

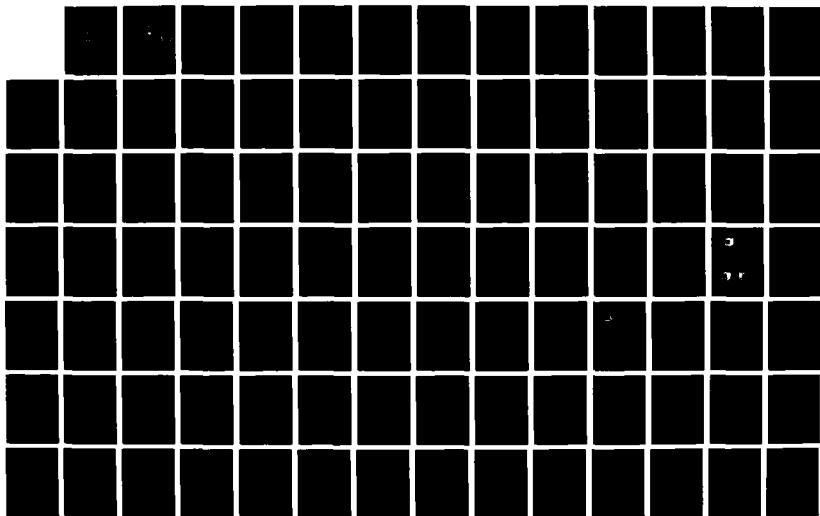
AD-A183 361

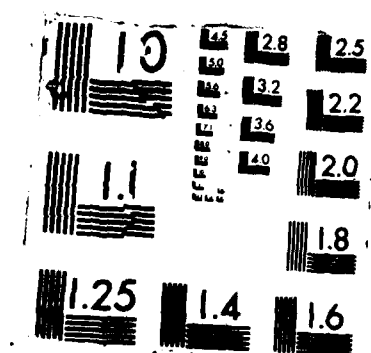
A DEMONSTRATION OF A TRUSTED COMPUTER INTERFACE BETWEEN 1/2
A MULTILEVEL SEC. (U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA G E RECTOR MAR 87

UNCLASSIFIED

F/G 25/5

NL





AD-A183 361

DTIC FILE COPY

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
AUG 17 1987
S D
A

THESIS

A DEMONSTRATION OF A TRUSTED COMPUTER
INTERFACE BETWEEN
A MULTILEVEL SECURE COMMAND AND CONTROL
SYSTEM AND
UNTRUSTED TACTICAL DATA SYSTEMS

by

George E. Rector, Jr.

March 1987

Thesis Advisor

Thomas J. Brown

Approved for public release; distribution is unlimited

87 8 13 039

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b RESTRICTIVE MARKINGS	
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b DECLASSIFICATION/DOWNGRADING SCHEDULE				
4 PERFORMING ORGANIZATION REPORT NUMBER(S)			5 MONITORING ORGANIZATION REPORT NUMBER(S)	
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b OFFICE SYMBOL (If applicable) Code 39	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000			7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	
8a NAME OF FUNDING/SPONSORING ORGANIZATION		8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)			10 SOURCE OF FUNDING NUMBERS	
			PROGRAM ELEMENT NO	PROJECT NO
11 TITLE (Include Security Classification) A DEMONSTRATION OF A TRUSTED COMPUTER INTERFACE BETWEEN A MULTILEVEL SECURE COMMAND AND CONTROL SYSTEM AND UNTRUSTED TACTICAL DATA SYSTEMS				
12 PERSONAL AUTHOR(S) George E. Rector, Jr.				
13a TYPE OF REPORT Master's Thesis		13b TIME COVERED FROM _____ TO _____		14 DATE OF REPORT (Year, Month, Day) 1987 March
15 PAGE COUNT 161				
16 SUPPLEMENTARY NOTATION				
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Computer Security, Multilevel Secure Computing, Trusted Systems, Tactical Data Systems,	
FIELD	GROUP	SUB-GROUP		
19 ABSTRACT (Continue on reverse if necessary and identify by block number) The task of this research is to demonstrate a multilevel secure interface between a system operating at multiple security levels and other untrusted systems operating at a single security level. Without a trusted interface device, these systems cannot be electronically connected. All communications between the systems must be done manually with all information transfer being reviewed by a security officer. Only releasable information is printed or stored in a removable medium and hand carried to the other system. In contrast, a trusted, multilevel secure guard can connect untrusted systems electronically and control the release of sensitive information. This task will demonstrate the ability of a multilevel trusted system to interface with untrusted systems operating at different levels of security.				
20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Thomas J. Brown			22b TELEPHONE (Include Area Code) (408) 646-2772	22c OFFICE SYMBOL Code 62Bb

Approved for public release; distribution is unlimited.

A Demonstration of A Trusted Computer Interface Between
A Multilevel Secure Command and Control System and
Untrusted Tactical Data Systems

by

George E. Rector, Jr.
Captain, United States Marine Corps
B.S., United States Naval Academy, 1976

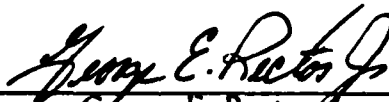
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS TECHNOLOGY
(Command, Control and Communications)

from the

NAVAL POSTGRADUATE SCHOOL
March 1987

Author:


George E. Rector, Jr.

Approved by:



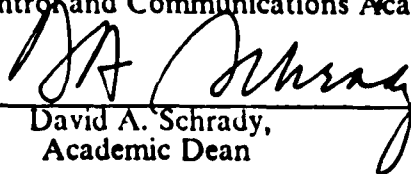
Thomas J. Brown, Thesis Advisor



Joseph S. Stewart, Second Reader



Michael G. Sovereign, Chairman,
Joint Command, Control and Communications Academic Group


David A. Schrady,
Academic Dean

ABSTRACT

The task of this research is to demonstrate a multilevel secure interface between a system operating at multiple security levels and other untrusted systems operating at a single security level. Without a trusted interface device, these systems cannot be electronically connected. All communications between the systems must be done manually with all information transfer being reviewed by a security officer. Only releasable information is printed or stored in a removable medium and hand carried to the other system. In contrast, a trusted, multilevel secure guard can connect untrusted systems electronically and control the release of sensitive information. This task will demonstrate the ability of a multilevel trusted system to interface with untrusted systems operating at different levels of security.

TABLE OF CONTENTS

I.	INTRODUCTION	10
A.	HISTORICAL PERSPECTIVE	10
B.	MARINE CORPS TACTICAL DATA SYSTEMS (TDS)	16
	1. General	16
	2. The Marine Tactical Command and Control (MTACC) Systems	16
	3. Marine Air Command and Control System	20
	4. Other Tactical Data Systems	21
	5. Interoperability	21
	6. Interface Requirements	22
	7. System Security	24
II.	BACKGROUND	27
A.	MULTILEVEL SECURE COMPUTING SYSTEMS	27
	1. Trusted Computer System Requirements	27
	2. The Security Kernel and Guard Technology	31
	3. Data Encryption	34
	4. Summary	36
B.	GEMINI TRUSTED MULTIPLE MICROCOMPUTER BASE	38
	1. Description of Gemini System Components	38
	2. Gemini Resource Management Overview	39
	3. Gemini Secure Operating System (GEMSOS) Architecture	40
	4. Application Development Environment	43
	5. Summary	48
III.	DEVELOPMENT OF A MULTILEVEL SECURE INTERFACE	49
A.	GENERAL	49
	1. Objectives	49
	2. Design Constraints	51

3.	Summary of Design Decisions	51
B.	SYSTEM IMPLEMENTATION	53
1.	Hardware Components	53
2.	Application Program Format	53
C.	SYSTEM SOFTWARE DESIGN	55
1.	Application Segment Development	55
2.	Process Synchronization	58
D.	DESIGN SUMMARY	58
IV.	DISCUSSION OF RESULTS	61
A.	SYSTEM OPERATION	61
B.	DEVELOPMENTAL PHASES	62
1.	JINTACCS Automated Message Preparation System Implementation	62
2.	Intersegment Linkage of Multiple Language Applications	63
3.	Communications	65
4.	Demonstration of GEMSOS Security Mechanisms	67
C.	SYSTEM TESTING	67
D.	OBSERVATIONS AND LESSONS LEARNED	68
1.	Applications Development with GEMSOS	68
2.	Development of Applications Using ADA	69
3.	Computer Security for Marine Tactical Data Systems	70
4.	Integrated Security Requirements	71
E.	SUMMARY	73
V.	CONCLUSIONS	74
A.	GENERAL	74
B.	RECOMMENDATIONS FOR FURTHER STUDY	75
1.	Integration of JAMPS into GEMSOS	75
2.	Implementation of a Polling Scheme	75
3.	Modern Programming Techniques	76
APPENDIX A:	GLOSSARY	77
APPENDIX B:	LIST OF ACRONYMS AND ABBREVIATIONS	81

APPENDIX C:	INTERFACE USERS GUIDE	83
APPENDIX D:	SYSTEM MANAGER PROGRAM LISTING	86
APPENDIX E:	GUARD.KMD LINKING FILE	121
APPENDIX F:	GUARD-CON.ZLI INCLUDE FILE OF CONSTANT DECLARATIONS	122
APPENDIX G:	GUARD-TYP.ZLI TYPE DECLARATION INCLUDE FILE	123
APPENDIX H:	TDS1 TERMINAL UTILITY PROGRAM LISTING (JANUS ADA)	124
APPENDIX I:	DEFS.LIB LIBRARY DEFINITION FILE	130
APPENDIX J:	DEFS.PKG PACKAGE BODY FILE USED TO SUPPORT DEFS.LIB	132
APPENDIX K:	TDS2 TERMINAL UTILITY PROGRAM LISTING	139
APPENDIX L:	TDS2.KMD LINKING FILE FOR TDS2.PAS	155
APPENDIX M:	GUARD.SSB SYSTEM GENERATION SUBMIT FILE	156
LIST OF REFERENCES		157
BIBLIOGRAPHY		159
INITIAL DISTRIBUTION LIST		160

LIST OF TABLES

1. TDS INTERFACE REQUIREMENTS AND INTERFACE LEVELS 23
2. TDS COMMUNICATION INTERFACES 25

LIST OF FIGURES

1.1	Marine Corps Tactical Data Systems	17
2.1	ECB Mode of DES Encryption	36
2.2	CBC Mode of DES Encryption	37
2.3	CFB Mode of DES Encryption	38
2.4	Compromise and Integrity Properties	41
2.5	Single and Multilevel Device Properties	43
2.6	GEMSOS Hierarchical Structure	45
2.7	GEMSOS Hierarchical Structure Including an Application	46
3.1	Proposed System Design	52
3.2	Final Hardware Diagram	54
3.3	System Manager Flow Chart	57
3.4	Terminal Utility Flow Chart	59
4.1	Revised Hardware Configuration	64
4.2	Computer System Vulnerabilities	72

I. INTRODUCTION

As automation increases and our reliance on computer systems grows, it becomes increasingly important to ensure that the information entrusted to these systems is protected. Techniques to address information protection can be as simple as procedural controls, or as complicated as controls embedded in the hardware and software of the computer system itself. Each technique addresses a particular information protection problem and assumes that other techniques are available to solve problems in other areas. This thesis addresses some of the problems involved with computer security, specifically, computer security of tactical data systems. The goal of this thesis is to provide an introduction to the concept of computer security, to focus on a particular area where computer security is vital--Marine tactical data systems, and to demonstrate how a secure guard may be added to Marine tactical data systems as a "first step" toward designing a multilevel secure computer system to protect information vital to command and control of Marine forces.

Chapter I of this thesis provides an overview of computer security and introduces Marine tactical data systems, their interoperability, and their interface requirements. Chapter II provides a basic understanding of multilevel computing systems, specifically, the Gemini Trusted Multiple Microcomputer System. Chapter III describes the development of a demonstration of the Gemini Trusted Computing Base (TCB) used as a guard between Marine tactical data systems operating at multiple levels of security. Chapter IV discusses the implementation of the demonstration, its testing and "lessons learned". Finally Chapter V provides conclusions drawn from this research and recommendations for further study.

A. HISTORICAL PERSPECTIVE

Computers are here to stay. They are permeating every facet of society from games to managing information critical to our national security. Information has become a strategic resource. The availability of computers has moved us from an industrial society to an information society. Our C³I posture is critically affected by and dependent on computers. Their speed and unfailing accuracy make them well suited to the massive information handling tasks in battle management for:

- Shared information storage, retrieval and dissemination
- Rapid and common processing systems

- Efficient and reliable communications process control [Ref. 1: p.271].

Relying on computers to carry out timely and reliable information transfer raises the question, *can they be trusted?* Trust and reliability in computer software and hardware are not necessarily synonymous. First, what vulnerabilities must we guard against and next, what is needed to bridge the gap between the plain vanilla computer and the *trusted* computer?

Rapid advances of hardware and software technologies in microelectronics, computers, network systems and man-machine interfaces are making major changes in today's C³I architecture approaches. Modern C³I systems will be implemented by new architectures consisting of a large number of networks, computers, processors and input/output devices. However, many of the computer and network systems used to support modern C³I technologies must handle sensitive information and work in a completely secure environment. Information used with today's C³I systems must be protected.

To better understand the role of computers in information protection, let us look at how our evolving use of computers has brought with it computer security problems. This will help determine what can and cannot be done to protect our newest strategic resource.

Until the mid-1950's, computers were commonly dedicated to one user at a time and security was of minor concern. The user either worked on his own behalf or as a programmer for someone else. The computer power was limited. With reasonable planning, the user kept the machine busy for his period of use. The jobs were typically processing of numerical data which required only a limited amount of software and data. The user brought with him all the data (card deck) and no one else could affect the machine while he used it. If he had sensitive data, he could easily purge the small bit of information in the machine and take all of his data and results with him when he finished. Thus, with no sharing of the machine and no sharing of data, the user was in control of his own security. [Ref. 2]

Later, in the 1960's more powerful and much more expensive computers could not be dedicated to just one user. The human was just too slow to efficiently employ the machine and usage expanded. Software packages called *operating systems* or *monitors* evolved and computers were shared by multiple users. In this mode, the machine was under the physical control of a computer operator, not the user. Most common and useful operating systems employ *multiprogramming* or *timesharing* so that

several jobs may run simultaneously. Regardless of the operating system particulars, the computer is in control of its resources, not the user. In this environment, the nature of computer security becomes quite clear. The operating system software is more privileged in some sense than the user. This poses no problem as long as the operating system is *trusted*. It did not take long to discover that a malicious user could easily penetrate the operating system and cause it to share its privileges. [Ref. 2]

An answer to this problem of non-existent internal security is to eliminate any user who is not authorized access to the information. This method of security can be effective, but it has two disadvantages. It can be quite expensive, since it limits sharing of the computer's resources to a common need to know. It can also encourage imprudent risks because of the temptation to increase sharing by treating all users as honest when they may have no need to know or may even be hostile. The distinctive characteristic of a shared system is that security can be provided by isolating the users into compatible groups that share machine resources.

Since the mid-1960's, computers have been increasingly used for this purpose in information management. The principle capability of these systems is access control vice processing. These controls can be as simple as distinguishing whether a user can read only or both read and write a file to more complex controls over access--such as those implicit in the military classification levels.

For example, wargaming centers often have a need to provide war games at several levels of classification using a single software system and a large classified data base. Frequently the users, who are expected to participate through hands-on use of the computer, have no clearance or clearance at a level lower than might be required. This forces downloading of the sensitive portions of the data base, a disruptive practice in a wargaming center. The use of a *trusted* guard could control the access to the data base and allow operation of games with various levels of classification using a single, fixed classified data base.

Information systems and networks are rapidly expanding in the private sector as well as the military. The available isolation techniques are not workable--since the need is to provide controlled (shared) access via the computer. This means that we have no alternative but to develop internal controls for the computer itself--new, more powerful, and *trusted* operating systems.

From this perspective, we can see that there are basically two kinds of responses; limit the opportunity to do harm and, in doing so, we reduce the vulnerabilities; and stimulate industry to apply emerging technology to counter these vulnerabilities.

A fundamental step in shoring up the technical weakness of today's computers is defining a clear, well formulated policy to influence secure computer system designs. The computer designer must be told exactly what policy the system must enforce. The computer cannot make a judgment regarding the user's request for access to stored information. It can only grant or deny access based on the authority that it has been given. Thus, what is needed, as a basis for any trusted system, is a policy that articulates well defined authorizations to information and computer resources.

There are many documents which attempt to define requirements for trusted computer systems. They have been generated at all levels of the government, and in some cases are in conflict with each other. In 1983 an attempt was made within the Department of Defense (DOD) to consolidate these documents as well as other information concerning trusted computer systems. The goal was to create a single source document which would define guidelines for the development and testing of new systems. The result was entitled "DOD Trusted Computer System Evaluation Criteria," commonly referred to as the 'Orange Book' [Ref. 3]. Published in 1983, it contains definitions and information essential to understanding trusted computer systems. The Orange Book goes into extensive detail concerning the implementation of automated data processing (ADP) security systems. As described in the Orange Book there is mandatory security which is defined as:

Security policies defined for systems that are used to process classified or other specifically categorized sensitive information must include provisions for the enforcement of mandatory access control rules. That is, they must include a set of rules for controlling access based directly on a comparison of the individual's clearance or authorization for the information and the classification or sensitivity designation of the information being sought, and indirectly on considerations of physical and other environmental factors of control. The mandatory access control rules must accurately reflect the laws, regulations, and general policies from which they are derived. [Ref. 3: p.72]

As the name implies, mandatory security policy is a strict limitation of access based on access level, which is determined by the user's security clearance. This policy can not be changed and represents the foundation for the second type of security policy. Discretionary security policy is a subset of mandatory security policy which represents a further restriction of access to information based on a user's "need to know" the information. The control objective for discretionary security is:

Security policies that are defined for systems that are used to process classified or other sensitive information must include provisions for the enforcement of

discretionary access rules. That is, they must include a consistent set of rules for controlling and limiting access based on identified individuals who have been determined to have a need-to-know for the information. [Ref. 3: p.73]

This type of security is a definite asset in a research and development environment. For example, when developing combat system software, a project manager may have teams developing several modules simultaneously on the same system. Although the modules may be of the same classification level, the manager may want to limit each team's access to the module on which they are working. This would be accomplished by establishing a discretionary security policy.

Traditional attacks on security systems have involved compromise of keywords which would allow unauthorized access to a system. This threat can largely be eliminated by physical means: changing keywords, multi-level identification, and restricting access to the system. A more subtle attack, and potentially more dangerous threat is the establishment of a covert channel in the system. A covert channel is defined as "any communications channel that can be exploited by a process to transfer information in a manner that violates the system security policy." [Ref. 3: p.79] In a multi-level computer system, the presence of a covert channel can be exploited to gain unauthorized access to information without alerting security mechanisms. Covert channels will be discussed further as a design consideration for a multi-level secure communications system.

One of the most difficult tasks in developing trusted computer systems is determining test criteria to evaluate their performance. As the security level is increased, the test criteria become more stringent and detailed. When operating in a network environment, the problem is compounded by requiring communications security between the trusted computer systems as well.

Although such an explicit policy is necessary, it is not sufficient; the problem remains the effectiveness of the operating system. One technology that can provide penetration-proof controls is a *security kernel*. A security kernel is essentially a small subset of an operating system and its associated hardware that will guarantee internal enforcement of an explicit security policy, independent of the rest of the operating system or user programs.

The security kernel is a breakthrough that has transformed the designer's game of wits with penetrators into a methodical design process. Since early kernels were first introduced in 1972, further research has demonstrated their feasibility, functionality

and broad certifiability. With such technology available, vendors are now beginning to offer kernel based operating systems. Its successful implementation will enable computers to thwart subversion attacks aimed at unauthorized access, disclosure, modification or disruption of service of a computer-based C³I system. This entails not only converting policy objectives into technical standards based on a mathematically sound means to specify security requirements, but also the means to verify their design and implementation and to maintain continuous control upon acceptance.

Computer development in the past decade has made considerable progress in meeting multiple level security requirements. There have been several approaches to these requirements. Security kernel technology has been more successful and is the main basis for practical computer security products. For example, under Navy sponsorship, Honeywell has developed the SCOMP (Secure Communications Processor) using a security kernel. Under Air Force sponsorship, IBM/ITT is about to deliver the SACDIN (Strategic Air Command Digital Network) processor which also uses a security kernel. Both the SCOMP and SACDIN secure processors use a single minicomputer.

With increasing recognition of the security problems in both computer and network systems and recent security policy developments in the Department of Defense (DOD), it would be useful for a research effort to start at the Naval Postgraduate School for the support of DOD wide security requirements in C³I developments.

A new distributed C³I testbed is being developed using the *Gemini* trusted multiple microcomputer base to support some of the C³I related research topics at the Naval Postgraduate School. There are three ways to address the security problems that form the foundation for this research.

- For untrusted computers and/or networks which already have been developed and are in operation today, such as the MILNET, new multilevel secure guard devices are potential candidates.
- For networks which are still being developed such as DDN, new trusted interface units could be developed to support multilevel security requirements.
- For new systems which have not been designed, new trusted computer system technologies can be included in the system design to support new multilevel secure C³I systems satisfying the DOD Computer Security Center's criteria.

This thesis investigates the use of a trusted computing base (TCB) to act as a multilevel secure interface between the tactical data systems (TDS) of the Marine Corps. This thesis proposes that the Marine Corps Tactical Intelligence Management System (TIMS) act as a multilevel, trusted, systems manager with the ability to

communicate with untrusted systems operating at single levels of security such as the Tactical Combat Operations (TCO) System, the Advanced Tactical Air Control Central (ATACC), the Marine Integrated Fire and Air Support System (MIFASS), and the Tactical Air Operations Module (TAOM). Therefore, TIMS would act as the primary interface and the center of information for the Marine Corps tactical data systems.

B. MARINE CORPS TACTICAL DATA SYSTEMS (TDS)

1. General

The automated systems discussed in this section are those currently existing or projected to support the exchange of tactical information for the Marine Corps. Each will be described below. More detailed descriptions may be found in current Marine Corps documentation such as the U. S. Marine Command and Control Master Plan (C²MP) [Ref. 4] and systems requirements documents and specifications. The Marine Corps tactical data systems are a conceptual association of command and control systems to support tactical operations. It consists of functionally oriented, tactical and training systems, using, where feasible, common equipment, operational procedures, data bases, and design philosophy. These systems, where appropriate, will interoperate through a common communication system as depicted in Figure 1.1

2. The Marine Tactical Command and Control (MTACC) Systems

In order to receive, process, store, display and forward the large quantities of information that will be available on the modern battlefield in a useful and timely manner, selective application of automation within operations centers is required to support the command and control of Marine Air-Ground Task Forces (MAGTF). A semi-automated, hardened, secure, and transportable command and control system, capable of interfacing with existing Marine communications systems, is required in operations centers to provide real-time data input/output, storage, display, retrieval and processing support to the commander in the performance of his operations, planning, and intelligence functions. The Marine Tactical Command and Control System (MTACCS) is a conceptual association of command and control systems to support tactical operations. It consists of six functionally oriented tactical and training systems, using, where feasible, common equipment, operational procedures, data bases, and design philosophy, and, where appropriate, interoperating through a common communications system.

The six systems included in the MTACCS concept are:

- Marine Integrated Fire and Air Support System (MIFASS)

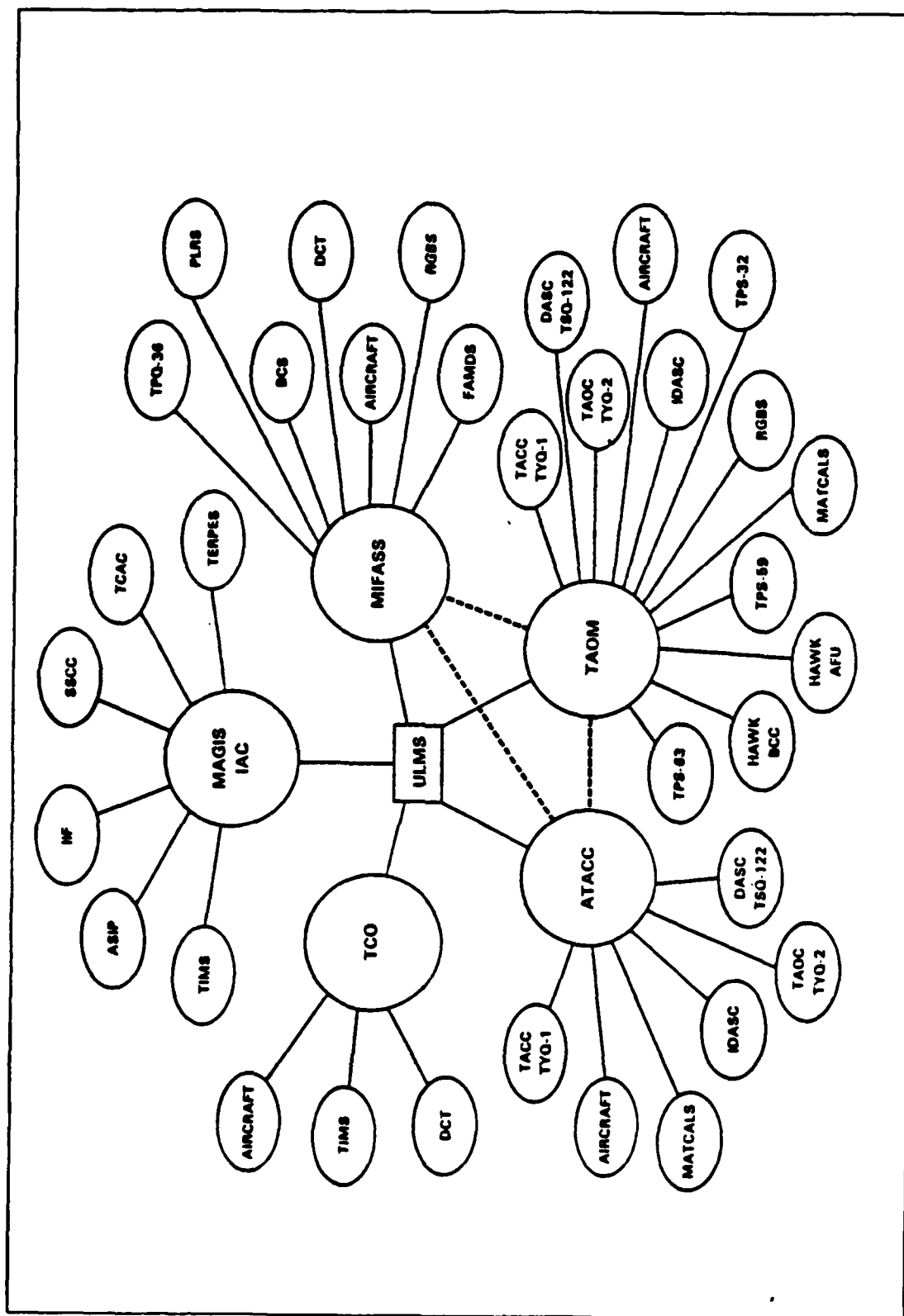


Figure 1.1 Marine Corps Tactical Data Systems.

