement *)
(* after conversion is used in the next invocation. Mouse *)
(* buttons are converted to keyscans as defined in the *)
(* constants at the beginning. The converted data is then *)
(* placed in the transmit buffer just as if it were *)
(* keyboard data. Note that the routine can only be called *)
(* when the real mouse driver interrupt vector is installed. *)
(* *****************************)
Var
MouseCount, Vcount : integer;
KeyScan : Byte;
K : word;

Procedure ReadMouseData;
(* HANDLE THE MOUSE INPUT *)
Begin
(* HANDLE MOUSE MOVEMENT *)
ReadMouseMotionCount := MouseCount, Vcount;
Hcount := Hcount + OldMouseHorMotion;
Vcount := Vcount + OldMouseVertMotion;
OldMouseHorMotion := MouseCount mod MouseSensit;
OldMouseVertMotion := Vcount mod MouseSensit;
(* horizontal motion *)
If Hcount > 0 then KeyScan := RightKeyScan;
else KeyScan := LeftKeyScan;
for k := 1 to Abs(Hcount) do KeyScan : div MouseSensit do
Begin
if SpecifiedLeft then
Begin
CLU (* int off to modify buffer *)
 TxBuf[TxTail_Ptr] := Arrow.ASCII;
 TxTail_Ptr := TxTail_Ptr + 1;
 KeyScan := KeyScan;
 if TxTail_Ptr < Buff and 1
 Then TxTail_Ptr := TxTail_Ptr + 2
 Else TxTail_Ptr := 0;
 STI (* int back on *)
 end;
end;
(* Vertical motion *)
If Vcount > 0 then KeyScan := UpKeyScan;
else KeyScan := DownKeyScan;
for k := 1 to Abs(Vcount) do MouseSensit do
Begin
if SpecifiedLeft then
Begin
CLU (* int off to modify buffer *)
TxBuf[TxTail_Ptr] := Arrow.ASCII;
TxTail_Ptr := TxTail_Ptr + 1;
KeyScan := KeyScan;
if TxTail_Ptr < Buff and 1
Then TxTail_Ptr := TxTail_Ptr + 2
Else TxTail_Ptr := 0;
STI (* int back on *)
end;
end;
(* handle mouse buttons *)
GetButtonPressInfo(); (* Left Button *)
for k := 1 to ButtonPressCount do
Begin
CLU (* int off to modify buffer *)
TxBuf[TxTail_Ptr] := LeftButton.ASCII;
TxTail_Ptr := TxTail_Ptr + 1;
if TxTail_Ptr < Buff and 1
Then TxTail_Ptr := TxTail_Ptr + 2
Else TxTail_Ptr := 0;
STI (* int back on *)
end;
GetButtonPressInfo(); (* Right Button *)
for k := 1 to ButtonPressCount do
Begin
CLU (* int off to modify buffer *)
TxBuf[TxTail_Ptr] := RightButton.ASCII;
TxTail_Ptr := TxTail_Ptr + 1;
if TxTail_Ptr < Buff and 1
Then TxTail_Ptr := TxTail_Ptr + 2
Else TxTail_Ptr := 0;
STI (* int back on *)
end;
End; (* If mouse is present *)
End;

Figure 25 Keystroke conversion code used in the prototype system.
The keystroke conversion method is optimized for use with applications where a single program cursor is present, and is manipulated by either keyboard or pointing device input (when present). For programs of this nature (including many major CAD packages), there is little or no difference between the normal performance of the program and the performance when keystroke conversion is used. Alternately, some programs either use the mouse for fast access to program services (via pulldown menus, etc.), or do not provide any mouse support at all. Programs of this type include many editors and spreadsheet packages. The keystroke conversion method will not provide the standard mouse support within programs of this variety. However, the ability to enter arrow keys and command sequences rapidly using the mouse provides a limited form of mouse support (even if the program supplies no mouse support at all!). Experimentation with different programs has shown that this form of support is often acceptable, though not as desirable as full mouse support. Finally, some programs rely on full mouse support to function properly. This is characteristic of programs which utilize a mouse-driven Graphical User Interface (GUI), including some drawing and painting programs. Programs of this genre will not function with the keystroke conversion technique. To support programs of this type, a synchronized mouse handling method must be used. Thus, the keystroke conversion method has been found to function adequately with a large cross section of available programs, with the notable exception of programs which rely on full mouse support.

This chapter has thus covered the operation of the keyboard and mouse
subsystems on the IBM PC. Two methods for capturing and distributing keyboard data have been developed, and their relative merits examined. Two methods for handling the mouse have also been discussed, and their relative merits examined. The prototype system uses the buffered method of keyboard distribution, and the keystroke conversion method of handling the mouse. This combination provides extremely efficient management of user input, and has been found to function well with a wide variety of application programs. However, as discussed, the choice of these methods has imposed certain limitation on what applications will function properly with the prototype system.
Chapter 5: Controlling Parallel Execution

To implement an RSColl system, copies of a program running on different computers must be constrained to behave identically. As discussed in Chapter 2, parallel execution among computers can be controlled by regulating the flow of data between application programs and the operating system on each computer. This chapter considers the task of controlling parallel execution among computers in this manner using an execution shell. First, the operation of an execution shell is reviewed, and the conditions on which the execution shell technique relies are examined. Second, some of the technical aspects in constructing a PC-based execution shell are explored. Finally, this chapter considers the limitations of an execution shell in the PC environment, and the resulting limitations on a PC-based RSColl system.

5.1 The Execution Shell Technique Revisited

5.1.1 Operation of an Execution Shell

As previously discussed, an execution shell is a set of interrupt service routines which replace certain operating system interrupts on each computer. By intercepting calls to the operating system, an execution shell can be used to regulate parallel execution among a set of workstations. When copies of a program running on multiple computers make an operating system request, the execution shells at each computer are activated. The execution shells collect information about the system request, then communicate with the master execution shell (via the synchronized protocol of chapter 3). The master
execution shell interprets the data received from the slaves, and instructs the slave execution shells on how to handle the system request. In the prototype system, for example, requests to open a file are intercepted by the execution shells at each workstation. The execution shells perform a checksum on the files at each workstation, and transmit this information to the master. The master decides the course of action based on the data from the slaves: If the files at each workstation are identical (based on checksum, name, etc.), the master instructs the slaves to pass the request on to the operating system (to open the file). If the files are not the same, or there is some problem with opening the file on one or more systems, the master instructs the slave execution shells to reject the request, and return an error to the application program. By ensuring that each computer returns the same response to a file open request, and by ensuring that files opened on the separate computers are identical, parallel execution of the application programs is maintained. Thus, the execution shell technique ensures parallel execution among workstations by controlling information flow between application programs and the operating system at each computer.

5.1.2 Assumptions of the Execution Shell Technique

The assertion that an execution shell can regulate parallel execution among separate workstations relies on several assumptions about the nature of the application programs and workstations used. These assumptions represent a set

---

39Note that there are two separate file management methods under DOS, both of which are handled by the prototype system.
of conditions which should (ideally) be satisfied for an RSColl system to work
properly. This section considers some of the assumptions on which an execution
shell relies.

First, for the execution shell technique to function properly, the operating
system interface at each computer must be the same. In the PC environment, the
same version (and configuration) of DOS must exist at each computer, and the
BIOS at each computer must have the same programming interface. If the
operating system interface at one computer provides different services than
another computer, or even handles the same services with different results, there
is no way to ensure that both computers will operate in parallel. For example,
when opening a file under DOS (using INT 21h, Fn 3Dh), some opening modes
are only available under DOS 3.X, and not DOS 2.X\textsuperscript{40}. If one computer is
running DOS 3.2, and another is running DOS 2.2, a file open request issued by
a program could result in different file-open modes on the two computers. The
results of file operations could then differ in a manner not controllable by an
execution shell, leading to divergent operation of the computers. Thus, to
regulate parallel operation among computers using an execution shell, the
operating system interface at each computer must be the same.

A program makes decisions based on information it retrieves from the
environment around it. For copies of a program running on multiple workstations
to behave identically, they must each receive the same information from the
surrounding environment. Thus, a second assumption of the shell technique is

\textsuperscript{40}See [20] for details.
that any information retrieved by a program is either identical at every workstation, or is controllable by the execution shell. Generally, this means that a program must retrieve all data it uses through the standard operating system calls, so that the execution shell can regulate it\(^{41}\). To illustrate this point, consider the retrieval of keystroke data from the keyboard buffer. If a program requests keystrokes via the BIOS keyboard I/O interrupt (INT 16H), the execution shell can regulate the information returned to the program at each call. However, if the program retrieves keystroke data directly from the buffer (by examining memory locations), there is no way to regulate how and when the program will retrieve keystrokes. Thus, if a program does not invariably go through the operating system (via the interrupt structure) to retrieve data, execution of the program may not be controllable among separate workstations.