- Tactical Combat Operations (TCO) System
- Tactical Air Operations Module (TAOM)
- Marine Air-Ground Intelligence System (MAGIS)
- Position Location Reporting System (PLRS)
- Tactical Warfare Simulation Evaluation and Analysis System (TWSEAS)

The following paragraphs from the C²MP provide an overview of how each MTACC system contributes to the whole configuration. This will be accomplished by presenting a summary description of their design objectives and by describing functional boundaries or limits for these systems.

a. Marine Integrated Fire and Air Support System (MIFASS)

MIFASS performs the critical function of coordination and control of supporting arms. This is accomplished by integrating, directing, and coordinating artillery, mortar, naval gunfire, and air support. Real-time friendly position location reporting such as that provided by PLRS and the TAOC is essential to the effective operation of MIFASS so that its rapid response capabilities can be fully realized. MIFASS consists of suites of modular microprocessing and display equipment and software tailored to the functions performed at a given level of command. It is characterized by the commonality of equipment with other MTACC systems, especially TCO with which it may share much of its equipment at some echelons.

b. Tactical Combat Operations (TCO) System

As the title implies, TCO is the system which will provide the commander his principal assistance in combat operations planning, monitoring, and coordinating. TCO will provide commanders with the capability to accomplish the planning and direction of combat operations. Modular microprocessing and display equipment, tailored for the using command, will be provided to MAGTF commanders at all levels from the Marine Amphibious Force (MAF) down to battalion and squadron. Digital Communications Terminals (DCTs), capable of transmitting and receiving combat-essential information to and from battalion and squadron level, will be provided below these levels. This will permit, in addition to other functions, the transmission of forward-echelon combat information via TCO channels for entry into the Marine Air-Ground Intelligence System (MAGIS). TCO is expected to become one of the principal means by which information is passed to MAGIS. Conversely, intelligence and information produced by MAGIS will be provided to echelons below MAF via TCO. Friendly position location information (PLI), such as that available from the

Position Location Reporting System (PLRS), is essential for TCO system operations. TCO will be used to support staff operations and intelligence functions at all echelons of command. By receiving, via the Landing Force Integrated Communications System (LFICS) and Joint Tactical Information Distribution System (JTIDS), and displaying selected data from MIFASS, MAGIS, PLRS, and TAOM, TCO will provide the focal point at which the commander can obtain his operational information and disseminate command decisions.

The TCO mission is to provide an accurate data input, storage, information retrieval, and processing system for the real-time support of ground, aviation and MAGTF operational functions and those intelligence functions below MAF level (not supported by MAGIS/IAC) within operations centers.

c. Tactical Air Operations Module (TAOM)

The TAOM will provide the Tactical Air Commander (TAC) with the means to monitor, coordinate, and control intercept aircraft and surface-to-air missile systems, including Forward Area Air Defense (FAAD) weapons, and to give enroute traffic control assistance to aircraft within the MAGTF area of responsibility. It is a modular system which can be tailored to meet the requirements of any size MAGTF from a Marine Amphibious Force (MAF) to a Marine Amphibious Unit (MAU). In order to augment its organic radar coverage capabilities for effective execution of the TAOM air defense mission, the TAOM system exchanges TADIL-B position location information with the senior MIFASS-supported FASC.

d. Marine Air-Ground Intelligence System (MAGIS)

The primary function of MAGIS is the processing of information concerning the enemy, weather, and terrain into timely, accurate, detailed, all-source intelligence in support of the tactical commander. The AN/TYQ-19(V) IAC, is a tactical intelligence processing facility which will serve as the heart of MAGIS. The IAC will accept information from the Imagery Interpretation segment, Tactical Electronic Reconnaissance Processing and Evaluation System, Integrated Signals Intelligence System, MAF Reconnaissance and Surveillance Center, division and wing intelligence centers, TCO, and other sources for processing into intelligence. Normally, one IAC is assigned to each MAF and may be retained at the MAF headquarters or assigned to the Air Combat Element (ACE), or the Ground Combat Element (GCE) Headquarters. The automated intelligence functions will be supported by the Tactical Intelligence Management System (TIMS). TIMS will handle intelligence information

with a classification above secret and TCO will handle intelligence information with a classification of secret or lower.

e. Position Location Reporting System (PLRS)

PLRS performs the function of position location reporting of PLRS-equipped units, vehicles, and aircraft essential to TCO and MIFASS maneuver and firepower coordination and control. Its capability of locating friendly units, vehicles, and aircraft brings the element of real-time, accurate, current PLI to the battlefield commander.

f. Tactical Warfare Simulation, Evaluation, and Analysis System (TWSEAS)

Tactical Warfare Simulation, Evaluation, and Analysis System. TWSEAS is a tactical command and control system, with multiple capabilities for supporting virtually all types of tactical exercises conducted in the Marine Corps. TWSEAS is not a combat system; its role is training support. TWSEAS is a mobile facility designed and configured for sustained operations in the field with tactical units during training exercises. TWSEAS can function as a control center for an exercise, or be used to support staff exercises, such as map maneuvers, which are not necessarily conducted in the field.

3. Marine Air Command and Control System

MACCS comprises those agencies that support the air commander in the exercise of centralized coordination and supervision of air operations while decentralizing control to subordinate agencies. The systems supporting MACCS agencies are described in more detail below. Their general concept of employment is discussed in the C²MP.

a. Tactical Air Operations Module (TAOM)

TAOM is a transportable, modular, software intensive, automated element of the aviation C² system. It provides the primary link between MTACCS and MACCS. It is therefore listed as a subsystem for both. The TAOM is capable of controlling and coordinating the employment of air defense weapons in support of any size MAGTF.

b. Advanced Tactical Air Command Central (ATACC)

ATACC is a computer-supported facility which provides the TAC with the resources needed for planning and directing the air battle. It is the senior Marine air command and control agency from which the TAC exercises overall supervision, coordination and general control of tactical air operations. Subordinate agencies from

which the TAC exercises these functions include the TAOC, DASC, Marine air traffic control squadrons, and terminal control action elements.

c. Marine Air Traffic Control and Landing System (MATCALS)

MATCALS provides semi-automated and automated capabilities for aircraft surveillance, identification, tracking, vectoring, track hand-over, and cross-telling. This system provides automated tracking based on correlation of radar; identification, friend or foe; and data-link replies. The system provides simultaneous landing control. It may control up to six aircraft with a routinely sustained landing rate of one aircraft per minute, and has the technical capability of increasing the rate to two aircraft per minute.

d. Other MACC Systems

Other systems to be included in the Marine Air Command and Control System include the Marine Remote Area Approach and Landing System (MRAALS), Tactical Data Communications Central (AN/TYQ-3), Direct Air Support Central (DASC), Hawk Missile System, and a host of radar sets.

4. Other Tactical Data Systems

Other tactical data systems include:

- Battery Computer System (BCS), AN/GYK-29.
- Radar Set, Firefinder, AN/TPQ-36.
- Field Artillery Meteorological Data System (FAMDS), AN/TMQ-31.
- Modular Universal Laser Equipment (MULE), AN/PAW-3.

Additional detail on each of the tactical data systems listed may be found in the C²MP [Ref. 4] and other reference materials associated with individual programs.

5. Interoperability

Exchange of information between agencies participating in a tactical (amphibious) operation is critical to the success of the overall mission. Equipment in operations centers will provide the facilities required to achieve necessary interoperability of information exchange with tactical data systems and the systems of other services to support commanders at all echelons. Data exchange standards will be in accord with the provisions of the U. S. Marine Corps Technical Interface Design Plan (TIDP) and Technical Interface Concept (TIC) [Ref. 5: p. 1-5]. The interface standards for data exchange between Marine tactical data systems and other service systems will be in accord with the provisions established by the Joint Interoperability of Tactical Command and Control Systems (JINTACCS) plan.

6. Interface Requirements

In describing interoperability between systems, the levels of interface are categorized into three levels: manual, semi-automated, and automated. In Table 1 from [Ref. 6: p. 4-4], the level of interface is defined for both sides of each interface. Where the same level of interface exists at the systems on both sides of the interface, a single designator appears in the matrix at the intersection of the column and row for the two systems. For example, at the intersection of the row opposite "TAOM" and the column headed "ATACC", the letter "S" appears, indicating that each system has a semi-automated interface with the other system. Where the levels of interface are not the same on both sides of the interface, the letter in the upper side of the box at the intersection indicates the level of interface of the system at the head of the column. The level of interface for the system named on the left side of the row is indicated by the letter in the lower half of the box.

Table 2 from the Technical Interface Concept for Marine Tactical Systems, [Ref. 6: p. 4-6], depicts the general character of the interface. The communication medium, requirements for controllers, and the use of TADILs are portrayed. The actual physical interface in each case is provided by either or both of the following means:

- Cable; e.g., fiber optic cable, 26-pair cable, and wire.
- Radio; e.g., single channel and multichannel.

An additional element of an interface may be an interface device or interface controller. Such a device is necessary where the interoperating systems use incompatible circuit and/or message standards or operate at different levels of security. Examples of interfaces requiring interface devices are:

- MIFASS - PLRS
- MIFASS - TACFIRE
- TCO - MAGIS/IAC
- TCO - ATACC
- TCO - PLRS
- TCO - MIFASS

Tactical Data Interface Links (TADIL) are used to achieve interoperability between Marine Corps TDSs and other service systems, and among some Marine Corps systems. A TADIL is a JCS-approved, standardized, communications link suitable for transmission of digital data. TADILs are characterized by standard

TABLE 1
TDS INTERFACE REQUIREMENTS AND INTERFACE LEVELS

	PLRS	MIFASS	TCO	DCT	AN/TYQ-1	TAOM	AN/TSQ-122	IDASC	RGDBS	MATCALS	HAWK BCC	HAWK AFU	BCS	FAMDS	AN/TPQ-38	MULE	IAC	TERPES	IIF	ASIP	TIMS	TCAC	CCO
PLRS	A	A																					
MIFASS	A	A	A	S		A			M				A	S	S								
TCO		A	A	S													S				M		
DCT		S	S	M				M					S			S							
AN/TYQ-1					S	S	M	M		M													
ATACC		S	S		S	S		S		S							S	S					
AN/TYQ-2					S	S	M	M	M	M	A	A											
TAOM		A			S	S	M	M	M	S	A	A											
AN/TSQ-122					M	M			M	M													
IDASC				M	M	M		M	M	M													
RGDBS		M				M	M	M															
MATCALS					M	S	M	M															
HAWK BCC						A																	
HAWK AFU						A																	
BCS		A		S									S	S	S								
FAMDS		S											S										
AN/TPQ-38		S											S										
MULE				S																			
IAC			S														S	S	S	S	M	S	
TERPES																	S				M		
IIF																	S				M		
ASIP																	S				M		
TIMS			M														M	M	M	M	M	M	
TCAC																	S				M	S	A
CCO																						A	S
AIRCRAFT		M	M		S	A	M	M	A	A								A					
AN/TPS-32						A																	
AN/TPS-39						A																	
AN/TPS-63						A																	

M: Manual

S: Semi-automatic

A: Automatic

message formats and transmission characteristics. TADILs appearing in table 2 are listed below:

- TADIL A (NATO Link 11).
- TADIL B.
- TADIL C (NATO Link 4A).
- TADIL J (NATO Link 16).
- ATDL 1.
- NATO Link 1.

7. System Security

The protection of tactical data systems is vital to landing force safety and imposes stringent requirements for system security. The primary elements which implement this protection are TDS equipment and program features, physical security measures, procedural features, and access controls. These features provide the in-depth, mutually supporting protection of classified material as well as system integrity.

TABLE 2
TDS COMMUNICATION INTERFACES

	PLRS	MIFASS	TCO	DCT	AN/TYQ-1	TAOM	AN/TSQ-122	IDASC	RGDBS	MATCALS	HAWK BCC	HAWK AFU	BCS	FAMDS	AN/TPQ-36	MULE	IAC	TERPES	IF	ASIP	TIMS	TCAC	CCO
PLRS	2	5																					
MIFASS	5	3	3	3		8			3				5	5	5								
TCO		3	3	3													3				3		
DCT		3	3	3				3					3			1							
AN/TYQ-1					3	8	3	3		3													
ATACC		8	3		8	8		3		8							3	8					
AN/TYQ-2					8	3	3	3	3	3	13	13											
TAOM		8			8	3	3	3	3	8	13	13											
AN/TSQ-122					3	3			3	3													
IDASC				3	3	3		3	3	3													
RGDBS		3				3	3	3															
MATCALS					3	8	3	3															
HAWK BCC						13																	
HAWK AFU						13																	
BCS		5		3									3	3	3								
FAMDS		5											3										
AN/TPQ-36		5											3										
MULE				1																			
IAC			3														3	3	3	3	3	3	
TERPES																	3				3		
IF																	3				3		
ASIP																	3				3		
TIMS			3														3	3	3	3	3	3	
TCAC																	3				3	3	3
CCO																						3	3
AIRCRAFT		2	2		6	12	2	2	2	9								10					
AN/TPS-32						3																	
AN/TPS-39						3																	
AN/TPS-63						3																	

TABLE 2
TDS COMMUNICATION INTERFACES (CONT'D.)

LEGEND

1. Cable; e.g., fiber optic cable, 26-pair cable and wire
2. Radio, e.g., single channel and multichannel
3. Cable and radio
4. Cable with an interface controller
5. Cable and radio with interface controller
6. Radio with TADIL A
7. Radio with TADIL B
8. Cable and radio with TADIL B
9. Radio with TADIL C
10. Radio with TADIL B
11. Radio with TADILs A and J
12. Radio with TADILs C and J
13. Cable and radio with ATDL 1
14. Cable and radio with NATO Link 1

II. BACKGROUND

A. MULTILEVEL SECURE COMPUTING SYSTEMS

1. Trusted Computer System Requirements

As computer systems have become more sophisticated and widespread in their applications, the need to protect their integrity has grown. Protection is often thought of as a supplement to multiprogramming operating systems, so that untrustworthy users might share a common logical space, such as a directory, or a common physical space such as memory. Modern security concepts have evolved to increase the reliability of any complex system which makes use of shared resources.

In general, secure systems will control, through the use of specific security features, access to information [Ref. 3: p. 3]. Protection refers to the mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specification of the controls to be imposed, together with some means of enforcement. There are several incentives for protection. Most obvious is the need to prevent mischievous, intentional violation of an access restriction by a system user. Of equal importance is the need to ensure that each program component active in a system uses the system resources only in ways consistent with stated policies. This is an absolute requirement for a trusted system.

A computer system may be viewed as a collection of processes and resources. To ensure the orderly and efficient operation of a system, the processes are subjected to policies that govern the use of resources. The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies may be established in a number of ways. Some are fixed in the design of a system, while others are formulated by the management of a system. A protection system must have the flexibility to enforce a variety of policies that can be declared to it.

Policies for resource use may vary, depending on their application, and they may be subject to change over time. For these reasons, computer security can not be considered solely as a matter of concern to operating system designers. It should be available as a tool for the applications programmer, so that resources created and

supported by an application may be guarded against misuse. There must be protection mechanisms provided so that applications designers may use them in designing their own protection software.

One important principle is the separation of policy from mechanism. Mechanisms determine how to do something while policies determine what will be done. This separation is very important for flexibility. Policies are likely to change from time to time or place to place, whereas, mechanisms should be more general, only requiring modification of certain parameters or minor change.

A computer system is a collection of processes (or subjects) and objects. Objects may be hardware objects (such as the cpu, memory segments, printers, card readers, or tape drives), and software objects (such as files, programs, and semaphores). Each object has a unique label that differentiates it from all other objects in the system and can be accessed only through well-defined and meaningful operations.

Obviously, a process should be allowed to access only those resources it has been authorized to access. Furthermore, at any time it should be able to access only those resources that it currently needs to complete its assigned task. This requirement is commonly referred to as the need-to-know principle or discretionary access control. This principle is useful in limiting the damage that a faulty process can cause a system.

These access controls shall be capable of specifying, for each named object, a list of named individuals and a list of groups of named individuals with their respective modes of access to that object. Furthermore, for each such named object, it shall be possible to specify a list of named individuals and a list of groups of named individuals for which no access to the object is to be given.
[Ref. 3: p. 43]

This introduces the concept of a protection domain. A process operates within a protection domain, which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object. The ability to execute an operation on an object is an access right. A domain is a collection of access rights, each of which is an ordered pair < object-name, rights-set > . For example, if domain A has the access right < file X, read,write > , then a process executing in domain A can both read and write file X, it cannot perform any other operation on that object.

A domain is an abstract concept which may be realized in a number of ways:

- A domain may be defined for each user. The set of objects which can be accessed depends on the identity of the user.

- A process may be a domain. Each object which can be accessed by that process is described. Also each operation which may be performed is defined.
- Each procedure may be a domain. Thus, each object which may be accessed by that procedure is defined.

The degree of protection provided in existing computer systems has usually been achieved through a security kernel, a guard which inspects and validates each attempt to access a protected resource. Since extensive access validation is potentially a source of considerable overhead, it must either be given hardware support to reduce cost, or one must accept that the system may be designed with lower goals of protection. It is difficult to satisfy all of the goals if the flexibility to implement various security policies is restricted by the support mechanisms provided.

As operating systems have become more complex, and particularly as they have attempted to provide higher-level user interfaces, the goals of protection have become much more refined. In this refinement, we find that designers have drawn heavily on ideas that originated in programming languages, especially on concepts such as abstraction, layering, virtualization, and information hiding. Protection systems are now concerned not only with the identity of a resource to which access is attempted, but also with the functional nature of the access.

Policies for resource use may also vary, depending on the application. For these reasons, protection can not be considered solely as a matter of concern to system designers, it must also be available as a tool for the applications designer in order that applications may be guarded against tampering or the influence of an error.

There are two documents which provide the basis for evaluation of the effectiveness of security controls built into computer systems or networks. These documents are distributed by the National Computer Security Center located at Fort Meade, Maryland. They are the Department of Defense Trusted Computer System Evaluation Criteria (CSC-STD-001-83) dated 15 August 1983 and the Department of Defense Trusted Network Evaluation Criteria (draft) dated 29 July 1985. These two documents form the foundation for all acceptable secure computer systems.

To call any computer system "secure," a set of requirements must be established. There are six fundamental requirements presented in [Ref. 3] as the absolute essentials for obtaining a secure system. Four of these requirements deal with the means to be provided to control access and two deal with how one can obtain credible assurances that this is accomplished in a trusted computer system.

- Requirement one -- SECURITY POLICY -- There must be an explicit and well defined security policy enforced by the system. Given identified subjects and

objects, there must be a set of rules that are used by the system to determine whether a given subject can be permitted to gain access to a specified object.

- Requirement two -- MARKING -- Access control labels must be associated with objects. In order to control access to information stored in a computer, according to rules of a mandatory security policy, it must be possible to mark every object with a label that reliably identifies the object's sensitivity level, and/or modes of access accorded those subjects who may potentially access the object.
- Requirement three -- IDENTIFICATION -- Individual subjects must be identified. Each access to information must be mediated based on who is accessing the information and what classes of information they are authorized to deal with.
- Requirement four -- ACCOUNTABILITY -- Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party. A trusted system must be able to record the occurrences of security-relevant events in an audit log.
- Requirement five -- ASSURANCE -- The computer system must contain hardware/software that can be independently evaluated to provide sufficient assurance that the system enforces requirements one through four above.
- Requirement six -- CONTINUOUS PROTECTION -- The trusted mechanisms that enforce these basic requirements must be continuously protected against tampering and/or unauthorized change. No computer system can be considered truly secure if the basic hardware and software mechanisms that enforce the security policy are themselves subject to unauthorized modification or subversion. [Ref. 3: pp. 3-4]

Derived from these six basic requirements are the criteria for evaluation of trusted computing systems. The criteria are divided into four divisions, D: minimal protection, C: discretionary protection, B: mandatory protection, and A: verified protection, ordered in a hierarchical manner from lowest to highest. Each division represents a major improvement in the overall confidence one can place in the system for the protection of sensitive information. A rating for a system is based on thorough testing of the security relevant portions of the system. The security relevant portion of the system is spoken of as the Trusted Computing Base (TCB).

Division D: Minimal protection has only one class and is reserved for systems that have been evaluated, but failed to achieve the standards of a higher class.

Division C: Discretionary protection contains two classes that provide discretionary access to information and the means to audit and account for such usage. The two classes are: class C1: discretionary security protection and class C2: controlled access protection. The Trusted Computing Base (TCB) of class C1 satisfies discretionary access requirements by separating users and data. The class C1 environment is expected to be one of cooperating users processing data at the same level of sensitivity [Ref. 3: p. 12]. Identification and authentication are required to determine authorized individual or group users. The discretionary control of class C2

is made more rigid through login procedures, auditing of security relevant events, and resource isolation. By limiting usage to specified individuals, accountability for sensitive data is more easily maintained.

Division B: Mandatory Protection contains three classes that are characterized by a Trusted Computer Base (TCB) that preserves the integrity of the security labels and uses them to enforce a set of mandatory access control rules by using the reference monitor concept. These three classes are: class B1: labeled security protection, class B2: structured protection, and class B3: security domains. class B1 systems have the same requirements found in class C2 along with an informal statement of the security policy model, data labeling and mandatory access control over named subjects and objects [Ref. 3: p.20].

Class B2 requires the presence of a formal security policy unlike class B1. This formal policy must state both mandatory and discretionary access controls. The TCB enforces a more rigid authentication mechanism. This is the first level that addresses covert channels (channels which allow transfer of information in such a manner that it violates the system's security policy). Systems which conform to class B2 requirements are considered to be relatively resistant to penetration [Ref. 3: pp. 28-30].

Class B3 systems must include a reference monitor that mediates over all user access to system information. They must be tamperproof and small enough for exhaustive testing and analysis. The audit mechanisms in class B3 systems are expanded to signal all security relevant events and recovery procedures [Ref. 3: pp. 33-40].

Division A: Verified Design contains one class, class A1 which has the most rigid security requirements given the state of current technology. Extensive documentation is required on the TCB to demonstrate the ability to conform to security requirements. Systems in this class are functionally equivalent to class B3. The primary difference is the formality of class A1. There are no architectural or policy differences. Formal security verification methods are required to assure that both discretionary and mandatory access controls protect all classified or sensitive information either stored or processed on the system.

2. The Security Kernel and Guard Technology

A security kernel is defined to be the hardware/software component that implements the concept of a reference monitor [Ref. 7]. Since the early 1970's several

efforts have been underway to build secure operating systems based upon the kernel approach. The concept of a security kernel grew out of the concept of a reference monitor--an abstract mechanism that controls the flow of information within a computer system by mediating every attempt by a subject (active system element) to access an object (information container) [Ref. 7]. The hardware/software mechanism that implements the reference monitor is called a security kernel.