The execution shell technique also relies on the assumption that every execution shell will receive operating system calls in an identical order. If a computer is running any background processes which interact with the main program or the operating system, this assumption may be violated. To see this, consider the case in which a background process on each computer invokes the operating system when triggered by a periodic interrupt (for instance, when an editor’s auto-save function is triggered by a timer interrupt). Because the background process is acting independently of the foreground process on each computer, calls to the operating system may occur in different orders on different computers. Under these conditions, it would not be possible to regulate parallel

\(^{41}\)Some exceptions to this statement will be discussed later in the chapter.
execution of the workstations, since the possible sequence of actions at each machine is not unique. Practically speaking, this assumption places two restrictions on the operation of the computers. First, no secondary processes (such as TSR programs) can be active on a computer while running under an execution shell. Second, application programs managed by an execution shell cannot contain interrupt service routines which are triggered by (uncontrolled) external events. Thus, the execution shell is constrained to operate with a single foreground task that does not rely on (uncontrolled) external interrupts.

The execution shell technique is designed to control the inputs to a program, and allow the program to manage its own output. However, if the output of a program affects different workstations differently, it may not be possible to regulate parallel execution of the workstations. For example, I/O port assignments for peripheral devices in a personal computer are often selectable by jumpers, and may vary from computer to computer. If copies of a program running on two computers write to a specific port address, trouble can result if the address maps to different devices on the two computers. The effects of writing the data to different devices may make it impossible to regulate the execution of the computers thereafter. Thus, the execution shell technique strongly relies on the program outputs not affecting computers in a significantly different manner.

As can be seen, operation of an execution shell system relies on several assumptions about the behavior of the computers and application programs used. These assumptions (and where they are violated) have a profound impact on the
implementation of a DOS-based RSColl system, and ultimately lead to some of the major limitations of such a system.

5.2 Issues in implementing an execution shell system

Several ancillary issues crop up when implementing an execution shell system in the PC compatible environment. This section treats some of the major issues.

5.2.1 Hooking Interrupt 21H

To trap operating system calls in the DOS environment, it is necessary to hook (redirect) DOS interrupt 21H. Two basic issues arise when replacing interrupt 21H with execution shell code. One is that not all calls to interrupt 21H return to the caller. As discussed in Chapter 8 of [19], calls to the DOS termination functions (INT 21, Fns. 0, 31, and 4C hex) return control to the parent process, instead of the calling process (which is terminating operation). Thus, when the execution shell intercepts a termination call, it cannot do any processing of data after calling the real DOS termination function. Any processing normally performed after the call to DOS must be performed beforehand, including the exit duties of the interrupt service routine. For example, any data stored on the stack by the execution shell should be cleaned up prior to calling DOS42. As long as all post processing and exit duties are

---

42In Turbo Pascal (and many other high level languages), register information is often stored on the stack as local variables, and must be cleaned up [27].
performed before transferring control to the real DOS handler, no problems will result.

A second issue that arises when hooking interrupt 21h is the fact that DOS INT 21h is partially reentrant. Because DOS uses multiple stacks internally, there is nothing to prevent DOS INT 21H code from calling INT 21H itself, so long as the proper stacks are used. In fact, this situation occurs quite often when internal DOS functions are utilized by DOS itself. To operate properly, execution shell code which intercepts INT 21h calls must be designed to either handle such reentrant conditions, or avoid them. To avoid reentry, a replacement INT 21H ISR could swap the vector table pointer to reference the real DOS INT 21H before issuing any DOS calls, as depicted in Fig. 26. This way, any reentrant DOS calls would get routed to the real DOS routine, and not through the execution shell. When the DOS calls made by the execution shell are completed, the execution shell ISR could restore its own address in the vector table, ensuring that future calls would be routed through the execution shell. One problem encountered with this technique is that the hook into the operating system is often lost if all INT 21H interceptions are handled this way. That is, at some point the execution shell’s address does not get restored in the vector table, effectively disabling it. The reason for this is not known, but it is suspected that some internal DOS calls do not return normally, so that the address is never restored on those calls. To solve this problem, the prototype system swaps the interrupt vector only on calls that are managed by the execution shell (which are known to return normally). Other DOS calls are passed directly on to the real INT 21H
Figure 26  Processing arrangement to avoid reentrance of execution shell code
handler, as shown in Fig. 27. In these cases, reentry is not a problem, because no critical data will be lost in a reentrant condition. Thus, by managing the interrupt structure properly, the reentrant properties of DOS can be managed safely.

![Diagram](image)

**Figure 27** Managing the INT 21H processing arrangement

### 5.2.2 Error Handling

Another issue that arises when constructing a DOS based RSColl system is the handling of system errors. Two types of errors are defined under DOS: function request errors and critical errors. Function request errors are merely
error codes returned to a program when DOS function calls are made, and can be easily managed. Critical errors are issued by DOS in certain crucial situations, such as when bad memory is encountered or when attempting to access a diskette drive whose door is open. These errors require more careful treatment, because DOS invokes an error handler interrupt when these situations occur.

Function request errors are part of the information returned to a program when it calls a DOS function. Most DOS functions indicate an error situation by setting the carry flag on return from the function call, and placing a standard error code in the AX register\textsuperscript{43}. Under DOS versions 3.X+, a program can retrieve extended error information by calling DOS function 59H when an error indication is returned. Because all of the information is exchanged through DOS function calls, function request errors are easily handled by an execution shell. Furthermore, under DOS 3.1+, an internal function (5D0Ah hex) is used by DOS to set up the extended error information\textsuperscript{44}. An execution shell can use this internal DOS call to set up its own information, or trap DOS when it sets up the information. Thus, function request errors can be easily controlled by an execution shell.

Critical errors are generated by DOS when it encounters certain critical situations while processing a function request (Table 4). When an error of this type occurs, DOS issues an INT 24h to invoke a critical error handler, whose function is to either handle the error or instruct DOS how to do so. DOS

\textsuperscript{43}See [20] for details on error information returned for various DOS calls.

\textsuperscript{44}See [19] for details on this (undocumented) internal DOS function.
provides a default error handler, which prompts the user for direction with the familiar "Abort, retry, or ignore?" error message. An application program can install its own critical error handler by redirecting interrupt 24h. A replacement handler receives critical error information from DOS, and may also use DOS I/O functions 00-12H to communicate with the user. The handler must either process the error itself, or instruct DOS how to handle it (i.e. fail the function request, terminate the program, etc.).

Because a critical error handler can itself access the operating system and determine the course of program operation, it is essential that it be directly controlled by the execution shell. To prevent critical errors from affecting parallel execution among computers (or to at least detect the problem), the prototype system installs its own critical error handler in place of the default handler. The execution shell then prevents DOS (or an application program) from replacing this critical error handler by trapping DOS requests to do so (and ignoring them). This way, any critical error occurrences are trapped by the execution shell error handler, and can be managed transparently to the application program. In the prototype system, the replacement handler merely instructs DOS to fail all function requests encountering errors, thus allowing the execution shell to deal with the failure when the function call returns. This has proven to be both simple and effective, and permits use of more sophisticated error handlers in the future.
<table>
<thead>
<tr>
<th>DOS Error Code</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Write attempt failed (write protected disk)</td>
</tr>
<tr>
<td>20</td>
<td>Specified unit is undefined</td>
</tr>
<tr>
<td>21</td>
<td>Drive not ready</td>
</tr>
<tr>
<td>22</td>
<td>Unknown command</td>
</tr>
<tr>
<td>23</td>
<td>Cyclic Redundancy Check (CRC) data error</td>
</tr>
<tr>
<td>24</td>
<td>Bad command sequence length</td>
</tr>
<tr>
<td>25</td>
<td>Seek error</td>
</tr>
<tr>
<td>26</td>
<td>Unknown disk type</td>
</tr>
<tr>
<td>27</td>
<td>Sector not found</td>
</tr>
<tr>
<td>28</td>
<td>Printer error: Out of paper</td>
</tr>
<tr>
<td>29</td>
<td>General write fault</td>
</tr>
<tr>
<td>30</td>
<td>General read fault</td>
</tr>
<tr>
<td>31</td>
<td>General failure</td>
</tr>
</tbody>
</table>

Table 4  DOS Critical Errors and standard error codes.

5.2.3 Managing the environment table

As previously discussed, the execution shell generally relies on a program retrieving all of its information through the standard operating system calls. Application programs often violate this assumption by use of the DOS environment table. Proper treatment of this situation is essential for implementing a DOS based execution shell system.