The basis of the security kernel idea is that a small central portion of an operating system (both hardware and software) can be designed in such a way as to control the rest of the system and in doing so, make sure that the system functions according to some system of good behavior. To provide security, a kernel must:

- mediate every access by a subject to an object,
- be protected from unauthorized modification,
- and correctly perform its functions. [Ref. 7]

A kernel satisfies the first requirement by creating an environment within which all non-kernel software is constrained to operate and by maintaining control over it. The requirement to protect against unauthorized modification is satisfied by isolating the security software in one or more protected domains. Finally, the requirement that the kernel correctly performs its functions can be satisfied by using a formal methodology. Such a methodology would include:

- proof that the kernel behavior enforces the desired policy, and
- proof that the kernel is correctly implemented with respect to the description of its behavior used in the first step. [Ref. 8]

There are two classes of applications designed for security kernels: uni-level and multi-level. Uni-level applications can be categorized as those that could be built on conventional, nonsecure operating systems, but use a secure system to eliminate the cost associated with operating several systems at different levels. Because uni-level applications contain no sharing of information across security levels, they contain no additional "trusted" code other than the security kernel and supporting software. [Ref. 9]

The more interesting class of applications are those that are inherently multi-level. Multi-level applications generally enforce a much richer policy than that provided by the underlying security kernel. In order to build multi-level applications, the concept of a "trusted process" was added to the security kernel. A trusted process is a security relevant process that requires the ability to ignore one or more of the

kernel supplied protection rules. In providing a trusted process with this capability, one must demonstrate that this process does not circumvent the integrity of the kernel and that the trusted process enforces the more complex policy for which it was intended. The architecture of a multi-level application consists of one or more untrusted application processes working together with the security kernel, as extended by the application's specialized process [Ref. 8].

One specific application of the multi-level secure kernel is the "Guard Application". In military operations, there is a great need to be able to interconnect computers of different classification levels. Unfortunately, such connections are very difficult to implement in a secure manner. This problem would be made much easier if all computers had secure operating systems available, but such is not the case.

There is also a recurring need to make a subset of classified data available for use at a lower level of classification. To prevent compromise, this is done through the use of "sanitization" and "downgrading." Sanitization is often accomplished by removing the identity of the source of the data or by reducing the precision of the data. If the system that performs these functions is not multi-level secure, then we cannot trust it to perform the sanitization and release function properly.

A guard application is an intermediate solution to this problem. A guard consists of a secure computer that acts as an interface between systems at different levels [Ref. 10]. Information may be transferred to systems at greater security levels without intervention. Information transferred to systems at a lower security level must be checked to ensure that no compromise occurs. This check is normally made via human review.

Security kernels were first justified as a basis for allowing users having multiple clearances to share a common hardware. The emerging problem appears to be the inability to communicate and process information effectively at multiple levels without imposing unnecessary constraints on the users. Security kernels provide a foundation for allowing multi-level secure computing.

One means of automatically checking the security level of information to be transferred to a lower level is the use of a cryptographic checksum, a function of the entire record, computed by a special authentication device as data is entered into a system. The checksum is appended to the record and stays with the record forever. As information enters the system, the guard computes a unique, non-forgable checksum which is appended to the entry before it enters the data base. When a record is

selected for output, it is routed to a dedicated system (the guard processor) where the cryptographic checksum is recomputed. If the recomputed value is identical to the checksum appended to the record when it entered the data base, the entry can be released without further review. If the checksum check fails, the item will not be forwarded to the requester and the record, destination and all other pertinent information will be written to an audit file and reviewed by a sanitization special security officer.

The cryptographic checksum of the original entry is produced using a secret key known only to the guard interface processor. The checksum is produced by block-chained encipherment of the incoming entry. In block-chained encipherment, the ciphertext of each block of the item being enciphered is dependent on the contents of all previous blocks. The last block of an item is dependent on the entire entry and is used as the checksum. [Ref. 11: p. 15]

With the cryptographic checksum keys physically isolated from all components but the guard interface, the only method of forging a checksum is to pick a 64-bit number at random and attach it to a record. The probability of picking a correct checksum is $1/2^{64}$ (the size of the checksum) or 5.24×10^{-24} [Ref. 11: p. 15].

In summary, the protection against forgery is provided by protecting the key. Key protection is provided by:

- Physically separating the guard interface and the message processor.
- Hard-wiring the key on the encryption board so that it is not even readable by the guard.
- Providing a security kernel in the guard (input check-sum generator) to control their operation. [Ref. 11: p. 17]

3. Data Encryption

Data encryption is fundamental to a secure communications network. The methods available vary widely as do the security levels for which they are approved. Approval is based on the computational power, and the amount of time required to break the code. A cipher that cannot be proven to resist all attacks is considered "computationally secure" if the computational cost involved in breaking it exceeds the value of the information gained [Ref. 12]. Recent technological advances have produced computer chips which reliably encrypt data with a high degree of security. The relatively low cost and high speed of these devices make them excellent choices for secure network applications. The major problem to date has been getting them approved for transport of DOD classified data. Two major encryption methods are the

Data Encryption Standard (DES) [Ref. 13], and the public key systems [Ref. 12]. The Gemini system used in this research uses DES as its encryption method, and therefore it will be the only method discussed.

The Data Encryption Standard (DES) is the National Bureau of Standards (NBS) cryptographic protection standard [Ref. 14]. It is widely used for the protection of commercial data. It has come under attack from several sources [Ref. 14: p. 171]. Because of these alleged weaknesses, DES is not currently authorized for transmission of DOD classified data. Despite its problems DES remains a highly secure and reliable method of encryption for official documentation which would otherwise be transmitted in unencrypted form. The remainder of this section will discuss characteristics of DES encryption and techniques which can be used to maximize the protection of transmitted data.

There are four modes in which the DES may operate. They are: the Electronic Code Book (ECB) mode, the Cipher Block Chaining (CBC) mode, the Cipher Feedback (CFB) mode, and the Output Feedback (OFB) mode [Ref. 15].

a. Electronic Code Book (ECB) Mode

Figure 2.1 shows how a DES device operates in this mode. ECB is the simplest of the DES modes, however, it is also the most vulnerable to attack. This is because identical blocks of cleartext code will always produce identical ciphertexts until the encryption key is changed. This method is not recommended for transmitting messages which contain repetition of data forms such as English text messages [Ref. 14: p. 178]. Since identical blocks yield identical ciphertexts, by observing over a period of time an intruder would eventually be able to determine the cleartext message.

b. Cipher Block Chaining (CBC) Mode

Figure 2.2 shows how the CBC mode operates. CBC is a block encryption method which overcomes the pattern recognition problems of ECB mode by using the ciphertext of each preceding block as an input to encrypt the next block. The process is started by applying an initialization vector to the first block of data to be encrypted. Incompleted blocks are padded as additional protection against pattern recognition attacks.

c. Cipher Feedback (CFB) Mode

Figure 2.3 shows the CFB mode of operation. CFB mode is a stream encryption technique in which a key stream is generated, then combined with plain text to produce a ciphertext. The ciphertext is then fed back as an input to the key stream

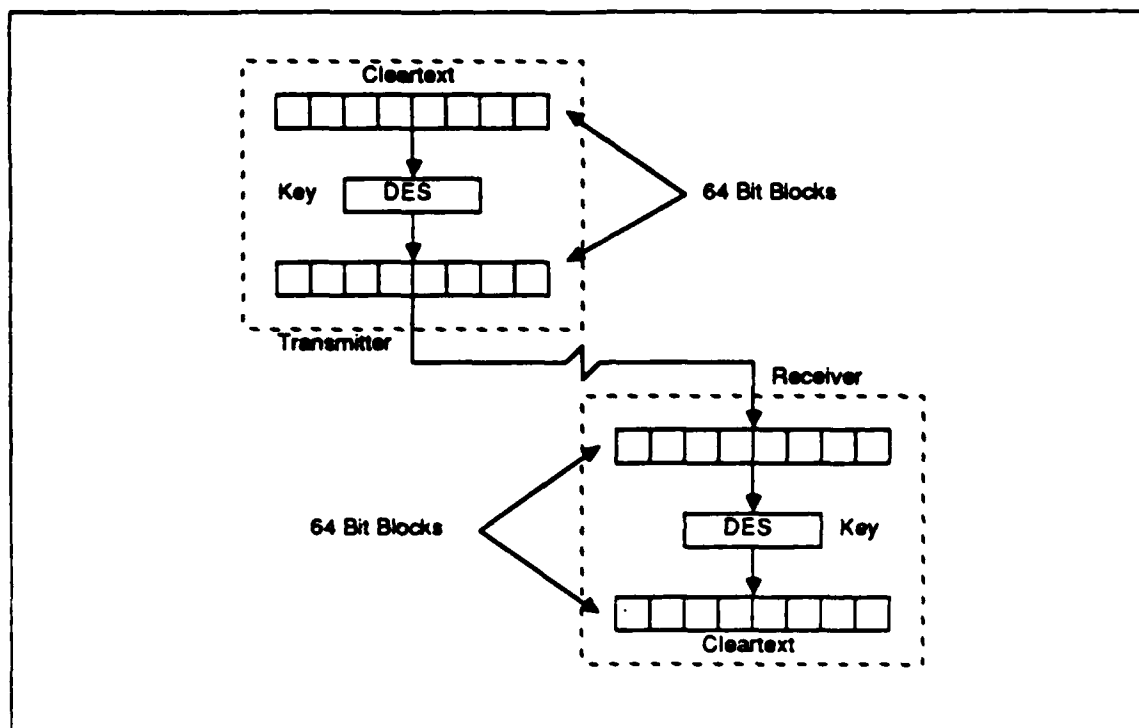


Figure 2.1 ECB Mode of DES Encryption.

generation process. Stream ciphers are in general slower than block ciphers [Ref. 16: p. 151], and are not used when large throughputs are required.

d. Output Feedback (OFB) Mode

The OFB mode is also a stream encryption method. In this method the key stream is completely independent of the plaintext and ciphertext streams. This eliminates the problem of error propagation and would seem to be a definite advantage. However, some degree of error propagation is required to detect message modification attacks [Ref. 16: p. 149]. As a result, OFB mode is not normally used in secure network environments. This mode is not implemented on the Gemini system's hardware encryption device because it is not self synchronizing.

The interface being developed in this thesis can best be implemented using the CBC mode of DES encryption. The system must be capable of quickly handling large volumes of data (large throughput), as well as official correspondence.

4. Summary

Assuming that the guard system works as described and it is interposed between TCO and other Tactical Data Systems, this approach has several interesting

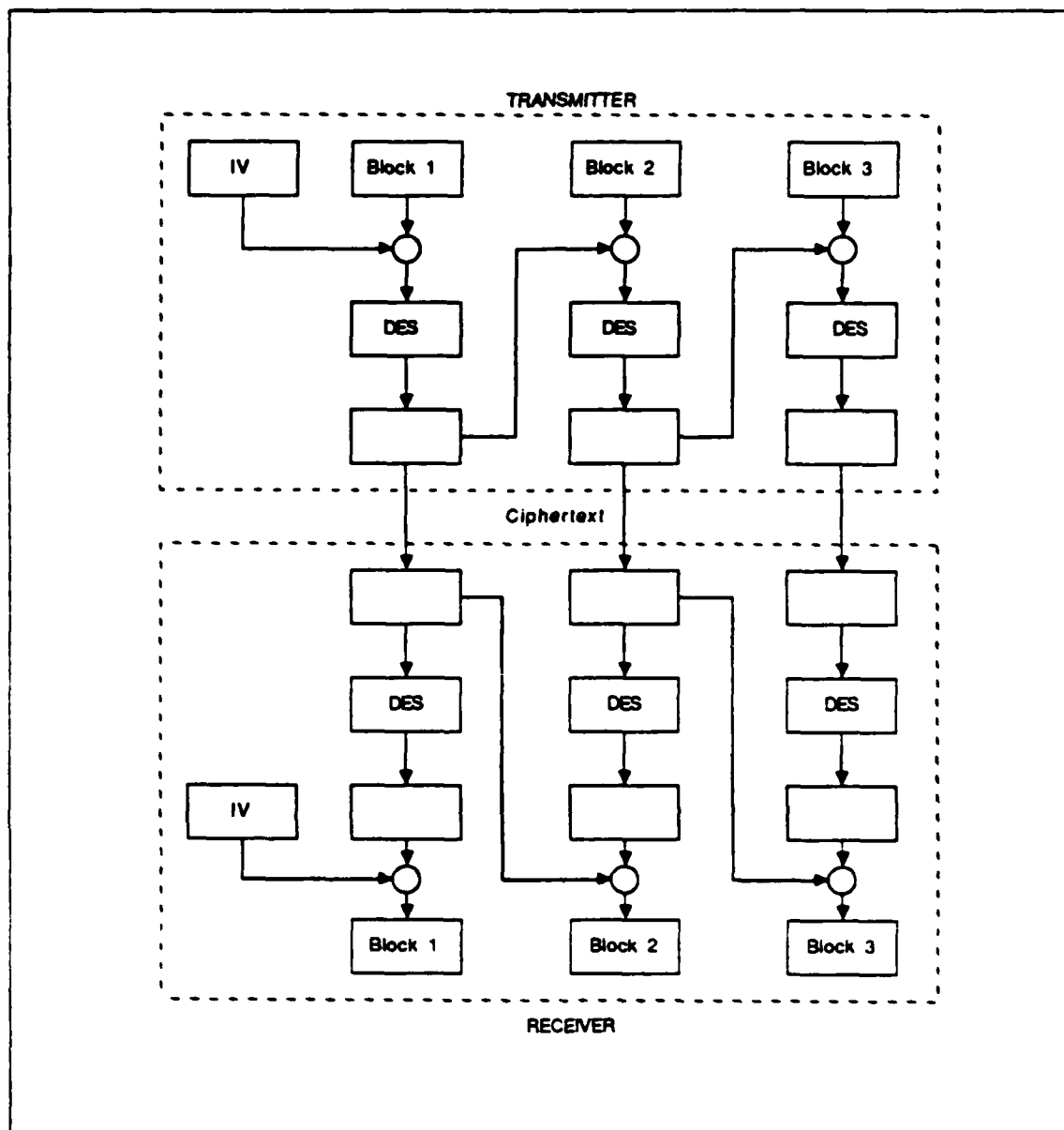


Figure 2.2 CBC Mode of DES Encryption.

properties with respect to multilevel security. First, no failure or compromise of hardware or programs in the TCO will permit data to spill from TCO to any other system. Second, no manipulation of TCO or its processor will release information across the guard interface. This is because the guard interface will be designed to only deal with TCO messages, and to escape the guard, a cryptographic checksum is recomputed before release of any message. If this checksum does not identically match

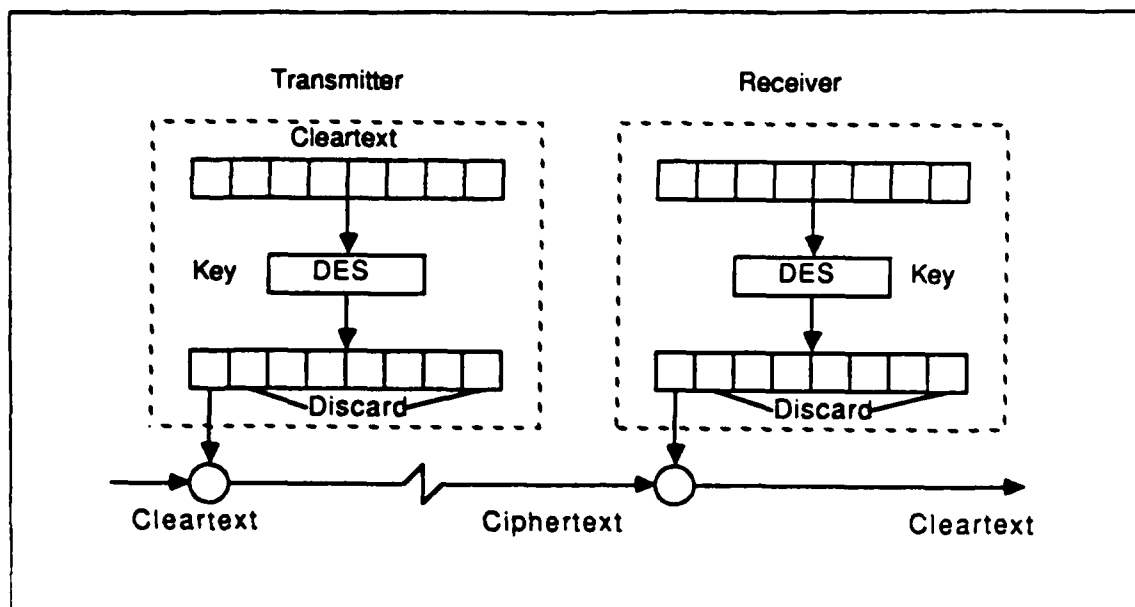


Figure 2.3 CFB Mode of DES Encryption.

the checksum computed when the message entered TCO, the message does not get released.

The guard can be designed in such a way as to permit TCO to test the correct operation of the guard by addressing various kinds of messages to itself. The only records that should return are those whose checksum is properly computed and recomputed. If there is a concern that an attacker could use this facility to generate and test checksums, it might be noted that it would take about 58,494 years to systematically try all possible 2^{64} checksums against a single record. At the rate of 10,000,000 trials/second on average, one could expect to find the correct checksum in one half the time, or 29,247 years [Ref. 11: p. 19].

B. GEMINI TRUSTED MULTIPLE MICROCOMPUTER BASE

1. Description of Gemini System Components

The Gemini Trusted Microcomputer Base is one of the systems currently being evaluated by the National Computer Security Center for certification at the B3 [Ref. 3] level of classification. Until recently, lack of evaluation criteria, as well as, microprocessor technology, made construction of such systems impractical. The foundation upon which all trusted computer systems are developed is the security kernel. The Gemini system employs the latest technology in both hardware and software engineering. Some of its major features are:

- The capability to operate up to eight Intel iAPX-286 microcomputers in parallel. This provides tremendous processing power, while communicating through shared memory increases throughput.
- The Gemini system is extremely flexible with regard to the types of peripheral devices which may be connected to the Multibus. These include fixed hard disk, removable disk, and high density floppy diskette drives, as well as, non-volatile memory devices. A maximum of eight devices may be attached to each RS-232 I/O interface board.
- With its multiple microcomputers, the Gemini system supports a variety of multiprocessing and multiprogramming applications. Processes can be pipelined to a single processor, or distributed in parallel among several processors.
- Other features include a NBS DES chip encryption device, real-time clock, and non-volatile memory to protect passwords and encryption keys. [Ref. 17]

The Gemini system also provides a self-hosting environment for software development [Ref. 17: p. 4]. This allows users to develop applications software. Gemini supports several advanced programming languages for development. These include JANUS/ADA, PASCAL, and C. Because of the secure environment, not all standard constructs are supported, especially in the input/output area. Because communications to and from devices require special formats, utility libraries are provided with the system containing routines which put calling arguments in the proper format for use in GEMSOS.

A major attraction for the Gemini system is its tremendous potential for growth. Its ability to handle a variety of hardware configurations is especially valuable in DOD applications where a trusted computer system may be required to communicate with systems using different protocols and hardware interfaces. When used as proposed in this thesis as a guard between two untrusted systems, the Gemini system could potentially communicate with a variety of communications devices using different I/O ports.

2. Gemini Resource Management Overview

The Gemini Secure Operating System (GEMSOS) kernel is logically divided into three management areas. These are: segment management, process management, and device management. Management functions are invoked by initiating a GEMSOS service call [Ref. 17: p. 5]. The formats for calling arguments are found in the GEMSOS interface libraries provided with Gemini's software development tools.

a. Segment Management

All data used in the GEMSOS environment is contained in segments. The applications programmer is mainly concerned with code segments, stack segments, and data segments. Bootstrap and kernel segments normally do not change when

developing basic applications software. There are eight segment management functions. A discussion of how to initiate these service calls is contained in [Ref. 20: pp. 16-28]. Segments can also be managed in groups. Secondary storage devices are represented by volumes which can be identified as separate entities to a calling process. Volumes and individual segments can be brought into the address space of the calling process by using resource management service calls.

b. Process Management

Through process management functions the Gemini system is able to support a full range of multiprogramming and multiprocessing applications. Each process is uniquely identified by code, stack, and data segments. Once created the process can be synchronized to run simultaneously with other processes using one of two methods. Eventcounts and sequencers were selected over other techniques because they are particularly well suited to operation in a secure environment [Ref. 17: p. 6]. All segments created in an applications program are assigned an eventcount and sequencer automatically. Process management calls to these devices allow the programmer to coordinate process functioning while maintaining access security.

c. Device Management

The third management area is device management. The Gemini approach to device management is to minimize the size of the security kernel code by reassigning device management functions to application level code whenever possible [Ref. 17: p. 9]. This has two effects. Reducing the size of the kernel makes verification easier, however it also makes writing applications software more difficult. Traditional input and output files are replaced by segments which can be read from or written into. Devices are attached and detached to allow them to be used by more than one calling process. Process synchronization primitives are used to control access to the segments made available to an attached device. The I/O device controller is treated as a process, which is then synchronized with the available segment eventcounts or sequencers to perform the required device management functions. Additional information concerning Gemini resource management functions is contained in [Ref. 17: pp. 5-10].

3. Gemini Secure Operating System (GEMSOS) Architecture

The Gemini system uses four hierarchical rings to implement its security structure. Ring 0 provides the most security, while ring 3 is least secure. It can support both discretionary and mandatory policies. The mandatory policy is contained in ring 0. This policy cannot be modified. Ring 1 is used to control the discretionary

or 'need-to-know' policy, supervise the use of the data encryption device, and support other security functions not contained in the mandatory policy. Ring 2 and 3 are available to the programmer for use in developing applications software. The security mechanism which coordinates inter-ring communications involves the control of access to subjects and objects. A subject is a process which is allowed to operate over a specific domain within the system. An object is a specific piece of information which is assigned a security label. All access between subjects and objects is controlled by the GEMSOS security kernel located in ring 0. Approval is based on comparison of the security labels of the two entities trying to gain access.

Compromise Properties:

- 1) If a subject has "observe" access to an object, the compromise access component of the subject must dominate the compromise access component of the object.
- 2) If a subject has "modify" access to an object, the compromise access component of the object must dominate the compromise access component of the subject.

Integrity Properties:

- 1) If a subject has "modify" access to an object, then the integrity access component of the subject dominates the integrity access component of the object.
- 2) If the subject has "observe" access to an object, then the integrity access component of the object dominates the integrity access component of the subject.

Figure 2.4 Compromise and Integrity Properties.

Security labels are used to identify the access class of all subjects and objects. The access class is further broken into a compromise (observe) level and an integrity (modify) level. Compromise and integrity protection are based on strict properties which must be observed in order for access to be granted. Figure 2.4 is taken from [Ref. 17: pp. 16,17], and contains a simplified statement of these properties. Domination as stated in these properties means that the level of the access component is greater than or equal to the entity it is trying to observe or modify.

Compromise protection property 1 is a traditional security policy. It states that in order to observe information, you must have a clearance equal to or greater than the information you want to observe. The second property is more subtle. This property prevents, for example, a secret user from modifying a file which could be viewed by a confidential user. This property is especially important in prevention of 'Trojan horse' type attacks [Ref. 18: pp. 65-69]. The integrity protection properties are similar to the compromise properties except that they refer to the ability to modify information. Property 1 states that in order to modify a confidential document, you must have at least a confidential integrity level. The second integrity property prevents, for example, a secret user from observing (and possibly being influenced by) information which could be modified by someone at a lower integrity level.

In addition to the access class integrity described above, the Gemini system also employs ring integrity. Ring integrity means that subjects at a certain level can only access objects of the same, or higher ring number. This policy reinforces the hierarchical structure of the GEMSOS rings.

These compromise and integrity policies are further complicated by the fact that the Gemini system is a multilevel system. This means that both users and resources may have clearance to access a range of security levels. Multilevel subjects are potentially very dangerous because within their range of operation they are not subject to the second compromise and integrity protection properties [Ref. 17: p. 20]. It is up to the applications programmer to ensure that he does not create subjects which will allow violation of these properties. This is especially important when interfacing with devices. Figure 2.5 is taken from [Ref. 17: pp. 21,22], and represents a summary of the security properties of single and multilevel devices.

Device access levels refer to the physical security of the environment in which the device is going to operate. This is separate from the security level of the process which is attempting to communicate with the device. For example, a terminal located in an unsecure room with an unclassified device access level, cannot receive information from a secret process. The term single level device implies that the maximum and minimum access classes for the device are the same. In multilevel devices the access classes are different, and represent the range over which the device is allowed to operate.

Single Level Devices:

- 1) To receive ("read") information:
 $\text{process maximum compromise} \geq \text{device minimum compromise}$
 $\text{device maximum integrity} \geq \text{process minimum integrity}$
- 2) To send ("write") information:
 $\text{device maximum compromise} \geq \text{process minimum compromise}$
 $\text{process minimum integrity} \geq \text{device minimum integrity}$

Multilevel Devices:

- 1) To receive ("read") information:
 $\text{process maximum compromise} \geq \text{device maximum compromise}$
 $\text{device minimum integrity} \geq \text{process minimum integrity}$
- 2) To send ("write") information:
 $\text{device minimum compromise} \geq \text{process minimum compromise}$
 $\text{process maximum integrity} \geq \text{device maximum integrity}$

Figure 2.5 Single and Multilevel Device Properties.

4. Application Development Environment

a. General Description

This section provides the foundation for the necessary steps to develop software for the Gemini system. It is important because it provides the basic components that are needed for applications development. This provides a bridge between the applications programmer and the operating system primitives to allow development of reliable software.

The objective of this section is to present an ordered method of developing application software within the GEMSOS environment. It should be considered as a guide and not as a fixed set of rules. The facts considered here are taken from [Ref. 20].

b. Hierarchical Storage Structure

The Gemini system provides a one-level secondary storage system for information. File concepts are not supported by Gemini. Instead, segments are used which are considered as objects having logical attributes related to them (i.e., security). Each segment has a maximum size of 64 K bytes. The segments are handled by the

system as a hierarchy, where each segment is identified by a unique path name corresponding to the index of an entry in the Location Description Table (LDT) of the process that creates and/or uses that segment.

The representation of segments follow a hierarchical structure in which the root is the system root (transparent to the user) and the whole collection of segments is assembled as an inverted tree. Each application program becomes part of the hierarchical structure and is statically created at system generation time using a system submit file as explained in [Ref. 19].

System generation consists of creating a hierarchical structure of all segments known to the system at system runtime. It is basically the inclusion of the segment hierarchy of the application program into the GEMSOS hierarchy. This is accomplished by using segments declared in the submit file. Figure 2.6 shows a typical hierarchical structure representing the segments that GEMSOS requires to run applications. This structure is fixed and must be considered in the development of any application program. Figure 2.7 shows the addition of segments necessary to execute a specific application. Entry 5 in the hierarchical structure is always dedicated as the mentor (or root) of user applications. This entry is the mentor of all the segments that are needed to implement the upper levels of the system application program. Under the Gemini concept, a segment can support up to 12 descendents numbered from 0 to 11. To support this concept there is a volume aliasing table (VAT) or volume mentor that relates the segments to their mentors [Ref. 19: p. 2].

When a segment is used by a process, it must be deleted before it can be made available for another process. The numbers indicated in both Figures 2.6 and 2.7 correspond to entry numbers of segments associated with a mentor (from 0 to 11); segment numbers have a different enumeration which corresponds to their entries in the LDT. Each segment in the system has a unique path name that identifies it, but not by the application process. Instead, a process-local segment number is used. When two processes share the same segment, each one recognizes the segment by a different segment number assigned in its own LDT.

For example, in Figure 2.7 the path 5,7 corresponds to a segment that holds the code for a child application program and must be declared in the submit file. Entry 5 is the mentor for entry 7. This creation is static and the path indicated must be known in the application program. On the other hand, the path 5,5,7 is used to hold data and will be created dynamically during execution of the application program. A more complete explanation is presented in [Ref. 19].

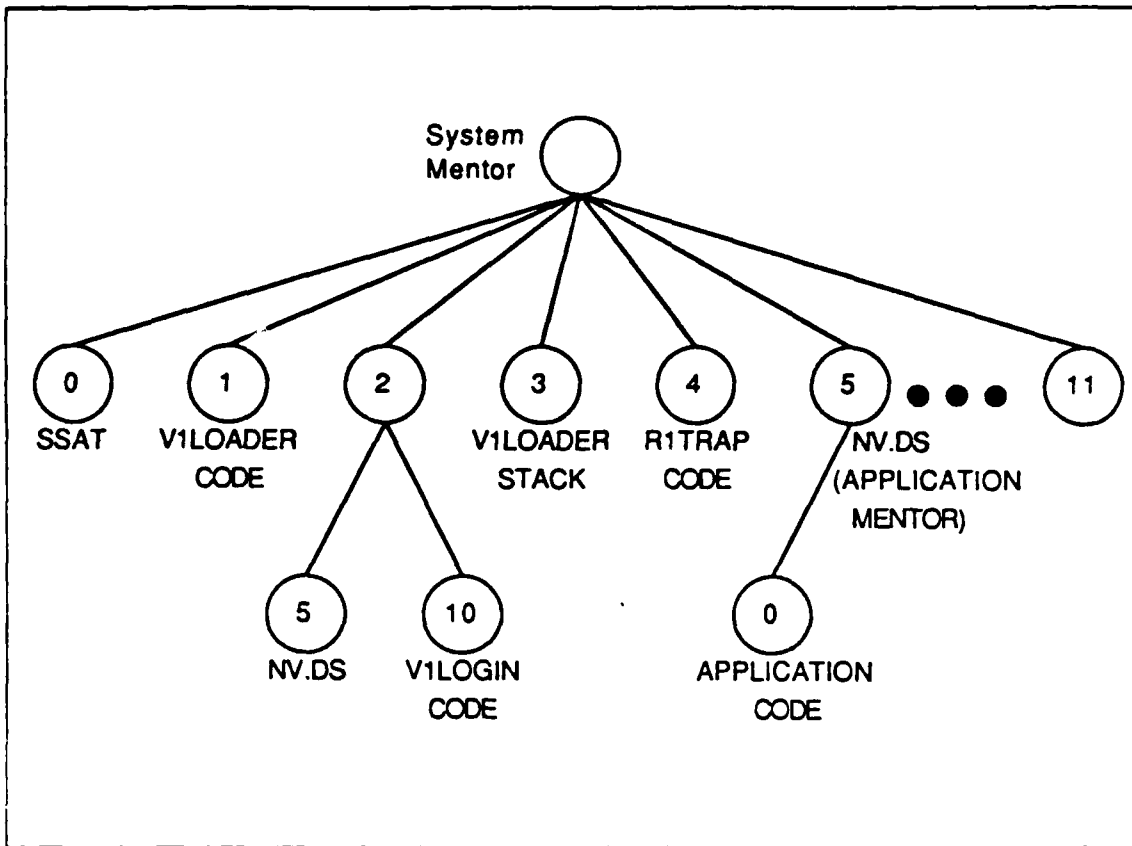


Figure 2.6 GEMSOS Hierarchical Structure.

c. I/O Device Assignment

The process of attaching devices must be accomplished before an I/O device becomes known to the system. This involves assigning a logical process to the I/O device so that the device is known by the process. The process then contains the device drivers to allow it to manipulate the device. A device may be declared as either a read (input) or write (output) device, as part of the information provided to the "ATTACH" primitive call in GEMSOS [Ref. 20: pp. 43-44]. A device may be attached to only one process at a time. An error will occur if an attempt is made to attach a device by more than one process. The inverse primitive is called "DETACH" device. This primitive detaches the associated process and the device becomes available for further assignment [Ref. 20: p. 45].

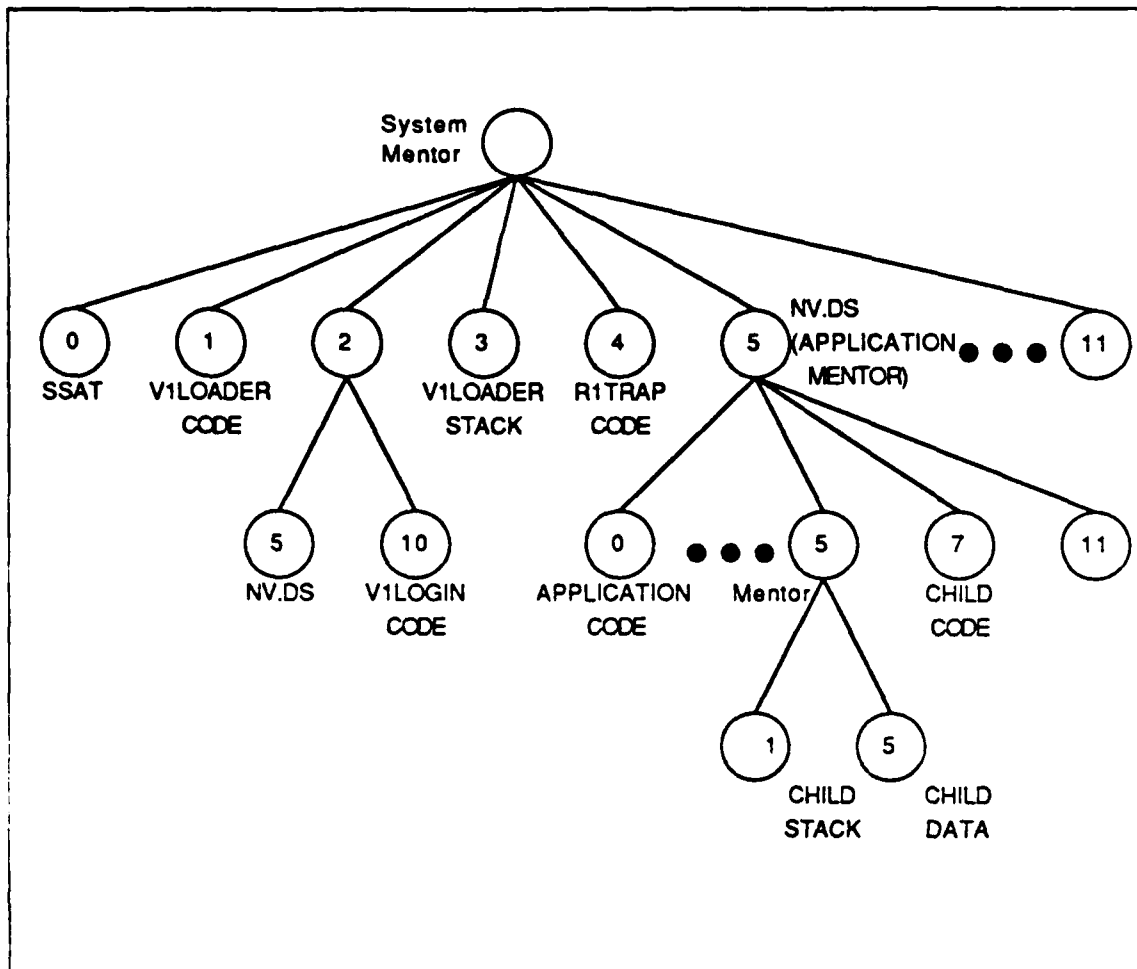


Figure 2.7 GEMSOS Hierarchical Structure Including an Application.

d. Process Creation

This section describes the steps that should be considered when creating a process. One process is created from another. The creator is known as the "parent" and the created process is known as the "child". The child process receives its resources from the parent. Each process is a collection of segments known to that process and managed using primitive functions or "calls" provided by the system. An address space is created to hold a segment and an application programmer must make use of GEMSOS primitives for the creation and use of address spaces.

The sequence of steps that must be followed in order to create a process starts with the "CREATE" primitive. When this primitive is called an address space is created for that segment. The next step links the address space with a specific process,

i.e., a recognition of the segment by the process as an entry in its Local Description Table (LDT). The result is the identification of this address space with a segment number. The primitive used for this is the "MAKEKNOWN" primitive.

The last step is related to the use of this segment--a segment must be in main memory in order to be used. This function is performed using the primitive "SWAP-IN", which loads the segment from secondary storage into main memory.

When the segment is no longer needed by the process, an inverse action must be performed in order to release the space used by the segment. As in the procedure described above, a logical sequence of steps must be followed, starting with the release of the memory used by the segment. This is performed by the "SWAP-OUT" primitive in which the segment is written back out to secondary storage. The next step is the elimination of the association of the segment and the process. Elimination of a segment from a process' address space is performed by the "TERMINATE" primitive, which breaks the association and makes the entry number used in the LDT of the process.

The final step is total removal of the segment from the system address space. This is accomplished by the "DELETE" primitive. The result is the removal of the segment from the system and the return of the address space to system resources.

e. Process Management

Because a process is a collection of segments, segment creation is an important step that should be considered during process creation. Each segment has its own attributes which are specified in a segment structure record and then declared in a "CREATE_SEGMENT" primitive. The segment is created with the specified attributes and the addition of a new branch in the hierarchical structure is made [Ref. 20: pp. 20-21]. The inverse action is the "DELETE_SEGMENT" call, where a segment previously created is now removed from the hierarchical structure [Ref. 20: p. 22]. This call is used when a process finishes its execution because segments are not automatically removed.

f. Makeknown/Terminate Segment

The "MAKEKNOWN_SEGMENT" call adds the specified segment to the address space of the calling process [Ref. 20: pp. 23-25]. Like all of the primitives it has its own record structure, which must be initialized with the pathname that the segments will use in the hierarchical structure. The inverse call, "TERMINATE_SEGMENT" eliminates the pathname from the hierarchical structure and frees the segment number from the LDT table [Ref. 20: p. 26].

g. Swapin/Swapout Segment

A segment is created in secondary storage by first using the primitive "SWAPIN_SEGMENT," to provide main memory space for the new segment. "SWAPOUT_SEGMENT" is used to write a segment from main memory to secondary storage. [Ref. 20: pp. 27-28]

h. Synchronization

Synchronization among processes is maintained by the use of eventcounts and sequencers [Ref. 21: pp 115-124]. Segments used to control synchronization must be common to all of the processes involved in that synchronization. An eventcount is maintained by an integer counter under control of the cooperating process. Completion of an operation (event) is signaled by incrementing the eventcount. The updated eventcount provides a signal to a waiting process that the operation is complete. The primitives used are described in [Ref. 20: pp. 37-41].

5. Summary

The Gemini Trusted Multiple Microcomputer Base is an extremely capable computer system which combines state-of-the-art technology with a high degree of flexibility to handle a variety of possible applications. Its multiple processor and multiprogramming configurations are valuable assets when functioning as a secure guard interface between two untrusted systems as proposed by this thesis. Its ability to simultaneously handle devices with different protocol requirements and security levels could potentially eliminate the need for time consuming review and sanitation of information for transfer between Tactical Data Systems.

III. DEVELOPMENT OF A MULTILEVEL SECURE INTERFACE

A. GENERAL

1. Objectives

The objective of this research is to develop a simple interface which will demonstrate how the Gemini Trusted Multiple Microcomputer Base may be effectively used as a trusted computing base (TCB) linking untrusted command and control systems. There are four major phases in the development of the system:

- 1) Demonstrate the transportability of the JINTACCS Automated Message Processing System (JAMPS) software to the Gemini Multiprocessing Secure Operating System (GEMSOS).
- 2) Demonstrate the inter-segment linkage for the use of multiple language applications programs and libraries for use with GEMSOS.
- 3) Establish two way communications between users at remote terminals using the Gemini system as a secure communications interface.
- 4) Demonstrate the use of Gemini security mechanisms to prevent unauthorized access to sensitive information.

The Air Force JINTACCS Program Office in Langley, Virginia and MITRE Corporation are currently working jointly to develop an automated, JINTACCS format, message preparation system. This system is the Joint Interoperability of Tactical Command and Control Systems (JINTACCS) Automated Message Preparation System (JAMPS). JINTACCS was established by the Joint Chiefs of Staff (JCS) so that the military services could have a common standard for communicating during joint operations. JAMPS is a software package developed to automate the preparation of these JINTACCS-standardized messages. The first phase of this research is to demonstrate the transportability of the JAMPS software to the Gemini Multilevel Secure Operating System (GEMSOS) in order to have JAMPS operate in a multilevel secure environment allowing automation of JINTACCS message preparation for communication between systems.

The Gemini system provides a self-hosting environment for software development and provides development environments in several languages to include PASCAL, C, and ADA. The second phase of this thesis is to develop applications using more than one programming language to display the intersegment linkage of the multi-language gates provided with GEMSOS and the flexibility of the Gemini application development environment. The intention is to demonstrate the use of

ADA in a secure environment and to show the capability that GEMSOS has for linking multiple languages. The languages used are PASCAL MT+86 and Janus ADA. The Janus ADA environment has not been completely developed for use with GEMSOS and a secure environment, specifically not all ADA constructs are supported in the GEMSOS environment. The majority of the modifications occur in the input/output area. Because communications to and from devices require special formats in a secure environment, Gemini provides utility libraries which put these calls in the proper format. Currently, these utility libraries are well developed for the PASCAL language. The goal of the second phase of this research is to demonstrate the flexibility of GEMSOS by writing segments of code in PASCAL and ADA using inter-segment linkage to have the system manager, a trusted process written in PASCAL, call and control multiple TDS terminal utility processes, written in both PASCAL and ADA while maintaining the rules of compromise and integrity required in the GEMSOS secure environment.

In order to create a realistic communications link, it is necessary to simulate having untrusted computer systems communicating with each other. This is accomplished by having the Gemini system communicate with itself using separate I/O ports. By routing the incoming and outgoing traffic from each port to separate processes, the "multiple computer" environment may be simulated. The interface is controlled by the multilevel security manager located at a data terminal linked to the Gemini system. This security manager is responsible for:

- System start-up and initialization.
- Assigning access levels for other terminals.
- Entering the cryptographic key.
- Control of communications to and from the external ports.
- Routing of message traffic to the appropriate terminals.

Each user terminal is assumed to be located in an area which would provide appropriate physical security for the access level of the terminal. Each may enter messages, transmit messages, and display incoming messages provided the terminal at which they are located has the proper access level.

The final goal, to test security of information and access, is demonstrated using a series of specific configurations and data sets to exercise security mechanisms. These tests are meant to demonstrate, rather than prove that information security and integrity are preserved. They are in no way intended to be exhaustive, however, they allow for a series of observations to be made concerning overall system security.

2. Design Constraints

The Gemini system used for the development of this demonstration has 12 ports available for attachment of I/O devices. This allows for up to eight user terminals and the security manager terminal. For this demonstration, only the system manager terminal and two TDS utility terminals are used for simplicity of design and testing.

Software constraints are generated by the environment in which this type of system would be used. An assumption is made that when acting as a multilevel trusted computing base and interface device, the system will most likely be adapted to "off-the-shelf" systems within the Department of Defense (DOD). Systems such as this have built-in physical and software security attributes. For this reason, no effort is made to provide security between the Gemini system and the remote terminals other than end-to-end encryption. This security would be provided by some mechanism other than the Trusted Computing Base (TCB). The lines are assumed to be secure, as are the locations in which the terminals are used. In order to provide overall system security, these security measures would have to be verified prior to installation of the trusted computer system at a particular activity. Another assumption is that the system demonstrated may communicate by a variety of means including hard wiring, secure teletype, Autodin, DDN, or satellite communications. For this reason, no specific protocol was adapted. JINTACCS message processing was used as the message format, adapted to the Gemini system. Header information and secure labels were placed on each message in order to allow transmission over a variety of communications devices.

3. Summary of Design Decisions

Figure 3.1 shows a block diagram of the proposed system design. For simplicity of the demonstration, only three terminals are used. One terminal is used as the multilevel system manager and the other two simulate untrusted tactical data systems such as MAGIS/IAC or ATACC. To provide additional flexibility, the access class of the untrusted terminal may be changed by the system security manager. The untrusted terminal may send and receive messages from the trusted computer system, however they must rely on the system security manager to actually transmit the messages and provide the appropriate security.

It is necessary for the system security manager to create a single level process for each user terminal attached. Figure 3.1 shows how a single level process is used to

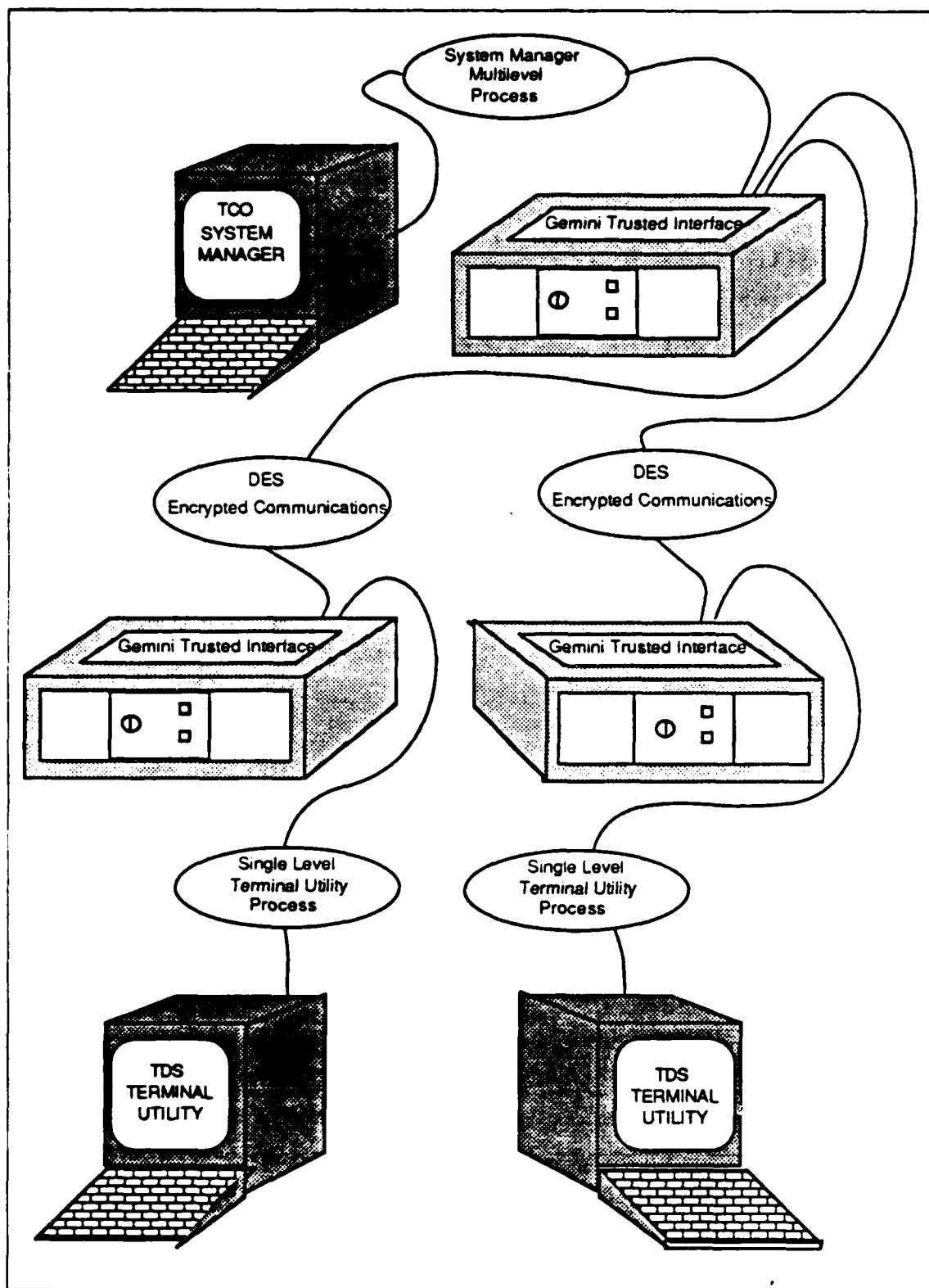


Figure 3.1 Proposed System Design.

provide a buffer to protect information. Even if an attacker were to cause information of a higher level to be misrouted by the multilevel process, it would still be protected from compromise by the single level process which interfaces directly with the user terminal.

In order for this design to work there must be synchronization among the processes. Interprocess communications are synchronized by using eventcounts [Ref. 22: p. 20]. Although the system simulates the link of two separate systems, only one multilevel communications process is used to simplify the synchronization problem. Since the communications processes would be identical this limitation does not adversely impact system design.

B. SYSTEM IMPLEMENTATION

1. Hardware Components

As discussed in the preceding section, the number of data terminals used in the system was limited for simplicity of design and testing. Figure 3.2 shows the final system hardware design. Terminal 100 is used by the system security manager to initiate and coordinate communications via the external communications interface. Remote user terminals are connected as read/write devices and attached to global serial ports 0 and 3 of the Gemini computer. This represents untrusted users exchanging information via the trusted process. The external communications interface is simulated with the use of a null modem allowing the Gemini computer to communicate with itself. Global serial ports 1 and 2 were used for this purpose. A user's guide for this demonstration is provided in Appendix C.