The DOS environment table is a block of space that is allocated with each process. As described in [19], the table contains a collection of environment variables, which consist of variable names and data stored in ASCII strings.
Predefined environment variables familiar to DOS users are the COMSPEC, PATH, and PROMPT variables. Other environment variables can be defined in the DOS shell through use of the "SET" command, and are often used by application programs to store configuration information. The environment table thus acts as a library of text information which may be used as a reference by applications programs.

The problem that arises with the environment table is that programs do not go through the operating system to access it. A program retrieves the address of the environment table from its Program Segment Prefix (PSP), and reads the table directly [19]. Because the execution shell cannot intercept program access to the environment table, the data contained in it must be the same at each computer. Thus, an execution shell system should either verify that the tables are the same at each computer, or directly control the contents of the environment tables on the slave computers.

5.2.4 Backdoor calling of INT 21h

Another issue that crops up when designing an execution shell under DOS is managing "backdoor" access to DOS. The execution shell technique relies on trapping calls to the operating system by redirecting the operating system interrupts. While most programs access the operating system through the interrupt structure, there are actually two other legitimate methods to access it. These methods allow backdoor access to DOS using an alternate DOS function dispatcher, as described in [13]. To control backdoor access to the operating
system, it is necessary to intercept calls to the alternate function dispatcher and redirect them through the execution shell. One technique for redirecting alternate dispatcher calls is documented in [13]. Use of this technique will allow execution shell control of programs which employ backdoor DOS access methods.

5.3 Limitations of the Execution Shell Technique

This section documents some of the fundamental limitations encountered when implementing an execution shell system in the PC environment. These limitations follow directly from the assumptions on which the execution shell technique relies, and how they are violated in practice.

5.3.1 Limitations imposed by information retrieval methods

As previously discussed, one assumption of the execution shell technique is that any information retrieved by a program is either identical at every workstation, or is controllable by the execution shell. Unfortunately, many commercially available programs retrieve information by interacting directly (and uncontrollably) with the computer, often to improve program performance. For example, programs often read data directly from absolute memory locations such as the BIOS data areas or the video tables. Because this information is often hardware dependent, there is no way to ensure that it is the same across computer systems unless they are identical. Thus, the execution shell technique is limited to either extremely "well behaved" programs, or to situations where identical workstations are used.
Not only do many application programs retrieve data without going through the operating system, the standard DOS Command Interpreter (COMMAND.COM) is also "misbehaved" in this sense. For example, COMMAND.COM will often directly examine computer memory to determine its actions\textsuperscript{45}. Therefore, COMMAND.COM itself cannot be controlled on disparate computer systems by an execution shell, as its actions may depend on local hardware conditions. Because many programs may spawn a copy of COMMAND.COM (to execute a DOS "Shell", or to use DOS utilities), extreme care is necessary when treating this situation. Essentially, an RSColl system must either replace the DOS Command Interpreter with one of its own, or not allow programs to spawn a copy of COMMAND.COM. This illustrates the seriousness of the limitations on information retrieval by application programs.

5.3.2 Limitations imposed by the device driver system

As previously discussed, the execution shell operates under the assumption that the operating system interface is the same at each computer. In the PC environment, this means that the same version of DOS must exist at each computer. However, additional restrictions are imposed by the existence of installable device drivers under DOS.

A device driver is a piece of software which provides a standard interface

\textsuperscript{45}One instance of this is when COMMAND.COM determines whether or not to reload its transient portion by directly examining memory.
between an application program and a piece of external hardware. An application program generally communicates with a device using the standard DOS I/O services. DOS converts these I/O requests into a series of (standard) device driver commands. The device driver then communicates with the device, and returns information to DOS in a known manner, as shown in Fig. 28. To interface with the device, the device driver must either contain or have access to the hardware-dependent control code for the device. Basic control code for many standard devices in the PC is available in the BIOS. A device driver may either use the BIOS routines to access a device (when applicable), or directly access the device through the PC I/O bus. Thus, a device driver acts as part of the operating system interface between an application program and the computer hardware, and can itself use BIOS services.

When a device driver is installed, it effectively customizes the operating system for interfacing to a particular device. Unfortunately, if two computers have different device drivers installed, the operating system interface may be different enough to violate the requirements of an execution shell. To see this, consider the case where a device driver named "foobar" is installed on one computer, but not another. If a program attempts to open "foobar" for output, the effects on the two computers could be completely (and unrecoverably) different, since in one case a file would be opened, and in the other a device would be opened. Thus, differences in device drivers can have a significant

---

46A detailed discussion of the use and design of MS-DOS device drivers can be found in [28].
Figure 28  Interactions between a program, DOS, a device driver, and the device.

impact on the ability of an execution shell to control a set of computers.