2. Application Program Format

Preparing programs to run in the Gemini Multilevel Secure Operating System (GEMSOS) environment is more complicated than running PASCAL programs in a nonsecure environment. In order to be accepted by the system they must first be put into a specific format which can be recognized by GEMSOS in order to gain access to the security kernel. There are several software tools which can speed up the process of preparing a program to be run in the secure environment. The fact that a program compiles successfully does not necessarily mean that it will run in the GEMSOS environment. For a PASCAL program, following compilation, the program is linked to the appropriate modules using a file named, 'application_name.KMD' [Ref. 23]. This file contains a formatted list of the modules with which the application segment

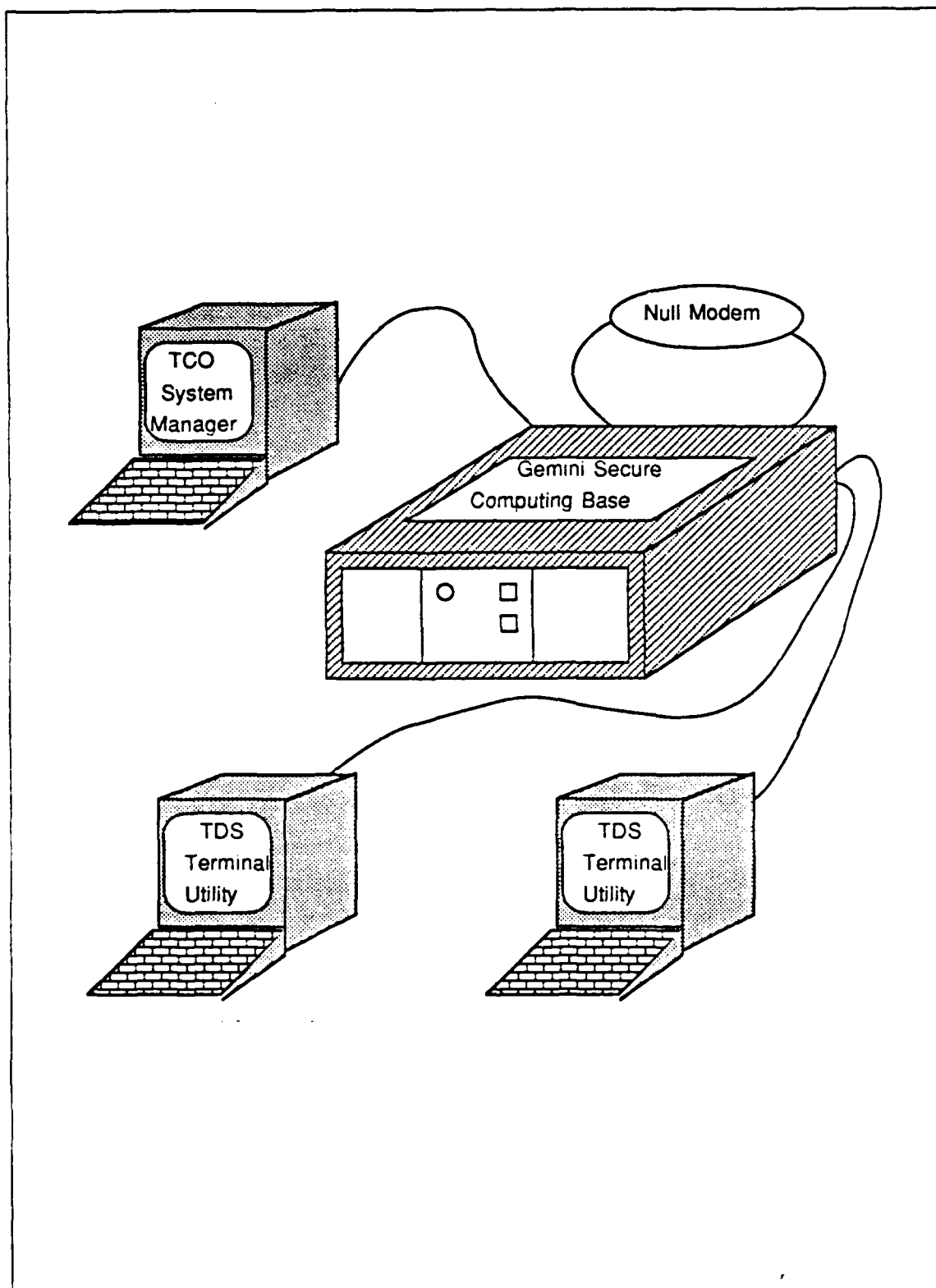


Figure 3.2 Final Hardware Diagram.

needs to be linked. The result of the linking process is a file named 'application_name.CMD' which still has no security classification assigned. To assign security classification, and prepare the program to execute in the secure environment, a secure volume must be created by running the operating system generation (SYSGEN) program.

When developing a Janus/ADA application which is to run on GEMSOS, 3 main points must be kept in mind:

- Standard Janus/ADA I/O file type utilities cannot be used, GEMSOS kernel calls must be made instead.
- The first statement in an application program must be a function call to the "get_r1_def" function. This function returns the ring 1 process definition record which is required for subsequent kernel calls.
- All applications must be linked with the "/c80" option, for example "JLINK TDSI/C80."

To assign security classification, and prepare the program to execute in the secure environment, a secure volume must be created in the same manner as mentioned above using the SYSGEN program.

Executing the SYSGEN program includes the application program in a segment structure which is then transformed into a "bootable system structure on formatted volumes." [Ref. 19: p. 1] Detailed procedures for using the SYSGEN program are contained in [Ref. 19: p. 7-20]. The key to proper use of the SYSGEN program is identifying the segment structure in which the application segment will be placed. The segment structure includes the boot-strap, kernel, application code, and data segments. The easiest way to identify this segment structure is to include it in a submit file named 'application_name.SSB.' For basic application programs, the segment structure does not change. Use of the submit (.SSB) file eliminates the need to enter the segment structure interactively each time the operating system generation program is run. Use of the SYSGEN submit mode is further explained in [Ref. 19: p. 9-14]. The submit file used for system generation of this demonstration (GUARD.SSB) is provided in Appendix M.

C. SYSTEM SOFTWARE DESIGN

1. Application Segment Development

Application software for this system was developed using modular programming construction techniques. This allowed for independent testing of each module prior to its inclusion in the main program. This technique was especially useful

because trouble-shooting GEMSOS related Ring 0 service calls can be difficult for new users. Processes were developed as application code segments. They are:

- 1) multilevel system manager process (PASCAL MT+ 86)
- 2) single level terminal utility process (PASCAL MT+ 86)
- 3) single level terminal utility process (Janus ADA)

Each process was developed as an independent application code segment. The single level terminal utility segment may be assigned different access levels by the system manager to demonstrate the interface to more than one system.

a. System Manager Segment

The system manager segment controls system configuration, data encryption, and communications through the external communications ports. Figure 3.3 shows a flow diagram of how this segment is constructed. A detailed source listing of the code implementation is contained in Appendix D.

Creation of a child process requires completion of four record structures. Each record structure has several entries. Each entry is completed in a specific order which builds to the 'CREATE_PROCESS' resource management call. Detailed instructions for process creation and record entry format are found in [Ref. 20]. Segment and process management are the most difficult concepts for someone unfamiliar with secure computer systems to grasp. The procedure developed in this segment could be used as a model for process creation in other programs. The specific entries may vary, however, the physical structure of the procedure is general enough to fit a variety of applications.

The system security manager located at terminal 0 has direct control over system assets. To provide this control, the system manager has the option of specifying (within predefined limits) how the system will operate. These parameters are entered when the system is initialized. They are interactively entered into a system operator record from which they can be drawn when required by other procedures. Parameters which do not need to be directly controlled by the system manager are fixed and cannot be directly accessed.

b. Terminal Utility Segments

As discussed earlier in this chapter the user terminals can input, transmit, and display messages. Each terminal is a single level device capable of sending and receiving messages of the same level. Figure 3.4 shows a flow diagram for the terminal utility application segment. The actual code for the programs which implement this

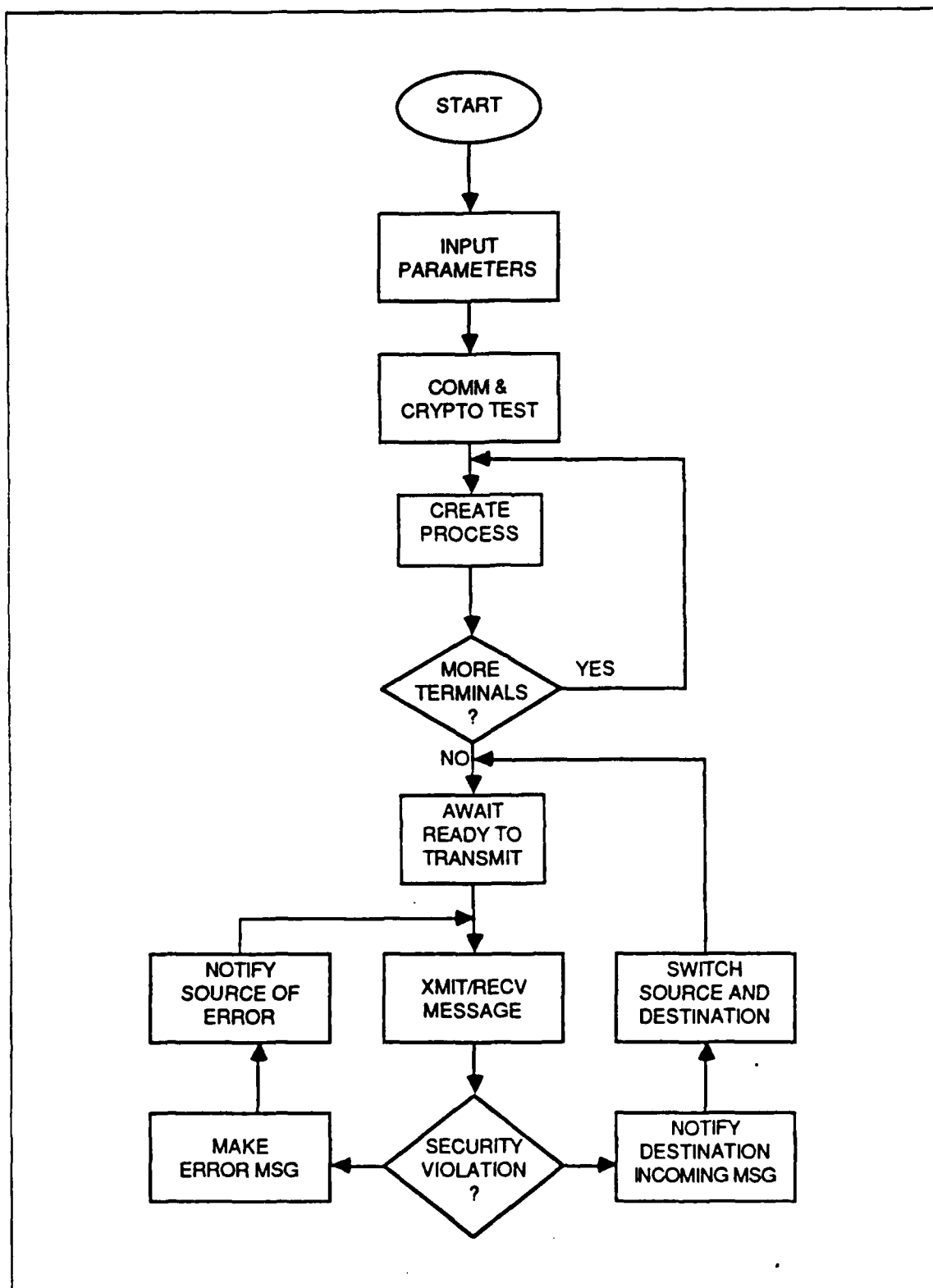


Figure 3.3 System Manager Flow Chart.

flow diagram is contained in Appendices H through L. This program is activated when the terminal process is created by the system manager process.

All messages sent and received at the terminal are stored in a specially designated message buffer segment. Access to this segment is shared by the user terminal and the system manager process. Each terminal has its own message buffer segment, and cannot access the other's segment without going through the system manager process. When the user has completed his message transactions, he initiates a logoff procedure. The logoff procedure deletes the terminal process and returns the resources allocated to the process to the system manager process which created it. These include memory space, process local segment numbers, and any attached devices.

c. Program Documentation

Each module in the application segments has a header describing its purpose and general operation. The intent was to provide clear programs which could be used as a basis for future research. In some cases this meant sacrificing efficiency in order to provide better clarity.

2. Process Synchronization

Process synchronization was accomplished using the eventcount of the message buffer segments of each terminal process created by the multilevel system manager. By advancing the proper stack eventcount the terminal process alerts the system manager that it is ready to begin message processing. The terminal advances the outgoing message buffer segment eventcount to notify the system manager process when it desires to transmit a message or when it has completed processing. When a terminal process indicates that it desires to transmit a message, the system manager transmits the message, and then unblocks the other terminal process to display the incoming message by advancing its incoming message buffer eventcount. This process can be continued indefinitely. The actual implementation of this sequencing is further explained in the applications code segment listings contained in Appendix D.

D. DESIGN SUMMARY

This chapter has discussed the system design process in terms of its objectives and limitations. The resulting hardware and software configuration was implemented using modular construction techniques which greatly reduced the number of software errors.

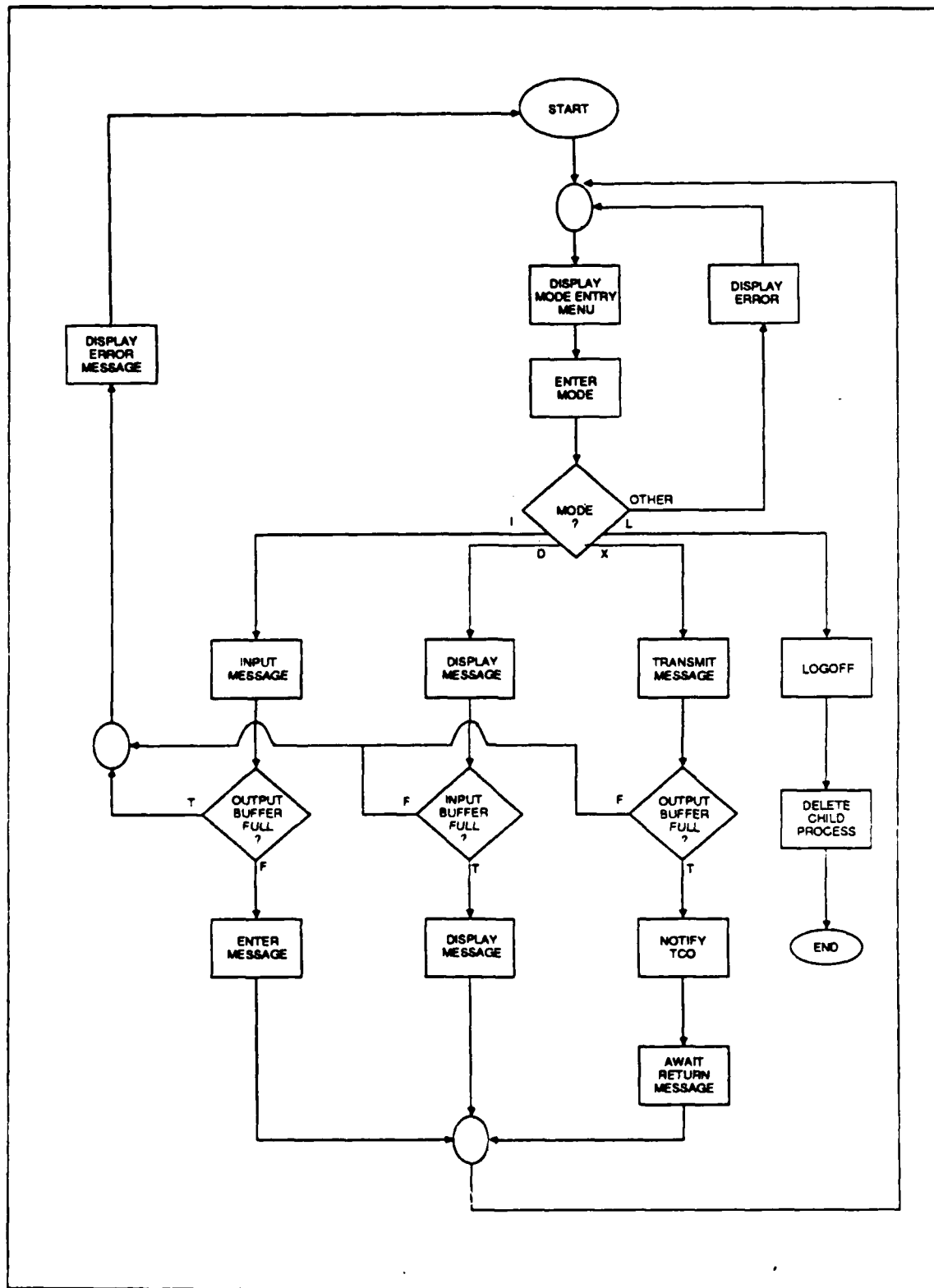


Figure 3.4 Terminal Utility Flow Chart.

The resulting system uses the Gemini as a secure communications interface and message processing facility. All communications are protected to the maximum extent possible using the Data Encryption Standard (DES) algorithm in the cipher block chaining (CBC) mode. Terminal processes are assigned single level access which prevents a user from gaining unauthorized access to classified information by allowing the kernel to enforce the rules of integrity (authorization to write) and compromise (authorization to read).

IV. DISCUSSION OF RESULTS

A. SYSTEM OPERATION

The model communication system developed in this thesis to demonstrate using the Gemini Trusted Multiple Microcomputer Base as a secure interface for data communications met or exceeded all design goals. Messages were passed between tactical data systems in a manner that ensured security from unauthorized access at both source and destination. Data encryption was used to maximize the security of the transmitted data. Finally, by varying the access class of the terminal processes it was possible to demonstrate the system's ability to detect and respond to security violations. Flexibility in determining system configuration allows modification of system parameters to meet a variety of test requirements.

System operation is initiated and controlled by the system manager. The multilevel system manager process creates the single level TDS processes at access levels based on two separate controls. First, the Gemini computer used for testing and evaluating the interface was configured to allow classification of the serial input/output ports. This means that the operating system, specifically the kernel, recognized the maximum access level of each serial port. This process was completed by burning the machine PROM specifying that each port had a maximum access level. The second control measure was accomplished by allowing the multilevel system manager to specify an access class at or below the maximum allowed by the serial port. Based on these constraints, a TDS terminal may only display messages created at or below its access level (enforcing the compromise level, authorization to read) and send messages to other terminals at or above its access level (enforcing the integrity level, authorization to write). It is important to note that, for this demonstration, the user does not assign the classification of the outgoing message. Message classification is assigned by the system manager according to the access level of the TDS sending the message. All security checks are performed within the multilevel system manager process to strictly enforce the security policy established for that system. To facilitate security testing, the TDS access levels in this demonstration are manually entered by the system manager. (If a local commander did not want to leave this choice up to the system manager, the access level information could be hidden in a file that he does not

have access to, or simply left as configured on the system PROM (built into the hardware). Once started, the system operates independently. This eliminates the possibility of a system manager manually misrouting information stored in the message buffers.

One potential problem was the possibility that an unclassified user could enter classified information in an unclassified message and transmit it to an accomplice who had tapped into the external communications line. To help prevent this, the outgoing message is encrypted using keys which are inserted by the system manager. Possible compromise of the key could further be prevented by having the key entered by someone other than the system manager or by a hardware key generation device. The goal was to develop a process which would demonstrate the enforcement of basic computer security measures and provide a baseline understanding of this technology and its application in a tactical command and control system. There are a wide variety of possible system configurations. Selection of a particular configuration would have to be based on the security policy of the particular command and its associated security requirements.

B. DEVELOPMENTAL PHASES

1. JINTACCS Automated Message Preparation System Implementation

a. Discussion

JAMPS is a portable set of computer controlled equipment for assisting operators in composing and exchanging JINTACCS messages. It provides one or more operators a work station for composing communications messages over an external link. Each work station provides the capability to store and retrieve messages composed at the work station and messages received via external communications. The original intent of this thesis was to completely integrate the JAMPS software into a secure environment within and controlled by the Gemini Trusted Computer. After close review it was determined that total integration of JAMPS was beyond the scope of this thesis and further study is recommended in the area of transporting JAMPS into the GEMSOS environment. JAMPS is used, however, in this thesis as a controlled process by each of the TDS terminal utilities as the mechanism for creation of messages used in the communication process.

b. Compromise Design

The primary objective of this thesis remains the same, except that the JAMPS software was not transported onto the Gemini computer. The JAMPS procedure is, however, controlled by the TDS terminal utility process. Figure 4.1 depicts the hardware configuration showing a personal computer running the JAMPS software attached to the Gemini computer as a write device of the same access class as its controlling TDS. This actually creates a multi-terminal workstation with the PC running the JAMPS software and transmitting JINTACCS formatted messages with appropriate compromise and integrity rules enforced by the Trusted Computing Base (TCB).

Each of the TDS workstations consists of a terminal and a personal computer and may be configured with a printer for hard copies of transmitted and received messages. Each element of the workstation would have the same access class and therefore would be controlled by the system manager for enforcement of security.

It is recommended that a project be undertaken to transport the JAMPS software onto the Gemini Trusted Computing Base to eliminate the need for the additional PC and to have the message preparation process completely controlled by the TCB rather than attaching it as an untrusted process.

2. Intersegment Linkage of Multiple Language Applications

The Gemini multiple microcomputer system is a self-hosting environment for software development using the CP/M-86 operating system. Any programming language that runs on CP/M-86 potentially can be used to develop concurrent computing and multilevel secure application software. Gemini is currently supporting the following languages: PASCAL MT+, Janus/ADA, PL/1, C, and FORTRAN. Additional software development tools are provided by several utility programs supplied as a part of the Gemini applications development package. Currently, the PASCAL MT+ applications development environment is the most thoroughly developed. This environment contains libraries of ring 0 and ring 1 calls and routines. The other languages have been supported for specific Gemini clients. An important aspect of this thesis was a demonstration of the use of ADA in a secure environment, therefore, the application development package provided by Gemini needed to be updated and more completely developed to support this project. As a result, additional libraries have been developed to support the ring 0 and ring 1 calls for Janus/ADA to bring applications support to a level comparable with PASCAL MT+. These libraries are listed in Appendices I and J.

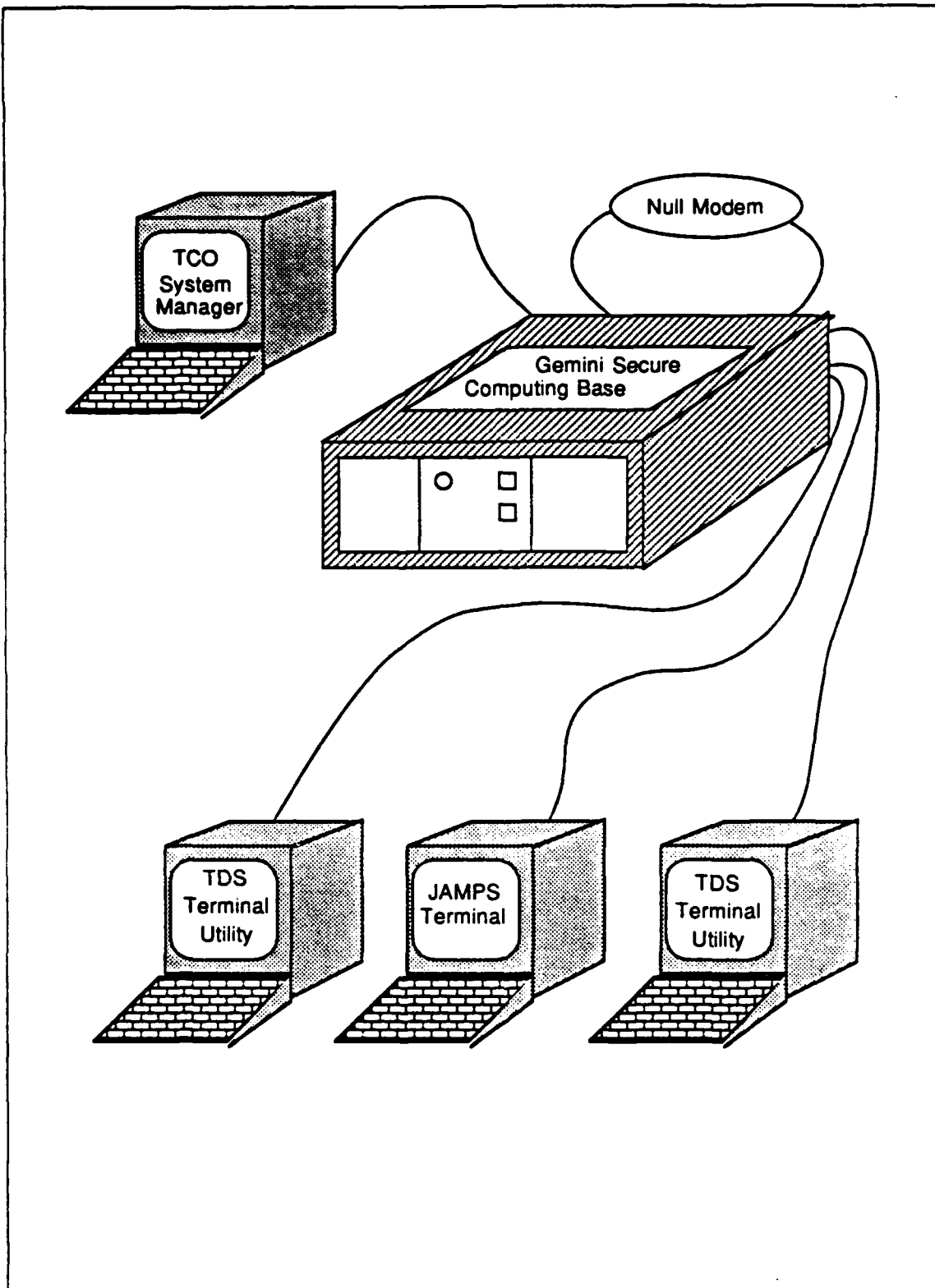


Figure 4.1 Revised Hardware Configuration.

All information in a Gemini system is contained in discrete, logical objects called segments. Each segment has an access class that reflects the sensitivity of information contained in the segment. Segments may be simultaneously shared by multiple subjects, but access to the segment on the part of each subject is controlled by the relationship between the segment's access class and each subject's access class.

Every segment in GEMSOS has a unique identifier. This identifier is effectively different for every segment ever created in any Gemini system. Unique identifiers are used to prevent "spoofing" of the system by substituting one segment for another.

Different languages pass information and identifiers to segments in different formats and using different protocols. Assembly routines can be used to "translate" the information and protocols in order for subjects created in different languages to access shared segments. This inter-segment linkage is necessary in the secure GEMSOS environment for the kernel to enforce the security rules in the trusted environment.

The difficulty of inter-segment linkage can be reduced by creating separate processes in different languages with their own code, stack, and data segments. By creating separate processes, no assembly routines are required because, once the source code is compiled, it is essentially identical and no protocol or translation is necessary. The multi-language code segments can be called and implemented without restriction.

The final design included the system manager, a trusted process written in PASCAL MT+, and untrusted TDS terminal utility processes written in PASCAL MT+ and Janus ADA. Each TDS terminal utility was accessed and controlled by the trusted process. Since each TDS utility process was separately compiled and linked, there was no need for an assembly routine to translate information between segments. Therefore, any language which can be compiled and linked into a CP/M-86 environment could be used to create untrusted processes.

3. Communications

In order to create a realistic communications link, it was necessary to simulate having untrusted computer systems communicating with each other. This was accomplished by having the Gemini system communicate with itself using separate I/O ports. By routing the incoming and outgoing traffic from each port to separate processes, the "multiple computer" environment could be simulated. The interface was controlled by the multilevel security manager located at a data terminal linked to the Gemini system. This security manager was responsible for:

- System start-up, initialization and testing.

- Assigning access levels for other terminals.
- Entering the cryptographic key.
- Control of communications to and from the external ports.
- Routing of message traffic to the appropriate terminals.
- Enforcing rules of integrity and compromise throughout the communications network.

Each TDS terminal was allowed to input JAMPS formatted messages (using the attached PC running the JAMPS software), transmit messages, and display incoming messages provided the terminal at which they were located had the proper access level.

A typical scenario might include:

- System startup, login, and initialization.
- Input of parameters such as communications ports, access class for each TDS terminal, and buffer sizes.
- Communications and encryption testing.
- Creation of the child process for each TDS terminal.
- Entering the transmit/receive loop.
- Message creation (activation of JAMPS software)
- Message transmission.
- Message display
- Enforcement of security rules and report of security violations.
- Detaching and terminating child processes.
- System shutdown.

Two way communications between TDS terminals could easily be maintained without error, while ensuring that the rules of integrity and compromise were enforced.

a. System Upgrade

Initially, the Gemini computer installed in the Wargaming, Analysis, and Research (WAR) laboratory was inadequate for development of a suitable demonstration of the scope of this thesis. Initial development was done on site at Gemini Computers in Monterey. During the development of the demonstration it was determined that a system upgrade, to include memory expansion and a global serial board (consisting of 8 serial ports) was necessary. A machine was configured at Gemini and used until an upgrade could be accomplished on the Gemini computer in the wargaming laboratory. Once this upgrade was accomplished the demonstration developed for this thesis could be modified slightly and used as a working, multilevel

communications network to support wargaming and research in a multilevel secure environment.

4. Demonstration of GEMSOS Security Mechanisms

The fourth task of this thesis was to demonstrate how GEMSOS would enforce security mechanisms to prevent unauthorized access to sensitive information. To demonstrate the enforcement of security mechanisms, a scenario was developed with planned attempts at security violations. Specifically, attempts were made to transmit messages to terminals at lower access levels (attempting to violate integrity) and attempts were made to read messages written at higher access levels (violation of compromise). In each case the system manager was able to interrupt the transmission and stop any violation of security. These tests were meant to demonstrate, rather than to prove that information security are preserved. They are in no way intended to be exhaustive, however, they allow for a series of observations to be made concerning system security.

C. SYSTEM TESTING

The test phase was designed to demonstrate particular security features of the model guard interface system. It was not intended to prove that the security of the system could be violated, but to demonstrate the security mechanisms enforced in the GEMSOS environment. Testing consisted of two major areas.

First, communications were established between users having the same access level. Messages were passed between the two terminals via the multilevel communications process. Initially, the multilevel system manager was created to coordinate the communications between two unclassified users. The system manager assigned the access classes of the TDS terminals and all information passed between the two was encrypted, transmitted, received, and decrypted by the system manager.

The second area was to test the enforcement of security rules as discussed in previous sections. To test this, TDS terminals were attached to ports with different access classes and the system manager assigned access classes to the TDS processes at or below the access class of the port. When messages were sent between the TDS terminals, the trusted process enforced the rules of compromise and integrity, and upon detecting a security violation it would issue the appropriate error message to the originator. In this case the security check consisted of a comparison of the incoming message header, with the system manager defined destination access level. The error

message interrupted the normal sequential passing of messages, to inform the originator that the destination of his message did not have the proper access level to receive the message.

Each TDS terminal operated independently, simulating being located at two different activities. They could send and receive messages from different physical ports, and communicate to each other using different external ports. Inter-process synchronization was accomplished by allowing only one terminal to send a message at a time. Once a terminal's message transmission was completed, control was passed to the other terminal to allow it to display the incoming message. This technique was chosen to facilitate testing. It is not the only method which could have been used. This technique would not be adequate for real-time communications flow. The software developed for this demonstration would have to be modified and updated with a view toward pipeline processing, making maximum use of the multi-processor Gemini system. The multiple microcomputers in a Gemini system are capable of multiprocessing as well as multiprogramming. Depending on the requirements of the application, the GEMSOS can multiplex processes onto a single processor or distribute them among several processors to support combinations of parallel and pipeline processing. A timed polling scheme with all terminals operating simultaneously would be appropriate and would be an avenue to take this particular application for further study.

D. OBSERVATIONS AND LESSONS LEARNED

The following sections provide a set of observations based upon the experience of developing a trusted application on the Gemini Trusted Multiple Microcomputer Base.

1. Applications Development with GEMSOS

Development of applications programs in a secure environment is more complicated than development in a nonsecure environment. Applications development tools such as example programs, demonstrations and libraries greatly speed up the process of preparing an application for the secure environment. Successful compilation and linking are the first step for development. Once linked with the appropriate modules, an application still has no security associated with it. To assign a security classification and prepare the program to execute in GEMSOS, a secure volume must be created by running the system generation (SYSGEN) program. This process places the application in the segment structure recognized by GEMSOS and creates a

bootable system segment structure on a formatted volume. The key to proper use of the SYSGEN program is identification of the segment structure in which the application will be placed. The segment structure includes the boot-strap, the kernel, the application code, and data segments. Understanding the format of your application is very important for system generation.

As with any new area of study, multilevel security has its own terms and concepts which must be thoroughly understood prior to implementation. As discussed in Chapter II, the manner in which the Gemini system manages resources is not like unsecure systems. The interaction of the process, segment and device management functions is key to understanding overall system operation.

Another step in the education process for secure applications development is the rigid control that the secure operating system must maintain over subjects and objects. A programmer cannot think in terms of "read" and "write" as in typical PASCAL programming. Commands such as these compile successfully, but cause trapping in the secure operating system. Once the programmer understands the restrictions of a secure environment and develops the "mind set" for programming of secure applications, the development of secure applications becomes routine.

Gemini provides applications development packages with their machines to make the development environment more friendly. Libraries, utilities and demonstrations are constantly being updated and revised as developments in secure technology are made. Any programmer with experience with 8086 or 80286 system programming and an understanding of high order languages can easily adapt to programming in the GEMSOS environment.

2. Development of Applications Using ADA

This section addresses the question: "How can a trusted computer system architecture be defined so that untrusted software applications programs can be coded in ADA and DoD security policy still be enforced?"

The 'Orange Book' requires that a system be divided into two pieces, the trusted computing base and the untrusted software. The trusted computing base is responsible for enforcing the system security policy. System security is independent of the untrusted software. Models of DoD security include the concepts of subject and object. The trusted computing base must assure that all accesses of subjects to objects adhere to the system policy. How can security subjects be defined in a system using ADA programs so that this is possible?

The ADA task is the active entity in an ADA program. Therefore, the task is the obvious candidate for a subject. However, selection of an ADA task as the subject leads to problems. In general, different tasks within the same ADA program can communicate bidirectionally in the following ways:

- They can rendezvous.
- They can read and change global data.
- They can use a common global package, which might change data that is private to the package.
- They can open, read, write, and close the same devices.

In most security models, each subject can be assigned a different security level. But two way communication between tasks at different security levels would violate the security policy. Thus, if different tasks in the same program are different subjects at different levels, the kernel must eliminate the two way communication between them. It must mediate each rendezvous, mediate access to shared global data, mediate access to global packages, and it must assign devices separately to separate tasks in a program. This type of approach becomes somewhat complicated. A direct approach to the solution is the use of a trusted subject to control each access or communication. The Gemini system does this by using the kernel as the trusted process and using the ADA task as an application built over the GEMSOS kernel. Tasks in different programs can communicate using the host operating system interprocess communications mechanisms. The system is as secure as the host operating system; system security is as independent of ADA as it is any application development language.

3. Computer Security for Marine Tactical Data Systems

The protection of information processed, stored and transmitted by Marine tactical data systems is vital to Marine Air Ground Task Force safety and imposes stringent requirements for system security. This protection is provided by hardware and software features, physical security measures, procedural security, and access controls. These features must provide in-depth protection of sensitive information as well as ensuring the integrity of transmitted data.

Presently we can have only limited security because of the untrusted components. Currently, security is enforced using only restrictive security policies and procedures such as:

- System high modes of operation.
- Dedicated modes of operation.

- Controlled modes of operation.

We must continue to build trusted perimeters around the untrusted components, subsystems and systems using methods which provide security when information is created such as:

- Trusted smart terminals.
- Trusted PCs.
- Trusted workstations.
- Trusted network interfaces.
- Trusted gateways

We must also provide security when information is released outside the security perimeter such as:

- Trusted guards.
- Trusted network interfaces.
- Trusted front end processors

These protection mechanisms come as "add-on" type security. We cannot limit ourselves to this "band-aid" approach. We must begin to develop "built-in" security. In order to do this we must think from the ground up. This can be done by designing and building systems consisting of untrusted components and trusted components developed together to form trusted systems, and by building completely trusted systems by using all trusted components from conception. Future security of Marine tactical data systems requires that we understand our needs, develop a rigid security policy and develop our command and control systems with security in mind from conception.

4. Integrated Security Requirements

Total security of Marine tactical data systems is the integration of:

- Physical security through the proper use of access controls and creating a secure environment.
- Personnel security with all personnel having appropriate clearances and "need-to-know".
- Administrative security to include a rigid security policy, appropriate responsibility, well established procedures, access controls, planning, and back-up.
- Computer security
- Communication security
- Information security

Every aspect of system security must be reviewed to make the Marine Corps' command and control structure secure. We cannot think in terms of physical security alone. The

security policy must be an integrated, well thought out plan protecting against every vulnerability. Figure 4.2 from the 1967 Joint Computer Conference shows what we are up against when evaluating the vulnerabilities of any information processing system. Secure front end processors, trusted guards, encryption, and physical security are all components in the secure system.

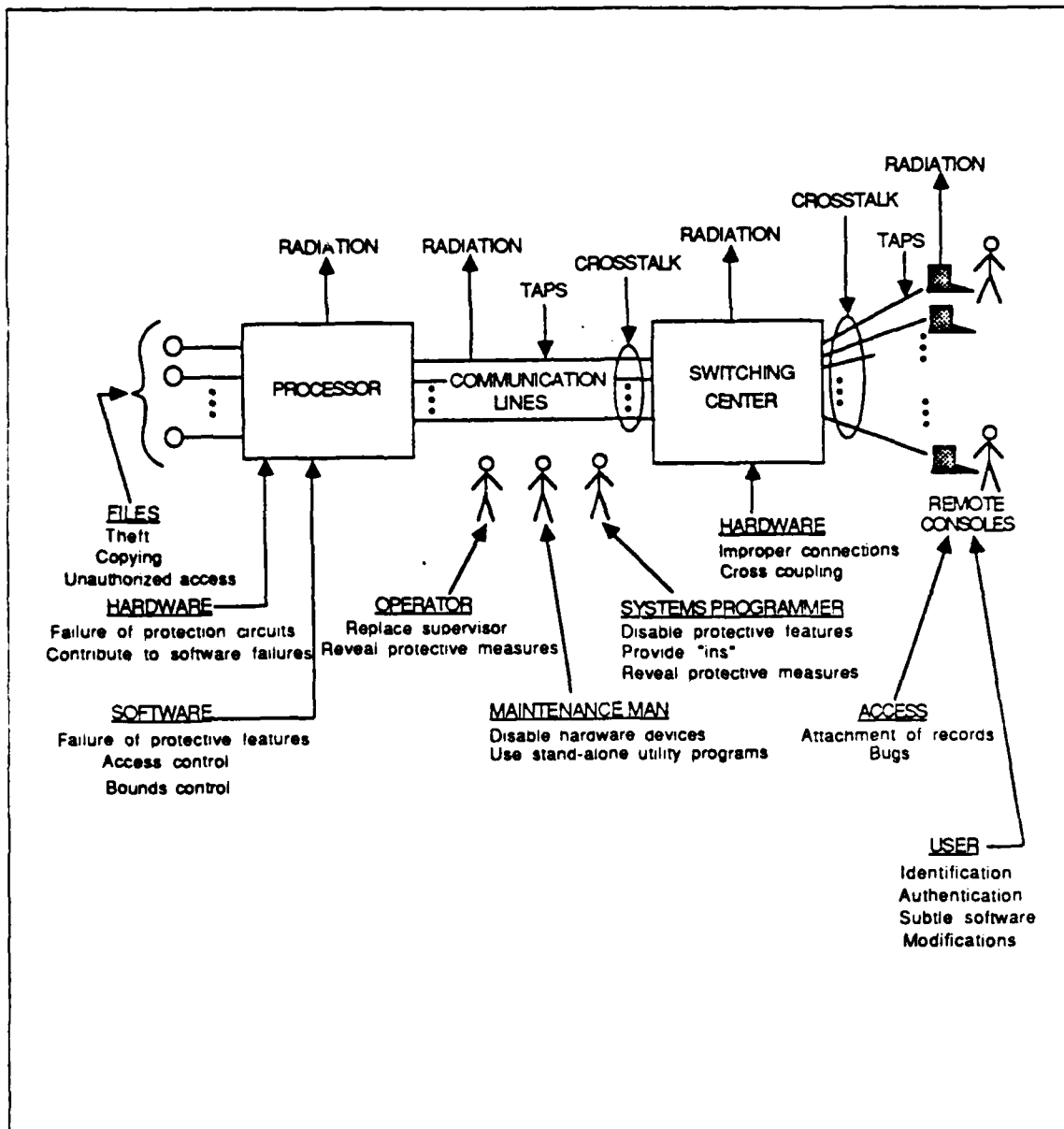


Figure 4.2 Computer System Vulnerabilities.

E. SUMMARY

The demonstration of a trusted interface between tactical data systems developed in this thesis met or exceeded all design goals. Messages were passed between two remote terminals simulating tactical data systems in a manner that ensured security from unauthorized access at both source and destination. Encryption was used to maximize security between terminals. By varying the access class of each TDS terminal, it was possible to demonstrate the system's ability to detect and respond to security violations. Flexibility in determining system configuration allows for modification of system parameters to meet a variety of requirements. Finally, by using modular programming techniques, the application may be altered to fit various demands based upon the particular security policy of an organization.

V. CONCLUSIONS

A. GENERAL

The multilevel secure interface designed in this thesis was developed to demonstrate the secure interface between untrusted, single level tactical data systems operating at different security levels. The system developed is capable of supporting the control of the dissemination of information either with or without the presence of a security officer. The principle security mechanism demonstrated is the enforcement of integrity and compromise by a reference monitor (security kernel) implemented by the Gemini Secure Operating System.

Most of the Marine Corps' tactical data systems rely solely on a physical security perimeter, protecting the computer and its users by guards, fences and other means. Communications between computer and remote devices may be encrypted to geographically extend the security perimeter. When only physical security is used, all users can potentially access all information in the computer system. Consequently, all users must be trusted in the same degree. When the system contains sensitive information that only certain users should access, additional protection mechanisms must be placed in the system. One solution is to give each class of users a separate machine. This solution is becoming increasingly less costly because of declining hardware prices. However, the proliferation of computers and the networks to interconnect them means that in the future, the potential users of many computers will include individuals not authorized access to much of the growing amount of valuable and sensitive information they contain.

It is becoming increasingly clear that in the future, many computers will need interconnections to other computers in order to do their jobs. JCS Pub 1, Jan 86 defines interoperability as:

The condition achieved among communications-electronics systems, or items of communications-electronics equipment, when information or services can be exchanged directly and satisfactorily between them and/or their users.

The increasing number of tactical data systems in the Marine Corps and all of the services leads to the absolute need to connect in some way to pass the wealth of

information to and from the multitude of subscribers. Information is becoming a national resource and protecting information is vital. Advanced tactical data systems facilitate the exchange of information, but unless they address the controlled sharing of information among users, the information could easily do us as much harm as good. Sharing information from a computer requires internal controls to isolate sensitive information from unauthorized users. The notion of controlled sharing implies that it is possible to define what controls on sharing are desired. This definition constitutes the security policy. Thus, in addition to defining interoperability of tactical data systems there must be a security policy defined as well.

There are two types of security policy: mandatory and discretionary. A mandatory policy contains security rules which are imposed on all users. A discretionary policy, on the other hand, contains security rules that can be specified at the option of each user. The protection policy enforced by a security kernel is encapsulated in a set of rules that constitute a formal security policy. Today the security kernel is the only available technology for demonstrating secure computer systems of practical proportions. During the past decade the maturation of this technology and the improvements in software engineering and microelectronics make the security kernel practical and affordable. The Marine Corps has a workable solution to the computer security problem through the use of a secure kernel as a guard between untrusted systems or as a trusted front end process linking tactical data systems at multiple levels of security.

B. RECOMMENDATIONS FOR FURTHER STUDY

1. Integration of JAMPS into GEMSOS

JAMPS is a portable set of computer controlled equipment for assisting operators in composing and exchanging JINTACCS formatted messages. JAMPS was used in the demonstration as an untrusted process solely for the creation of JINTACCS messages. It is recommended that JAMPS be fully integrated into GEMSOS and be run as a trusted process. Transport of the JAMPS software onto a trusted machine would establish a secure, multilevel workstation for creation, storage, and communication of JINTACCS formatted messages.

2. Implementation of a Polling Scheme

The current communications process created for this demonstration used an event driven environment which only allowed one terminal to have control at any

particular time. The establishment of an interrupt driven environment could make maximum use of the multiprogramming and multiprocessor environment of GEMSOS. A polling scheme or an interrupt driven environment would allow each terminal process more access to the multiple processors in the Gemini system and would create a more real time, secure communications system.

3. Modern Programming Techniques

The technique of top down, sequential programming was used for this demonstration. This was simply due to inexperience with working with a multiprocessor system. A completely different approach must be used to take full advantage of the Gemini multiple microprocessor environment. Techniques such as parallel, pipeline, and concurrent processing must be used for applications development. A new "mind set" must be established to think of these advanced techniques when developing applications. A topic recommended for further study is the restructuring of this demonstration using parallel and pipeline processing techniques.

APPENDIX A

GLOSSARY

Access - A specific type of interaction between a subject and an object that results in the flow of information from one to another.

Access Level - The combination of the security level and the integrity level of a subject or object.

Audit Trail - A set of records that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions.

Bell-LaPadula Model - A formal state transition model of computer security policy that describes a set of access rules. In this formal model, the entities in a computer system are divided into abstract sets of subjects and objects. The notion of a secure state is defined and it is proven that each state transition preserves security by moving from secure state to secure state; thus, inductively proving that the system is secure. A system state is defined to be "secure" if the only permitted access modes of subjects to objects are in accordance with a specific security policy. In order to determine whether or not a specific access mode is allowed, the clearance of a subject is compared to the classification of the object and a determination is made as to whether the subject is authorized for the specific access mode. The clearance/classification scheme is expressed in terms of a lattice.

Category Set - A category set, part of a security level, is a list of information categories applicable to an object or subject. Categories correspond roughly to the topic areas of the information. A subject must be cleared for all categories on an object to read the object.

Channel - An information transfer path within a system. May also refer to the mechanism by which the path is effected.

Compartment - Compartments are a mutually exclusive way to assign categories. If an object is compartmented, then it generally has only one category, called a compartment. Compartments are used mostly in the intelligence community. They are implemented in a kernel-based system using categories.

Computer - Any device capable of storing and processing information and, if linked by a network, of communicating with other computers. Computers used in this manner are commonly referred to as Hosts, as contrasted with those used in communications applications, called Switches.

Controlled - Some users having neither a security clearance nor a need-to-know for some information processed by the system, but separation of users and classified material not essentially under operating system control.

Covert Channel - A communication channel that allows a process to transfer information in a manner that violates the system's security policy.

Dedicated Security Mode - All system (Computer or Network) equipment used exclusively by that system, and all users cleared for and having a need-to-know for all information processed by the system.

Descriptive Top-Level Specification (DTLS) - A top-level specification that is written in a natural language, an informal program design notation, or a combination of the two.

Discretionary Access Control - A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission on to another subject.

Discretionary Security - The aspect of security policy implementing the "need-to-know" requirement for access to information.

Domain - A set of objects that a subject has the ability to access.

Flow Control - A strategy for protecting the contents of information objects from being transferred to objects at improper access levels.

Formal Top-Level Specification (FTLS) - A Top-Level Specification that is written in a formal mathematical language to allow theorems showing the correspondence of the system specification to its requirements to be hypothesized and formally proven.

Formal Verification - The process of using formal proofs to demonstrate the consistency between a formal specification of a system and a formal security policy model or between the formal specification and its program implementation.

Mandatory Access Control - A means of restricting access to objects based on the sensitivity of the information contained in the objects and the formal authorization of subjects to access information of such sensitivity.

Multilevel - Some users having neither a security clearance nor a need-to-know for some information processed by the system: separation of personnel and material accomplished by the operating system and associated software.

Multilevel Device - A device that is used in a manner that permits it to simultaneously process data of two or more security levels without risk of compromise. To accomplish this, sensitivity labels are stored on the same physical medium and in the same form as the data being processed.

Multilevel Secure - A class of system containing information with different sensitivities that simultaneously permits access by users with different security clearances and needs-to-know, but prevents users from obtaining access to information for which they lack authorization.

Network - An entity composed of any of a number of communications media used to link Computers and transfer information.

Non-discretionary (Mandatory) Security - The aspect of DoD security policy which deals with security levels. A security level is comprised of a security classification and one or more categories of access restriction. Classifications are totally ordered while categories are partially ordered. To access a piece of information, a user must have a classification greater than or equal to the classification of the information, and at least all of the categories of access restriction of the information.

Object - A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are: records, blocks, pages, segments, files, directories, directory trees, and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, network nodes, etc.

Process - A program in execution. It is characterized by a single current execution point and address space.

Read - A fundamental operation that results only in the flow of information from an object to a subject.

Reference Monitor Concept - An Access control concept that refers to an abstract machine that mediates all access to objects by subjects.

Security Kernel - The hardware, firmware, and software elements of a Trusted Computing Base that implement the reference monitor concept. It must mediate all accesses, be protected from modification, and be verifiable as correct.

Security Policy - The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information.

Sensitivity Label - A piece of information that represents the security level of an object and that describes the sensitivity of the data in that object. Sensitivity labels are used by the TCB as the basis for mandatory access control decisions.

Single-Level Device - A device that is used to process data of a single security level at any one time. Since the device need not be trusted to separate data of different security levels, sensitivity labels do not have to be stored with the data being processed.

Subject - An active entity, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state. Technically, a process/domain pair.

System - A collection of two or more computers linked by a network.

System High - All equipment protected in accordance with requirements for the most classified information processed by the system. All users cleared to that level, but some not having a need-to-know for some of the information.

Top-Level Specification - A non-procedural description of system behavior at the most abstract level. Typically a functional specification that omits all implementation details.

Trusted Computer System - One which employs sufficient hardware and software integrity measures to allow its use for processing simultaneously a range of sensitivity or classified information.

Trusted Computing Base - All of the protection mechanisms within a computer system (hardware, firmware, and software) which enforce a security policy on that computer. It creates a basic protection environment and provides additional user service required for a trusted computer system.

Trusted Network Base - All the protection mechanisms within a network which enforce a security policy on that network.

Trusted System - One which employs sufficient hardware and software integrity measures to allow its use for processing simultaneously a range of sensitivity or classified information.

Verification - The process of comparing two levels of system specification for proper correspondence (e.g., security policy model with top-level specification, TLS with source code, or source code with object code). This process may or may not be automated.