Differences in device drivers can also affect operation of an execution shell on another level, due to the fact that device drivers are allowed to access the BIOS. Consider the case of two device drivers (on two computers) which perform the same function, but interact differently with the BIOS when prompted with the same command. If an execution shell system monitors the part of the BIOS (on each computer) in which different interactions occur, parallel execution could not be regulated, since the BIOS calls at the two machines are different. Thus, device drivers must be functionally equivalent at the BIOS level as well as the DOS level for an execution shell to reliably control a set of computers.

The fact that the device drivers at each computer must be (functionally)
identical at every computer places strong limitations on the execution shell technique. Because the device drivers at every system must be identical, the computers must be sufficiently identical to be driven by the same device drivers. This is a very stringent requirement, since device drivers are inherently very hardware dependent.

As can be seen from this chapter, controlling parallel execution of workstations by the execution shell technique relies on several assumptions about the computers and programs being controlled. Because of these assumptions, certain restrictions are placed on both the computers and applications programs employed in an RSColl system. Programs must be sufficiently well behaved in the way they retrieve data and interact with the computer system. The computers must be configured the same (at the operating system level), and the computer hardware must be identical from the device driver standpoint.
Chapter 6: Evaluation and Conclusion

In addition to the technical issues involved in constructing a shell-based system, there is the question of the utility of such a system in the real world. This chapter first describes a preliminary evaluation of the RSColl 'shell' approach to determine under what conditions it would be useful. Following this, overall conclusions are drawn about the research conducted in this thesis. Finally, some suggestions are made for future work in the area.

6.1 Preliminary evaluation of the RSColl approach

The need for computer-based tools which support remote collaboration has motivated the development of the RSColl 'shell' approach. In spite of the known difficulties in appraising such multi-user interfaces [29], a preliminary evaluation of the shell approach has been made using the prototype system as a test vehicle. The goal of this preliminary evaluation was to gain a basic understanding of the benefits and liabilities inherent in a shell-based interactive environment.

6.1.1 Evaluation Methodology

The exploratory study consisted of four interactive sessions between pairs of collaborators, each lasting no longer than one hour, and each separated by about a week. The study was set up such that the collaborators were colleagues who had worked together previously, and who were each familiar with the tasks at hand.

The four sessions were conducted over 1200 baud serial links, with the
addition of two-way audio communication. Each collaboration session was conducted in the environment of a different commercially available software package, with which at least one of the users was experienced (see below).

The first collaboration session involved joint editing and review of a partially completed scientific document. The session was carried out in the environment of a word processing package with WYSIWYG and graphics capabilities. While both collaborators were familiar with the word processor, only one of them was versed in its equation editing tools, providing an opportunity to examine the instructional potential of the shell-based system.

The second collaboration session involved editing and review of technical illustrations in the environment of a popular spreadsheet / graphics package. This session provided a good opportunity to see the performance of a shell-based system in a situation where a shared view of graphical images was central to the task at hand.

The third collaboration session was carried out in the environment of a popular CAD package, with the intent of performing preliminary "brainstorming" for a graphic design. No predefined data were brought into the design session, except for a vague verbal specification of the desired result. The object of this session was to examine the usefulness of a shell-based system in situations where the two collaborators were attempting to initiate a project, rather than extend existing work.

The final collaboration session involved review and discussion of a partially completed circuit design, in the environment of a widely used electrical
CAD package. This session provided an opportunity to examine the potential of a shell-based system in an engineering design context, where graphical communication is often important.

6.1.2 General Observations

The four formal collaboration sessions, in conjunction with more casual use of the prototype system, yielded a wealth of information about shell-based collaboration and its potential advantages and liabilities. One conclusion that can be drawn from preliminary experiences with the prototype system is the need for separate audio communications links between collaborators. Early in the project, it was hypothesized that effective collaboration might be feasible using only chat boxes (textual conversation) or multiplexed voice/data. With experience, it became clear that an independent voice link was a highly desirable addition for the system to be effective. The addition of an independent voice link allows simultaneous action (in the workspace) and verbal discussion, which greatly facilitates the communication of ideas. This finding corroborates the conclusions of [3] and [8] that multiple channels of communication are necessary for effective computer-mediated collaboration support. While there is an apparent need for a separate audio connection, this does not seem to pose any problem, since all that is required is another low-bandwidth (phone) link. As long as two-way audio communication was provided, the prototype system allowed users to communicate in an acceptable manner.