APPENDIX B

LIST OF ACRONYMS AND ABBREVIATIONS

AADCCS - Army Air Defense Command and Control System
AAW - Antiair warfare
ABIC - Army Battlefield Interface Concept
ACE - Aviation Combat Element
ADOC - Air Defense Operations Center (NATO)
AFATDS - Advanced Field Artillery Tactical Data System (Army)
AMASS - Advanced Marine Airborne SIGINT System
AOC - Air Operations Center (Navy)
ARTADS - Army Tactical Data System
ASCII - American Standard Code for Information Exchange
ATACC - Advanced Air Tactical Command Central
ATF - Amphibious Task Force
Btry - Battery
BUCS - Backup Computer System
C3 - Command, Control and Communication
CAS - Close Air Support
CCP - Communications Control Panel
CDU - Control and Display Unit (GPS)
CLF - Commander Landing Force
COMSEC - Communication Security
DASC - Direct Air Support Center
DCC - Display Control Console (PLRS)
DSCS - Defense Satellite Communications System
DSD - Dynamic Situation Display
EMCON - Emission Control
FAMDS - Field Artillery Meteorological Data System
FASC - Fire and Air Support Center
FDC - Fire Direction Center
GPS - Global Positioning System
IAC - Intelligence Analysis Center (MAGIS)
JAMPS - JINTACCS Automated Message Preparation System

JINTACCS - Joint Interoperability of Tactical Command and Control Systems
LFICS - Landing Force Integrated Communications System
MAB - MArine Amphibious Brigade
MACCS - Marine Air Command and Control System
MAGIS - Marine Air-Ground Intelligence System
MAGTF - Marine Air-Ground Task Force
MATCALS - Marine Air Traffic Control and Landing System
MIFASS - Marine Integrated Fire and Air Support System
MPIC - MIFASS-PLRS Interface Controller
MRAALS - Marine Remote Area Approach and Landing System
MTACCS - Marine Corps Tactical Command and Control Systems
MTDS - Marine Tactical Data System
NCCS - Navt Command and Control System
PLI - Position Location Information
PLRS - Position Loation Reporting System
REAL FAMMIS - Real-Time Finance and Manpower Management Information System
SIGINT - Signals Intelligence
TAC - Tactical Air Commander
TACC - Tactical Air Command Center
TACFIRE - Tactical Fire Direction System (Army)
TAOM - Tactical Air Operations Module
TCC - Tactical Communications Center
TCO - Tactical Combat Operations System
TDS - Tactical Data System
TERPES - Tactical Electronic Reconnaissance Processing and Evaluation System (MAGIS)
TIC - Technical Interface Concept
TWSEAS - Tactical Warfare Simulation, Evaluation, and Analysis System
WWMCCS - Worldwide Military Command and Control System

APPENDIX C

INTERFACE USERS GUIDE

This Demonstration simulates a secure multi-level system manager and two unsecure TDS terminals. The unsecure TDS terminals may create JINTACCS formatted messages by transferring control to a PC operating the JINTACCS Automated Message Preparation System (JAMPS) and may then transmit and receive messages from other TDS terminals.

All messages must first pass through the trusted system manager. The system manager computes a checksum on the message header, which consists of the source and destination, the access classes, the message number and the number of blocks in the message. The checksum and the message are encrypted and transmitted to the destination system manager (guard). All messages must first pass through the system manager. The receiving system manager computes a checksum on the message and compares this checksum with the checksum that was sent along with the message. If the two checksums are not exactly the same, the message cannot be transferred to a TDS terminal. The encrypted checksums will not match if the message contents or clearance level has been modified during the transmission of the message. If the checksums are the same, the message is sent to the TDS terminal, if the file's clearance level is the same or less than the TDS terminal.

The demonstration uses the standard loader and login processes. The standard loader process creates a system manager. The system manager creates the two processes that actually implement the demonstration. The two processes that implement the TDS terminal utilities of the demonstration are called the TDS terminal utility processes.

The system manager attaches the Local SIO port 100, which must be connected to a terminal, so it can prompt the operator to input parameters such as how many terminals the demonstration will use, the ports to attach as transmit and receive devices and the size of the message buffers.

The trusted process attaches the GSIO ports 0 and 3, for reading and writing, so that the TDS processes may display a menu of options to the operator. The operator is allowed to input, transmit and display messages. Each TDS process also attaches an LSIO port to allow a PC running JAMPS to allow it to create the JINTACCS formatted messages.

When an operator requests to input a message, control is shifted to the PC and a JINTACCS message may be created using JAMPS. Once the message is input control is shifted back to the TDS terminal and the menu returns allowing the terminal to transmit the message to another terminal through the system manager.

When the system manager transmits the message to another TDS terminal, control is shifted to that terminal allowing it to display the message. That terminal now has control and may create a message to be transmitted, and the cycle continues until a security violation creates an error or one of the terminals logs off the network. Control is then shifted back to the system manager and the process may begin again.

The menu that the TDS terminal displays is:

Enter mode desired

I = INPUT MESSAGE

D = DISPLAY MESSAGE

X = TRANSMIT MESSAGE

E = LOGOFF

ENTER MODE HERE

The "INPUT" option allows the operator to shift control to the PC running JAMPS and create a JINTACCS formatted message. Once the message is created, control shifts back to the terminal and the operator is prompted for the destination terminal. If the operator enters a destination terminal with an improper access level, an error message is sent to the system manager and the message "IMPROPER DESTINATION TRY AGAIN" is displayed on the TDS screen. After the proper destination has been entered, the original menu reappears.

The "TRANSMIT" option allows the operator to attempt to send a message. When the operator selects "X", control is shifted to the system manager and the system manager is prompted for an encryption key. Once the key is entered the message is encrypted, transmitted, received, and decrypted. If there has been no security violation the message is placed in the destination terminal's read buffer and control is shifted to the destination terminal. The menu now appears on his screen and the operator may enter "D" to display the received message. If the operator tries to retrieve a message that does not exist, the error message, "INCOMING MESSAGE BUFFER EMPTY" is displayed.

The following is a list of all the segments used in this demonstration (segments are listed using pathnames):

- 5 Mentor segment for all segments used for demo.
- 5,0 System manager code segment.
- 5,2 TDS1 terminal utility code segment.
- 5,3 TDS2 terminal utility code segment.

APPENDIX D

SYSTEM MANAGER PROGRAM LISTING

The source code for the system manager is written in in PASCAL MT+. With the exception of GUARD-CON.ZLI (Appendix F) and GUARD-TYP.ZLI (Appendix G), all include files are library utility files which were delivered as part of the development environment for the Gemini operating system. Information concerning how to invoke library functions is contained in [Ref. 20]. Once the source file is compiled, the required modules are linked together by using the GUARD.KMD file with the PASCAL MT+ linker. A listing of this file is provided as Appendix E. Upon completion of the linking process, the resulting GUARD.CMD must be prepared to run in the Gemini Secure Operating System (GEMSOS) environment. This is accomplished by operating the system generation (SYSGEN) program and using the GUARD.SSB submit file provided as Appendix M. Procedures for operating the SYSGEN program are contained in [Ref. 19]. Once the secure volume is created, the Gemini system is reinitialized using the secure volume. This begins the execution of the system manager process.

{*****}

MODULE -- GUARD.PAS

DATE -- 6 FEBRUARY 1987

AUTHOR -- G. E. RECTOR CAPT/USMC

ADVISOR -- T. J. BROWN MAJ/USAF

PURPOSE -- This module is initialized as a multi-level process which allows a terminal attached to port 0 to simulate the multilevel TIMS workstation as the system manager. It allows the system manager to configure and operate a multilevel secure interface between TIMS and single level, untrusted terminals simulating untrusted tactical data systems. Once the system is initialized, configured and tested, it runs independently allowing remote terminal users to transmit messages via the multilevel secure interface.

*****}

module guard;

{ constant include files }

const

{Si gate-con.zli} {Gate constant include file}
{Si r1-con.zli} {Ring 1 constant include file}
{Si guard-con.zli} {Application constant include file}

{ type include files }

type

{Si gate-typ.zli} {Gate type include file}
{Si lib-typ.zli} {Library type include file}
{Si rlp-typ.zli} {Ring 1 type include file}

```

{Si kst-typ.zli}    {Known segment table type include file}
{Si guard-typ.zli} {Application type include file}

```

```

{ library include }

```

```

{Si io-dec.zli}    {I/O integer manipulation include file}
{Si io-hex.zli}    {I/O hexadecimal include file}
{Si seg-mgr.zli}   {Segment manager include file}
{Si gate.zli}      {Gate include file}
{Si lib.zli}       {Library include file}
{Si lib-b24.zli}   {Library include file}
{Si io-str.zli}    {I/O string manipulat in include file}

```

```

{ *****

```

Procedure: input_param

Purpose: This procedure allows the system manager to input parameters necessary to setup and test system operation.

```

*****}

```

```

procedure input_param(    class    : access_class;

```

```

                        var sys_rec : sysmgr_rec;
                        var proc_suc : boolean);

```

```

var

```

```

    temp_str:string;
    temp_char:char;
    temp_int:integer;
    i:integer;

```

```

begin {input_param}

```

```

    putln(w_dev,'PLEASE ENTER THE SYSTEM PARAMETERS');

```

```

    putln(w_dev,'ENTER 1ST PHYSICAL PORT USED FOR EXTERNAL COMM ');

```

```

getchar(r_dev, temp_char);
putchar(w_dev, temp_char);

sys_rec.comm_port[1] = ord(temp_char)-48;
putln(w_dev, ' ');

putln(w_dev, 'ENTER 2ND PHYSICAL PORT USED FOR EXTERNAL COMM ');
getchar(r_dev, temp_char);
putchar(w_dev, temp_char);

sys_rec.comm_port[2] = ord(temp_char)-48;
putln(w_dev, ' ');

sys_rec.ch_size = 400;

putln(w_dev, 'CHILD SIZE IS ');
putdec(w_dev, sys_rec.ch_size);
putln(w_dev, ' ');

putln(w_dev, 'BUFFER SIZE IS 100 BYTES');

sys_rec.b_size = 100;
putln(w_dev, ' ');

putln(w_dev, 'ENTER TERMINAL ACCESS LEVEL');
putln(w_dev, 'UNCLASS = 0');
putln(w_dev, 'CONF = 2');
putln(w_dev, 'SECRET = 4');
putln(w_dev, 'TOP SECRET = 6');

putln(w_dev, 'ENTRY MUST BE WITHIN ACCESS RANGE');

for i = 1 to num_term do begin

    putstr(w_dev, 'TERMINAL ');
    putdec(w_dev, i);
    putstr(w_dev, 'ACCESS LEVEL IS ');
    getchar(r_dev, temp_char);
    putchar(w_dev, temp_char);
    temp_int = ord(temp_char)-48;

```

```

temp_int := temp_int & 50007;
sys_rec.ch_access[i] := class;
sys_rec.ch_access[i].compromise[0] :=
    class.compromise[0] & 5fff8;
sys_rec.ch_access[i].compromise[0] :=
    sys_rec.ch_access[i].compromise[0] ! temp_int;
putln(w_dev, ' ');

end; {for}

putln(w_dev, 'ENTER 8 CHARACTER INITIAL CRYPTO KEY (NO ECHO)');

for i:= 1 to 8 do begin

    getchar(r_dev, temp_char);
    sys_rec.key[i] := ord(temp_char);

end; {for}

putln(w_dev, 'CRYPTO KEY INSERTED');

    { *** fixed parameters *** }

    { code segment entry numbers }

sys_rec.chld_ent[1] := 2;
sys_rec.chld_ent[2] := 3;
putln(w_dev, 'ENTRY OF PARAMETERS IS COMPLETE');

end; {input_param}

```

```

{ ****

```

Procedure: sys_config

Purpose: This procedure configures the external communication ports identified in parm_input for port to port communications with flow control. They are attached to read and write sequentially 8 bytes at a time to be compatible with the data encryption device.

```

*****}

procedure sys_config( send_port: integer;
                      rcv_port: integer;
                      var config_suc:boolean);

var

    rd_dev,wr_dev: dev_name;
    rd_parm,wr_parm: dev_parm_rec;
    success: integer;

begin { sys_config }

    config_suc:= false;

    putln(w_dev,'CONFIGURING TRANSMIT AND RECEIVE PORTS');

    { attach xmit and rcv ports for computer to computer communications }

        { *** fill-in attach_device calling arguments *** }

            { *** receiver should be attached first *** }

    rd_dev.name:= sior;
    rd_dev.num:= rcv_port;
    rd_dev.d_type:= io;

                                { device mode entries }

    rd_parm.sior.mr1:= S04d;
    rd_parm.sior.mr2:= S03e;
    rd_parm.sior.io_mode:= rts_oflow;
    rd_parm.sior.max:= 8;
    rd_parm.sior.delim_active:= false;

    attach_device(rd_dev,rcv_slr,rd_parm,success);
    show_err(' RECEIVER ATTACH ERROR',success);

        { *** attach transmitter *** }

```

```

wr_dev.name:= siow;
wr_dev.num:= send_port;
wr_dev.d_type:= io;

                                { device mode entries }

wr_parm.siow.mr1:= S04d;
wr_parm.siow.mr2:= S03e;
wr_parm.siow.io_mode:= asrt_none;

attach_device(wr_dev,xmit_slr,wr_parm,succes);
show_err('TRANSMITTER ATTACH ERROR',succes);

putln(w_dev,'COMMUNICATION DEVICES ATTACHED');

{ *** xmit and rcv attached computer to computer no flow control *** }

config_suc:= true;

end; {sys_config}

```

```

{*****}

```

Procedure: comm_tst

Purpose: This procedure checks communications in both directions by transmitting a test string of data. Once communications have been checked the comm devices are detached.

```

{*****}

```

```

procedure comm_tst(init: r1_process_def;
                    send_port: integer;
                    rcv_port: integer;
                    var comm_tst_suc: boolean);

```

```

var

```

```

    charin,charout: array [1..8] of char;
    wr_class,rd_class: access_class;

```

```

        i,succes: integer;
        size: integer;

begin { comm_tst }

comm_tst_suc:= false;

putln(w_dev,'BEGINNING COMMUNICATION TEST');

        { *** transmitter access_class for comm test *** }

wr_class.compromise:= init.resources.max_class.compromise;
wr_class.integrity:= init.resources.min_class.integrity;

putstr(w_dev,'TEST MESSAGE IS ');

for i:= 1 to 8 do begin

        charout[i]:= 'T';
        putchar(w_dev,charout[i]);

end; {for}

putln(w_dev,' ');

write_sequential(xmit_sl,addr(charout),8,wr_class.succes);
show_err('WRITE SEQUENTIAL ERROR',succes);

read_sequential(recv_sl,addr(charin),size,wr_class.succes);
show_err('READ SEQUENTIAL ERROR',succes);

for i:= 1 to 8 do

        putchar(w_dev,charin[i]);

putln(w_dev,' ');

detach_device(xmit_sl,succes);
show_err('TRANSMITTER DETACH ERROR',succes);

detach_device(recv_sl,succes);
show_err('RECEIVER DETACH ERROR',succes);

```

```
putln(w_dev,'COMMUNICATION TEST COMPLETE');
```

```
comm_tst_suc:= true;
```

```
end; { comm_tst }
```

```
{ ****
```

Procedure: att_crypto

Purpose: This procedure uses four process local device slots to attach the required encryption and decryption devices. Crypto key and feedback key are provided in the procedure call. Devices are attached using the cipher block chaining (CBC) mode.

```
****
```

```
procedure att_crypto(cry_key: buf8;
```

```
                  cry_fbkey: buf8;
```

```
          var att_crypto_suc: boolean);
```

```
var
```

```
    rendev,wendev,rdev,wdev: dev_name;
```

```
    ren_parm,wen_parm,wde_parm,rde_parm: dev_parm_rec;
```

```
    success: integer;
```

```
begin { att_crypto }
```

```
att_crypto_suc:= false;
```

```
    { *** attach read encryption device *** }
```

```
    rendev.name:= dcp_ren;
```

```
    rendev.num:= 0;
```

```
    rendev.d_type:= io;
```

```
    ren_parm.ren.blk_size:= 8;          { 8 bytes per blk }
```

```

attach_device(rendev,ren_sl,ren_parm,success);
show_err('ATTACH READ ENCRYPTION DEVICE ERROR',success);

    { *** attach write encryption device *** }

    wendev.name:= dcp_wen;
    wendev.num:= 0;
    wendev.d_type:= io;

    wen_parm.wen.mode:= 1;          { 1 for CBC mode }
    wen_parm.wen.key:= cry_key;
    wen_parm.wen.fb_key:= cry_fbkey;

attach_device(wendev,wen_sl,wen_parm,success);
show_err('ATTACH WRITE ENCRYPTION DEVICE ERROR',success);

    { *** attach read decryption device *** }

    rddev.name:= dcp_rde;
    rddev.num:= 1;
    rddev.d_type:= io;

    rde_parm.rde.mode:= 1;          { 1 for CBC mode }
    rde_parm.rde.key:= cry_key;
    rde_parm.rde.fbkey:= cry_fbkey;
    rde_parm.rde.blk_size:= 8;

attach_device(rddev,rde_sl,rde_parm,success);
show_err('ATTACH READ DECRYPTION DEVICE ERROR',success);

    { *** attach write decryption device *** }

    wddev.name:= dcp_wde;
    wddev.num:= 1;
    wddev.d_type:= io;

    { wde_parm is blank record }

attach_device(wddev,wde_sl,wde_parm,success);
show_err('ATTACH WRITE DECRYPTION DEVICE ERROR',success);

```

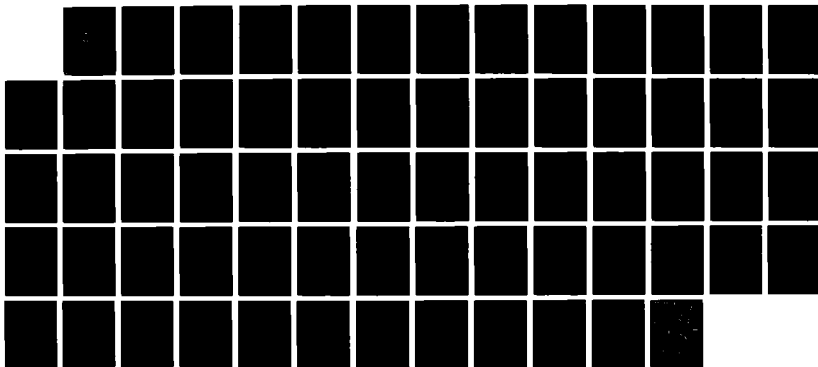
AD-A183 361

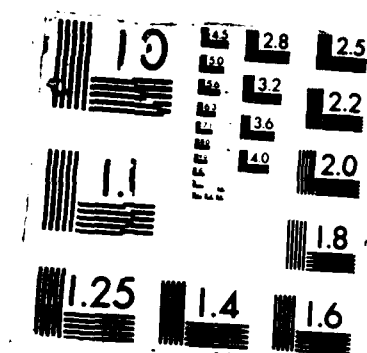
A DEMONSTRATION OF A TRUSTED COMPUTER INTERFACE BETWEEN 2/2
A MULTILEVEL SEC. (U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA G E RECTOR MAR 87

UNCLASSIFIED

F/G 25/5

NL





```
att_crypto_suc:= true;
```

```
end; { att_crypto }
```

```
{*****}
```

```
Procedure: crypto_tst
```

Purpose: This procedure verifies that the encryption and decryption devices are working properly. A test string is encrypted then decrypted using test keys. Results are output to the tims terminal. When complete all data ciphering devices are detached.

```
*****}
```

```
procedure crypto_tst(init: r1_process_def;  
                     crypto_tst_suc: boolean);
```

```
var
```

```
    encryptin,encryptout,decryptout: array [1..8] of char;  
    wr_class,rd_class: access_class;  
    size: integer;  
    i: integer;  
    success: integer;  
    proc_suc: boolean;
```

```
begin { crypto_tst }
```

```
crypto_tst_suc:= false;
```

```
putln(w_dev,'BEGIN ENCRYPTION DEVICE TEST');
```

```
wr_class.compromise:= init.resources.max_class.compromise;
```

```
wr_class.integrity:= init.resources.min_class.integrity;
```

```
putstr(w_dev,'ENCRYPTION TEST STRING IS ');
```

```

for i:= 1 to 8 do begin
    encryptin[i]:= 'T';
    putchar(w_dev,encryptin[i]);

end; {for}

putln(w_dev,' ');

    { *** write test string to encryption device *** }

write_sequential(wen_slr,addr(encryptin),8,wr_class,success);
show_err('WRITE ENCRYPTION SLOW ERROR',success);

    { *** read encrypted string *** }

read_sequential(ren_slr,addr(encryptout),size,rd_class,success);
show_err('READ ENCRYPTION SLOW ERROR',success);

putstr(w_dev,'ENCRYPTED STRING IS ');

for i:= 1 to 8 do
    if (ord(encryptout[i]) >= 40) and
        (ord(encryptout[i]) <= 176) then
        putchar(w_dev,encryptout[i]);

putln(w_dev,'');

    { *** write encrypted string to decryption device *** }

write_sequential(wde_slr,addr(encryptout),8,wr_class,success);
show_err('WRITE DECRYPTION SLOW ERROR',success);

    { *** read decrypted string *** }

read_sequential(rde_slr,addr(decryptout),size,rd_class,success);
show_err('READ DECRYPTION SLOW ERROR',success);

putstr(w_dev,'DECRYPTED STRING IS ');

for i:= 1 to 8 do

```

```

        putchar(w_dev,decryptout[i]);

putln(w_dev,' ');
putln(w_dev,'ENCRYPTION TEST COMPLETE');

        { *** detach encryption/decryption devices *** }
det_crypto(proc_suc);

crypto_tst_suc:= true;

end; { crypto_tst }

```

```

{*****}

```

Procedure: term_proc_create

Purpose: This procedure creates a single level child process for an untrusted tds using the parameters specified by the system manager. The child process code segment is a single level utility program which attaches the child process at the desired physical port. Four segments are passed to the child process. The stack segment contains the ch_init:rl_process_def record. The two common message buffer segments (inbuf and outbuf) are used to pass messages between the parent and child processes. Finally a code segment is required for all child processes. Process local segment numbers as well as pointers to the message buffer segments are passed back to the main procedure when the child process has been created.

```

*****

```

```

procedure term_proc_create(init: rl_process_def;
                           ch_parm: sysmgr_rec;
                           chld_num: integer;
                           var stk_slot:ch_array;
                           var outbuf_slot: ch_array;

```

```

var inbuf_slot: ch_array;
var out_ptr: pointer;
var in_ptr: pointer;
term_create_suc: boolean);

```

```

var

```

```

    DEBUG1 : INTEGER;
    DEBUG2 : INTEGER;

```

```

    ch_cde_seg_num: ch_array;
    chld_entry: integer;
    ch_init: rl_process_def;
    ch_addr_rec: rl_addr_array;
    ch_reg_rec: rl_reg_record;
    ch_res_rec: rl_res_record;
    ch_seg_list: seg_array;
    stk_init_ptr:                rl_process_def;
    stk_ptr: var_pointer;
    inbuf_ptr,outbuf_ptr: array [1..num_term] of pointer;
    seg_mgr_bytes: integer;
    stack_size,chld_size,buf_size: integer;
    stk_evc_val: ch_array;
    size,success: integer;
    i,j: integer;
    class:access_class;

```

```

begin { term_proc_create }

```

```

    term_create_suc: = false;

```

```

        { *** initialize child parameters *** }

```

```

    chld_size: = ch_parm.ch_size;

```

```

    buf_size: = ch_parm.b_size;

```

```

    j: = chld_num;

```

```

    chld_entry: = ch_parm.chld_ent[j];

```

```

{ .....