One question that was of great concern when developing the prototype
system was whether control locking and passing mechanisms would be necessary for the system to be effective. Because a shell-based system only permits a single entry point into an application program (which must be shared among users), there was concern that users trying to act simultaneously would consistently interfere with one another, perhaps significantly degrading the quality of interaction. If this were the case, then some type of control locking and passing mechanism would be needed to allow only one user to enter data at a time. Experience with the system has shown that, at least for small groups, formal control passing is unnecessary. Two collaborators working together within the framework of a program rapidly develop a sensitivity for alternately acting within the shared environment. This can be likened to the experience of a dialogue over a two-way radio, where the two users naturally alternate speaking and listening. In addition, it was found that informal rules for sharing the workspace rapidly developed between users, further reducing the need for control passing. For example, when editing a document jointly, it rapidly became accepted that a collaborator could add, mark or rearrange text, but could not delete text without permission. Thus, it has been found that for (small) groups of (nonantagonistic) collaborators, control locking and passing is not necessary, and would probably only hinder collaboration in a shell based system.

The apparent utility of the prototype system was observed for various functions associated with collaboration. Brainstorming is one collaborative function that is often considered when developing collaboration tools [7],[9]. As considered here, a brainstorming session "involves the initial generation of ideas"
[7], where users start with a clear workspace and attempt to formulate new approaches or concepts. Preliminary experience with the prototype system has indicated that the shell-based approach is not well suited to this particular type of activity. The major reason for this seems to stem from the nature of collaborative interaction at the genesis of a project. When a project begins, each user often needs to spend relatively large amounts of time to create representations of ideas to share with collaborators. During this period of time, it is difficult to build upon each others' ideas, since their representations have not been sufficiently fleshed out. Therefore, at this stage there is a need for collaborators to work semi-independently (and preferably simultaneously). In an RSColl shell system, the users must all share a single entry point into a program's workspace, thus restricting users to taking turns at expressing themselves and pursuing ideas. Because this restriction conflicts with the needs of collaborators at the beginning stages of a project, a shell-based system will not perform well for that situation. Thus, it seems from our experience that the nature of a shell-based user interface inhibits its use for pure brainstorming interactions.

In contrast to its performance as a brainstorming tool, the prototype system was found to be very effective for development and extension of existing collaborative projects. This finding seems to support the prediction in [2] that "computer-mediated communication will be more valuable for coordinating already existing collaborative projects than for starting new ones". The reason performance is so much better when collaborating on an existing project seems to be related to the focus of the interactions. When two collaborators operate
semi-independently (such as when brainstorming), the single entry point interface of a shell-based system obstructs the flow of ideas. However, when both collaborators are focussed on a single object or section within the shared workspace (which is more prevalent when extending existing work), a single data entry point does not seem to inhibit interaction or development of ideas. In fact, by enforcing a single focus within a shared workspace, the shell approach seems to help maintain the "sense of closeness" necessary in a remote collaboration tool [4],[5]. (Other researchers have also noted this benefit of enforced turn taking; see [7], for example). Thus, the shell-based interface shows great promise for supporting ongoing remote collaboration, with the benefits of synchronous interaction and a shared workspace intact.

The shell approach also performs well in instructional situations, where one collaborator is learning from another how to use an application program's environment. The major reason for this seems to be the identical "view" the collaborators have into the program environment: one collaborator can observe the other, and easily give a "helping hand" when necessary. It is not surprising that this type of identical view interface is good for instructional purposes. The Remote Takeover Software described in [11] and [12] provides a qualitatively similar interface, and is often used for these purposes. Thus, instructional use is one promising application for shell-based systems.

Finally, it should be noted that this type of interface seems to yield the most benefit when the information is inherently graphical in nature. Thus, the shell technique may hold great promise in fields such as engineering design,
where information is best represented graphically.

The observations made here have been drawn from a somewhat limited exploratory study. The tasks at hand were relatively short, and executed under somewhat controlled conditions. Additionally, the collaborators were extremely familiar with working together, and were operating in familiar program environments. However, it is felt that the observations that have been made are generally valid, and will hold true under a much broader set of circumstances.

6.2 Conclusions

The desire to facilitate interaction among geographically distributed workgroups has underscored the need for better computer-mediated collaboration tools. Development of effective collaboration tools which permit remote, synchronous interaction is a challenging task, especially in applications where real-time graphics displays are necessary. The purpose of this thesis is to develop a technique which permits existing single-user programs to be used as part of a multi-user collaborative interface.

6.2.1 Brief statement of the problem

A collaboration system allowing people in different locations to work synchronously and interactively would be valuable. In this work, a shell technique has been investigated which gives single-user applications a multi-user interface by linking together multiple workstations. When the shell is in operation, the linked workstations operate identically, thus providing a uniform
computer interface for a group of collaborators.

To investigate the technical requirements of a shell-based system, and to characterize the resulting multi-user interface, a prototype has been developed. One basic design requirement is that the system allow users to interact in the environment of existing application programs, while remaining independent of any particular program. A second requirement is that the resulting system be able to function over low bandwidth communications links, regardless of the amount of graphics output or database changes produced by an application program.

6.2.2 Methodology

A parallel processing approach is used, in which each workstation runs the application program independently. A shell program is set up to control interaction between the application program and the operating system at each computer. The shell distributes the user input at each system globally, and regulates parallel execution among workstations. Since each workstation does its own calculations, all graphics data and database changes are generated locally. This approach reduces the required data transmissions to an acceptable level for low-bandwidth communications. Furthermore, reliance on a particular software package is avoided by use of a shell.

One challenge in implementing such a system is capturing keyboard and pointing device input from each workstation and distributing it in a robust and consistent manner. A second implementation challenge is to control the system in a manner that ensures parallel execution among the independent workstations.
Properly addressing these challenges in the PC Compatible environment constitutes a major portion of this thesis.

6.2.3 Results

The issues involved in capturing user input at each workstation and distributing it globally are considered in Chapter 4. Two techniques for managing the keyboard are developed, and their relative merits for different applications are compared. Two methods for handling pointing device input are also discussed, and their relative merits compared.

A technique for regulating parallel execution among PC type computers under DOS is introduced in Chapter 2. The assumptions on which this technique is based and issues in its implementation under DOS are considered in Chapter 5. Chapter 5 also examines the practical limitations of this technique in the PC environment, both in terms of the hardware and application software used.

The ancillary issue of multitasking the DOS operating system is considered in Chapter 2. Chapter 3 considers the communication requirements of a shell-based system. Communications techniques and protocols are developed for both synchronized data exchanges and background data transfers. Finally, a preliminary evaluation of the prototype system and the shell technique in general has been presented in this chapter.

6.3 Suggestions for future research

While the groundwork for shell-based systems has been laid, there is much
work to be done. First, there remain many technical issues which need to be
resolved. The actual amount of communications reduction achieved by use of a
parallel processing approach needs to be quantified for different types of
programs, in order to determine the suitability of the technique in different
situations [30]. Furthermore, implementation of an execution shell system has
only been investigated for one operating system environment, and even there the
complete power of the environment has not been fully exploited (see [31], [32],
for example). Refinements of the shell technique under DOS, and investigations
of its characteristics under other operating systems are needed to better determine
its potential as a general approach.

Another remaining challenge is to more fully evaluate and characterize the
benefits and disadvantages of shell-based systems as collaborative tools. The
preliminary evaluation was carried out under very controlled circumstances, and
did not account for many factors which could impact the effectiveness of a shell-
based system in the real world. For example, the long term (more than one
interactive session) structure of cooperative activity has been completely neglected
as a factor in evaluating the prototype system, which clearly limits the
applicability of the results [33]. Furthermore, even the technical utility of such
a collaboration tool is not sufficient to guarantee its viability as a commercial
product [29], [34]. Thus, a much more rigorous evaluation of shell-based
systems is needed to establish their potential for commercial development.

Finally, the use of a shell technique to support collaborative work has not
been fully exploited. For example, [4] and [5] conclude that the process of
creating artifacts in a collaborative environment can be as important as the resulting artifacts themselves. Because a shell intercepts all user input into a program environment, it would be possible to use the shell to record collaborative interactions for review at a later date. It would likely be worthwhile to investigate the use of a shell for such purposes. Thus, while the groundwork has been laid, there are many opportunities for continuing research into the design and use of shell-based collaborative systems.
References