```

create, makeknown, and swapin child segments

*****}

{ makeknown terminal utility code segment located at child
entry number specified in sysmgr.ssb file }

seg_makeknown(init.initial_seg[root_offset],chld_entry,
 ch_cde_seg_num[j],r_e,size,class,success);
show_err('MAKEKNOWN CHILD ENTRY ERROR',success);

{ Determine required size for stack. It must be large
enough required information for child initialization.}

seg_mgr_bytes = sizeof(stack_header) + sizeof(kst_header)
 + (sizeof(kst_entry)*init.num_kst);

stack_size = r1_stack_size + vect_size + seg_mgr_bytes - 1;

{ *** create, makeknown, and swapin child stack segment *** }

seg_create(ch_cde_seg_num[j],0,stack_size,success);
show_err('CREATE CHILD STACK ERROR',success);

seg_makeknown(ch_cde_seg_num[j],0,stk_slot[j],r_w,size,
 class,success);
show_err('MAKEKNOWN CHILD STACK ERROR',success);

swapin_segment(stk_slot[j],success);
show_err('SWAPIN CHILD STACK ERROR',success);

{ stack eventcount is used to notify sys mgr that the
terminal process is activated. It is also used as an
entry in the ch_init record }

read_evc(stk_slot[j],stk_evc_val[j],success);
show_err('READ STACK EVC ERROR',success);
stk_evc_val[j] := stk_evc_val[j] + 1;

{ *** create message buffers *** }

{ outgoing message buffer }

```
seg_create(ch_cde_seg_num[j],1,buf_size,success);
show_err('CREATE OUTBUF ERROR',success);

seg_makeknown(ch_cde_seg_num[j],1,outbuf_slot[j],
              r_w,size,class,success);
show_err('MAKEKNOWN OUTBUF ERROR',success);

swapin_segment(outbuf_slot[j],success);
show_err('SWAPIN OUTBUF ERROR',success);
```

{ incoming message buffer }

```
seg_create(ch_cde_seg_num[j],2,buf_size,success);
show_err('CREATE INBUF ERROR',success);

seg_makeknown(ch_cde_seg_num[j],2,inbuf_slot[j],
              r_w,size,class,success);
show_err('MAKEKNOWN INBUF ERROR',success);

swapin_segment(inbuf_slot[j],success);
show_err('SWAPIN INBUF ERROR',success);
```

{ *** fill in ch_seg_list *** }

{ ch_seg_list determines order in which segments are passed
to the child process }

```
ch_seg_list[stack_offset] = stk_slot[j];
ch_seg_list[code_offset] = ch_cde_seg_num[j];
ch_seg_list[root_offset] = init.initial_seg[root_offset];
ch_seg_list[outbuf_offset] = outbuf_slot[j];
ch_seg_list[inbuf_offset] = inbuf_slot[j];
```

{ *** fill in child init record *** }

{ ch_init record is placed on stack for use by child
process when created }

```
ch_init.cpu = init.cpu;
```

```

ch_init.num_kst: = init.num_kst;
ch_init.root_access: = init.root_access;
ch_init.s_seg: = stack_offset;
ch_init.s_seg_event: = stk_evc_val[j]-1;

ch_init.resources.priority: = init.resources.priority-10;
lib_integer_to_b24(chld_size,ch_init.resources.memory);
ch_init.resources.processes: = 2;
ch_init.resources.segments: = 90;

```

{ min_class and max_class determine the access level of the child process. Since the terminal process is single level, they are the same. Levels are specified by the sysmgr during the parm_input initialization. }

```

ch_init.resources.min_class: = ch_parm.ch_access[j];
ch_init.resources.max_class: = ch_parm.ch_access[j];
ch_init.sp2: = 0;
ch_init.ring_num: = 1;

```

{ *** create stack pointer *** }

{ stack pointer is offset to start of r1_process_def }

```

stk_ptr.seg: = lib_mk_sel(ldt_table,stk_slot[j],1);
stk_ptr.off: = stack_size-(vect_size + seg_mgr_bytes
                                + sizeof(r1_process_def)) + 1;
stk_init_ptr: = stk_ptr.p;

```

{ copy ch_init on to stack }

```

move(ch_init,stk_init_ptr, sizeof(r1_process_def));

```

{ create pointers to message buffers }

{ point to start of message buffer, no offset }

```

outbuf_ptr[j]: = lib_mk_ptr(ldt_table,outbuf_slot[j],1);
inbuf_ptr[j]: = lib_mk_ptr(ldt_table,inbuf_slot[j],1);

```

```

    { fillin remaining records for create_process call }

    { child address record }

    { a maximum of 5 segments may be passed in ch_addr_array }
for i:= 0 to 4 do begin

    ch_addr_rec[i].segment_number:= ch_seg_list[i];

    { code segment must be of type read_execute }
    { others are type read_write }
    if i = 1 then begin

        ch_addr_rec[i].segment_type:= r_e;

    end else begin

        ch_addr_rec[i].segment_type:= r_w;

    end; {if}

    { swapin allsegments except root_offset }

    if i = 2 then begin

        ch_addr_rec[i].swapin:= false;

    end else begin

        ch_addr_rec[i].swapin:= true;

    end; {if}

    ch_addr_rec[i].protection:= 1;

end; {for}

    { *** child register record *** }

ch_reg_rec.ip:= 580;
ch_reg_rec.sp:= stk_ptr.off;
ch_reg_rec.sp1:= stack_size-(vect_size+ seg_mgr_bytes)+ 1;
ch_reg_rec.sp2:= 0;
ch_reg_rec.vec_seg:= 0;

```

```

ch_reg_rec.vec_off:= stack_size-vect_size+1;

        { *** child resource record *** }

        { child 1 is located at ch_res_rec.chld_num= 0 }
ch_res_rec.child_num:= chld_num-1;
ch_res_rec.priority:= ch_init.resources.priority;
ch_res_rec.memory:= ch_init.resources.memory;
ch_res_rec.processes:= ch_init.resources.processes;
ch_res_rec.segments:= ch_init.resources.segments;
ch_res_rec.min_class:= ch_parm.ch_access[chld_num];
ch_res_rec.max_class:= ch_parm.ch_access[chld_num];

in_ptr:= inbuf_ptr[j];
out_ptr:= outbuf_ptr[j];

putln(w_dev,'CREATING CHILD PROCESS');

create_process(ch_addr_rec,ch_reg_rec,ch_res_rec,success);
show_err('CREATE CHILD PROCESS ERROR',success);

        { wait for child process to advance stack eventcount
          indicating that child process is active }

{ **** }
read_evc(stk_slot[j],DEBUG1,success);
read_evc(ch_cde_seg_num[j],DEBUG2,success);
putstr(w_dev, 'AWAITING STACK VALUE ');
putdec(w_dev,stk_evc_val[j]);
putstr(w_dev, 'CURRENT STACK VALUE ');          *****
putdec(w_dev,DEBUG1);                          USED FOR DEBUGGING
putln(w_dev,'');                               *****
putstr(w_dev, 'CURRENT CODE VALUE ');
putdec(w_dev,DEBUG2);
putln(w_dev,'');
await(stk_slot[j],stk_evc_val[j],success);
show_err('AWAIT STACK ADVANCE ERROR',success);
read_evc(ch_cde_seg_num[j],DEBUG1,success);

```

```

putln(w_dev,'CODE VALUE AFTER AWAIT ');
putdec(w_dev,DEBUG1);
putln(w_dev,'');
        DEBUG1 := outbuf_ptr[j]          ;
        show_err('CHILD ATTACH FAILED ', DEBUG1);
*****}

putln(wdev,'CHILD PROCESS CREATED');

term_create_suc:= true;

end; { term_proc_create }

```

```

{ *****

```

Procedure: xmit_rec

Purpose: This procedure takes the message stored in the outgoing buffer of the source terminal, encrypts each block, and transmits it sequentially via the appropriate external communications port. The cryptographic is provided by the sysmgr_rec. The fb_key is initially input by the system manager. (Then the encrypted block becomes the fb_key for the next block.) At the receiver the message is decrypted and stored in the incoming message buffer of the destination terminal. Access levels of the msg and dest are compared. If they do not match the msg is not delivered, and an error msg is returned to the source.

```

*****}

```

```

procedure xmit_recv( ch_parm: sysmgr_rec;
                    orig_term: integer;
                    dest_term: integer;
                    o_out_ptr: pointer;
                    o_in_ptr: pointer;

```

```

d_out_ptr: pointer;
d_in_ptr: pointer;
var int_mess_num:integer;
var recv_suc: boolean);

```

```

var

```

```

out_rec,in_rec: buf_rec;
in_ptr,out_ptr: array [1..num_term] of pointer;
time: cc_array;
fb_key: buf8;
srce,dest: char;
int_dest: integer;
str_mess_num:string;
encryptout,decryptin,decryptout: array [1..8] of char;
i,j: integer;
size: integer;
recv_class,xmit_class,class: access_class;
count: integer;
success: integer;
dest_comp,mess_comp: integer;
proc_suc: boolean;
temp_char: char;

```

```

begin { xmit_recv }

```

```

recv_suc:= false;

```

```

putln(w_dev,'ENTERING TRANSMIT/RECEIVE MODULE');

```

```

sys_config(orig_term,dest_term,proc_suc);

```

```

{ *** retrieve message stored in originator's outgoing msg buf ***}

```

```

move(o_out_ptr                                ,out_rec,sizeof(out_rec));

```

```

{ *** fill in remaining address block entries *** }

```

```

{ outgoing message number }

```

```

out_rec.num:= int_mess_num;

        { message classification }

out_rec.block[1][6]:= chr(ch_parm.ch_access[orig_term].
                        compromise[0] + 48);

        { insert message number in address block }

binascii(int_mess_num,4,str_mess_num,'0');
for i:= 1 to 3 do
    out_rec.block[1][i + 2]:= str_mess_num[i];

        { increment message number counter }

int_mess_num:= int_mess_num + 1;

putln(w_dev, 'ENTER THE 8 CHARACTER ENCRYPTION KEY. ');

for i:= 1 to 8 do begin

    getchar(r_dev,temp_char);
    fb_key[i]:= ord(temp_char)-48;

end; {for}

putln(w_dev, 'ENCRYPTION KEY INSERTED');

        { transmitter access class }

xmit_class:= ch_parm.ch_access[orig_term];

{ ****

        begin transmit/receive loop

        **** }

for i:= 1 to out_rec.num_blk do begin

    { in cbc mode crypto devices must be reattached to transmit

```

each block. this is required because the previous encrypted block is used as the fb_key to encrypt the next block.}

```
att_crypto(ch_parm.key,fb_key,proc_suc);
```

```
    { write to encryption device }
```

```
write_sequential(wen_slst,addr(out_rec.block[i]),8,xmit_class,succes);
```

```
show_err('WRITE ENCRYPTION SLOW ERROR',succes);
```

```
    { read encrypted text }
```

```
read_sequential(ren_slst,addr(encryptout),size,class,succes);
```

```
show_err('READ ENCRYPTION SLOW ERROR',succes);
```

```
    { transmit encrypted block }
```

```
write_sequential(xmit_slst,addr(encryptout),8,xmit_class,succes);
```

```
show_err('TRANSMIT ERROR',succes);
```

```
    { determine fb_key for next block }
```

```
for j:= 1 to 8 do
```

```
    fb_key[j]:= encryptout[j];
```

```
    putln(w_dev,' ');
```

```
{ ***** }
```

```
begin receiving message
```

```
***** }
```

```
    { receiver access class }
```

```
recv_class:= ch_parm.ch_access[dest_term];
```

```
    { read encrypted text }
```

```
read_sequential(recv_slst,addr(decryptin),size,class,succes);
```

```
show_err('RECEIVE ERROR',succes);
```

```
putln(w_dev,'RECEIVED MESSAGE IS');
```

```
for j:= 1 to 8 do
```

```

        putchar(w_dev,decryptin[j]);
        putln(w_dev,' ');

        { write to decryption device }

write_sequential(wde_slst,addr(decryptin),8,recv_class,success);
show_err('WRITE DECRYPTION SLOW ERROR',success);

        { read decrypted text }

read_sequential(rde_slst,addr(decryptout),size,class,success);
show_err('READ DECRYPTION SLOW ERROR',success);

putln(w_dev,'DECRYPTED MESSAGE IS');

for j:= 1 to 8 do begin
    in_rec.block[i][j]:= decryptout[j];
    if (ord(decryptout[j]) >= 40) and
        (ord(decryptout[j]) <= 176) then
        putchar(w_dev,decryptout[j]);

end;

putln(w_dev,' ');

        { count is number of blocks in received message }

count:= count + 1;

        { detach crypto devices to prepare for next block }

det_crypto(proc_suc);

end; {for}

detach(xmit_slst);
detach(recv_slst);

{ **** }

```

message transmitted and received

*****}

{ insert number of blocks into incoming record }

in_rec.num_blk:= count;

{ decode address block }

srce:= in_rec.block[1][1];

dest:= in_rec.block[1][2];

putstr(w_dev,'DESTINATION IS');

putchar(w_dev,dest);

putln(w_dev,' ');

int_dest:= ord(dest)-48;

putstr(w_dev,'INT_DEST IS');

putdec(w_dev,int_dest);

putln(w_dev,' ');

dest_comp:= ch_parm.ch_access[int_dest].compromise[0];

mess_comp:= ord(in_rec.block[1][6])-48;

putln(w_dev,' ');

putstr(w_dev,'DEST_COMP-MESS_COMP');

putdec(w_dev,dest_comp);

putdec(w_dev,mess_comp);

putln(w_dev,' ');

{ compare message and destination access levels for
possible security violation }

if mess_comp < > dest_comp then begin

{ if srce= '0' then incoming message is an error
message concerning a security violation }

if srce < > '0' then begin

putln(w_dev,'SECURITY VIOLATION MESSAGE NUMBER');

for i:= 3 to 5 do

```

        putchar(w_dev,in_rec.block(1[i]));
        recv_suc: = false;

        { prepare error msg for transmission }

err_msg(srce,d_out_ptr,proc_suc);

end else begin

        { if incoming traffic is an error msg then
        move it to the incomming message buffer of
          the destination terminal }

        move(in_rec,d_in_ptr ,sizeof(in_rec));

        { reset recv_suc }

        recv_suc: = true;

    end;

end else begin

        { if no violation, move msg into incoming msg
        buffer of destination terminal }

        move(in_rec,d_in_ptr ,sizeof(in_rec));
        recv_suc: = true;

end; {if}

end; {xmit_rec}

```

{

Procedure: err_msg

Purpose: In the event of a security violation, this procedure fills destination outgoing buffer with an error message. This error message is then transmuted to the source for display at the originator's terminal.

Purpose: This procedure detachs all data encryption/decryption devices.

.....}

procedure det_crypto(var proc_suc:boolean);

begin {det_crypto}

detach(wen_slk);

detach(ren_slk);

detach(wde_slk);

detach(rde_slk);

end; {det_crypto}

{.....}

Procedure: show_err

Purpose: This procedure is called to display the success code of the resource management call if it is other than zero. If the success code indicates no_error then no message is output.

.....}

procedure show_err(str: string;

code: integer);

begin {show_err}

if code < > no_error then begin

putstr(w_dev,str);

putstr(w_dev,' ');

putdec(w_dev,code);

putln(w_dev,' ');

```

        end;

end; { show_err }

```

```

{ *****

```

Procedure: main

Purpose: This procedure initializes system operation. It performs comm and crypto checks and then creates a single level process for the remote terminal. Once the system is on-line, it controls access to the external communications ports. Messages are transmitted and received, and security checks are performed on all incoming traffic.

```

*****}

```

```

procedure main( var init : rl_process_def);

```

```

var

```

```

{   DEBUG : INTEGER;          *** USED FOR DEBUGGING ***)

```

```

    i: integer;
    stk_slc,bufout_slc,bufin_slc: ch_array;
    bufout_ptr,bufin_ptr: array [1..num_term]
                                of pointer;
    bufout_evc,bufin_evc: ch_array;
    mgr_rec: sysmgr_rec;
    test_key,test_fbkey: buf8;
    mess_dest,mess_srce: integer;
    templ_port,temp2_port: integer;
    success: integer;
    ch_num: integer;
    proc_suc: boolean;
    recv_suc: boolean;

```

```

    mess_num:integer;
    temp_class : access_class;

begin {main}

if (init.cpu < > 0) then begin

    self_delete(init.initial_seg[stack_offset], success);
    await(init.initial_seg[stack_offset], 32000, success);

end; {if}

attach(0,w_dev,false,success);
if (success < > no_error) then begin

    self_delete(init.initial_seg[stack_offset], success);
    await(init.initial_seg[stack_offset], 32000, success);

end; {if}

attach(0,r_dev,true,success);
if (success < > no_error) then begin

    show_err('ATTACH R_DEV FAILED ', success);
    self_delete(init.initial_seg[stack_offset], success);
    await(init.initial_seg[stack_offset], 32000, success);

end; {if}

putln(w_dev,'SYSTEM MANAGER TERMINAL ATTACHED');

    { call procedure to enter system parameters }

temp_class.compromise := init.resources.max_class.compromise;
temp_class.integrity := init.resources.min_class.integrity;
input_param(temp_class,mgr_rec,proc_suc);

    { xmit rcv ports for comm tst }

temp1_port:= mgr_rec.comm_port[1];
temp2_port:= mgr_rec.comm_port[2];

    { configure xmit rcv ports }

```

```

sys_config( temp1_port,temp2_port,proc_suc);

        { test comm channel pass 1 }

comm_tst(init,temp1_port,temp2_port,proc_suc);

        { reconfigure xmit/recv ports to transmit in opposite dir }

sys_config(temp2_port,temp1_port,proc_suc);

        { test comm channel pass 2 }

comm_tst(init,temp2_port,temp1_port,proc_suc);

        { attach crypto devices in CBC mode }

att_crypto(mgr_rec.key,mgr_rec.key,proc_suc);

        { test crypto devices }

crypto_tst(init,proc_suc);

putln(w_dev,'SYSTEM INITIALIZATION COMPLETE');

        { loop to create child process for each remote terminal }

for i:= 1 to num_term do begin

    ch_num:= i;

        { create child process }

    term_proc_create(init,

                                mgr_rec,
                                ch_num,
                                stk_slc,
                                bufout_slc,
                                bufin_slc,
                                bufout_ptr[i],
                                bufin_ptr[i],
                                proc_suc);

```

```

putstr(w_dev,'CHILD PROCESS CREATED TERMINAL ');
putdec(w_dev,i);
putln(w_dev,' ');

        { initialize buffer event counts }

bufin_evc[i]:= 0;
bufout_evc[i]:= 0;

end; {for}

        {Initial message destination is terminal 2.}

mess_dest:= 2;
mess_num:= 0;

        { To start system advance inbuf_evc for terminal 1.}

advance(bufin_slc[1], success);
show_err('START SYSTEM INBUFFER ADVANCE ERROR', success);

        hhPP { initialize message receipt success value }

recv_suc:= true;

{ *****

        begin independent system operation loop

*****}

while true do begin

        { inner loop synchronizes terminal to terminal communications }

for i:= 1 to num_term do begin
        mess_srce:= i;
        mess_num:= mess_num + 1;

        { check for received message security violation }

if recv_suc = true then begin

        { if no error then wait for next outgoing message }

```

```
bufout_evc[i] := bufout_evc[i] + 1;
```

```
{ *****
```

USED FOR DEBUGGING

```
read_evc(bufout_slst[i],DEBUG,success);  
putstr(w_dev, 'IN LOOP AWAITING VALUE ');  
putdec(w_dev,bufout_evc[i]);  
putstr(w_dev, 'CURRENT VALUE ');  
putdec(w_dev,DEBUG);  
putln(w_dev,'');
```

```
*****}
```

```
await(bufout_slst[i],bufout_evc[i],success);  
show_err('AWAIT MESSAGE READY FOR TRANSMIT',success);  
putln(w_dev, 'MESSAGE READY FOR TRANSMISSION');
```

```
    { transmit and receive outgoing message }
```

```
xmit_rcv(mgr_rec,mess_srce,mess_dest,  
bufout_ptr[mess_srce],bufin_ptr[mess_srce],  
bufout_ptr[mess_dest],bufin_ptr[mess_dest],  
mess_num,rcv_suc);
```

```
putln(w_dev,'MESSAGE SENT');
```

```
    { notify message source that message was xmit }
```

```
advance(bufin_slst[mess_srce],success);  
show_err('ADVANCE SOURCE INBUF ERROR',success);
```

```
    { check for received message security violation }
```

```
if rcv_suc = true then begin
```

```
    { if no error then notify dest terminal  
      to display incoming message }  
    advance(bufin_slst[mess_dest],success);
```

```

        show_err('ADVANCE DEST INBUF ERROR',success);

        putln(w_dev,'MSG RECVD AND DELIVERED');

        { new dest term is message srce }

        mess_dest:= i;

end else begin

    { if security violation did occur then
    transmit error msg back to source. error
    msg has already been placed in outgoing
    buffer by procedure xmit_rcv. }

        xmit_rcv(mgr_rec,mess_dest,mess_srce,
        bufout_ptr[mess_dest],bufin_ptr[mess_dest],
        bufout_ptr[mess_srce],bufin_ptr[mess_srce],
        mess_num,proc_suc);

        putln(w_dev,'ERROR MSG TRANSMITTED');

        { notify source of incoming error message }

        advance(bufin_slc[mess_srce],success);
        show_err('NOTIFY SRCE OF ERROR ADVANCE',success);

end; {if}

end else begin

    { if received message had a security violation the
    loop will return control to the message source so
    that he can display the error message }

    { rcv_suc = true to allow display of error msg }

    rcv_suc:= true;

end; {if}

end; {for}

```

```
end; {while}
```

```
putln(w_dev,'PROGRAM COMPLETE');
```

```
while true do;
```

```
end; {main}
```

```
modend.
```

APPENDIX E

GUARD.KMD LINKING FILE

{*****}

Program: GUARD.KMD

Date: 7 February 1987

Author: G. E. RECTOR, JR., CAPT/USMC

Advisor: T. J. BROWN, MAJ/USAF

Purpose: This program is used when linking GUARD.
It eliminates the need to manually enter each of the file
names each time a new version or update of the program
is compiled.

*****}

guard = rl-init, guard, rllib/s, paslib/s/p:80

APPENDIX F

GUARD-CON.ZLI INCLUDE FILE OF CONSTANT DECLARATIONS

{*****}

Program: GUARD-CON.ZLI

Date: 7 FEBRUARY 1987

Author: G. E. RECTOR, JR., CAPT/USMC

Advisor: T. J. BROWN MAJ/USAF

Purpose: This file is developed as an include file and contains constant declarations used for the multilevel interface application program. This file must be included in the constant declarations for each module.

*****}

num_term = 2;
mess_buf_size = 4;

xmit_slit = 6;
recv_slit = 7;

wen_slit = 2;
ren_slit = 3;
wde_slit = 4;
rde_slit = 5;

stack_offset = 0;
code_offset = 1;
root_offset = 2;
pb_offset = 3;
outbuf_offset = 3;
inbuf_offset = 4;

{***** end of guard-con.zli *****}

APPENDIX G

GUARD-TYP.ZLI TYPE DECLARATION INCLUDE FILE

```

{*****
    Program:  GUARD-TYP.ZLI
    Date:     7 FEBRUARY 1987
    Author:   G. E. RECTOR, JR, CAPT/USMC
    Advisor:  T. J. BROWN, MAJ/USAF

    Purpose:  This file was developed as an include file of
type declarations to be used with each module of this application.
*****}

sysmgr_rec = record
    comm_port : array [1..2] of integer;
    b_size : integer;
    ch_size : integer;
    ch_access : array [1..num_term] of access_class;
    chld_ent : array [1..num_term] of integer;
    key : buf8;
end;

buf_rec = record
    num : integer;
    num_blk : integer;
    block : array [1..mess_buf_size] of array [1..8] of char;
end;

ch_array = array [1..num_term] of integer;
{***** end guard-typ.zli *****}

```

APPENDIX H

TDS1 TERMINAL UTILITY PROGRAM LISTING (JANUS ADA)

The TDS1 terminal utility package is compiled and prepared for execution in almost the same manner as the system manager module. By modifying certain parameters which are identified in the program listing, the system manager may specify the specific port and terminal number of each remote terminal process. TDS.CMD files are created for each terminal attached and are essentially identical except that TDS1 is written in Janus ADA and TDS2 is written in PASCAL MT+. Each is compiled, linked, and sysgened separately, but perform identically. A TDS1.CMD file is submitted with the GUARD.SSB file for each remote terminal attached. To attach additional terminals, (as more ports become available), entry numbers would have to be specified in the GUARD.SSB file. Janus ADA is compiled and linked in a manner very similar to PASCAL MT+. A library file was created providing the ring 0 and ring 1 calls similar to those developed by Gemini for PASCAL MT+. A listing of the library file DEFS.lib and the package body DEFS.PKG to support that library are listed as Appendices I and J.

WITH agate, agatej, arl, alib, alibj, strlib, util, defs;

PACKAGE BODY TDSI IS

USE agate, agatej, arl, alib, alibj, strlib, util, defs;

```
-- *****
--
--
-- Program: TDSI
--
-- Date: 11 FEBRUARY 1987
--
-- Author: G. E. RECTOR CAPT USMC
--
-- Advisor: MAJ. T. J. BROWN MAJ USAF
--
-- Purpose: This program is initiated when the system
-- manager creates the single level TDS process. It allows
-- the TDS operator to enter and send messages via the system
-- manager process, as well as display incoming messages.
-- Message specifications and terminal access level are deter-
-- mined by the system manager process and passed to the TDS
-- process in its rl_process_def record. Other system constants
-- are provided in the guard-tyt.zli and guard-con.zli files.
-- Incoming and outgoing buffers are used to store messages.
-- Eventcounts for these segments are used to synchronize system
-- communications.
--
-- *****

-- Constants used by this program :

t_phys_dev : CONSTANT Integer := 100;
pc_dev     : CONSTANT Integer := 101;
term_num   : Constant Character := '1';

-- Variables used by this program :
```

```

Debug : Integer;
Debugptr : Pointer;
Success : Integer;
Seg_num : Integer;
Mode : Character;
Xmit_buf_stat, Rec_buf_stat : Boolean;
Temp_str : String(1);
i, Level : Integer;
Inbuf_evc : Integer;
Stk_evc : Integer;
Sys_start : Boolean;

```

Begin

```

    -- initialize terminal process parameters
Xmit_buf_stat := False;
Mode := "0";
    -- sys_start = false for twterminal 1 only all
    -- other terminals should have sys_start = true
Sys_start := (Term_num = "1");
Inbuf_evc := 0; -- attach terminal as read/write device
Attach(T_phys_dev, W_dev, False, Success);
If (Success = No_error) Then
    Debug := Success;
    Swapin_segment(Init.Initial_seg(Outbuf_offset), Success);
    Debugptr := Lib_mk_ptr(Ldt_table, Init.Initial_seg(Outbuf_offset), 1);
    Debugptr := Debug;
End If;
Attach(T_phys_dev, R_dev, True, Success);
If (Success = No_error) Then
    Show_err("ATTACH R_DEV FAILED ", Success);
End If;
Putln(W_dev, " ");
Putln(W_dev, "TERMINAL ACTIVE");
Puchar(W_dev, Term_num);
Putln(W_dev, " ");

```

```

-- stack eventcount is advanced to notify
-- sysmgr that terminal is activated
Advance(Init.Initial_seg(Code_offset),Success);
Advance(Init.Initial_seg(Stack_offset),Success);
Show_err("STACK ADVANCE ERROR",Success);
-- loop until operator enters 'e' to indicate logoff
While Mode = "e" Loop
    -- inbuf_evc is used to have the terminal wait after
    -- transmitting a message until a reply is received
    -- from the dest term. It is initially advanced for
    -- terminal 1 to start the system and then is advanced
    -- upon receipt of an incoming message
    Inbuf_evc := Inbuf_evc + 1;
    Await(Init.Initial_seg(Inbuf_offset),Inbuf_evc,Success);
    Show_err("AWAIT INCOMING MESSAGE",Success);
    -- sys_start is used to avoid the 'display incoming
    -- message' prompt at terminal 1 when the system is
    -- started. Once the system is operating it will
    -- always be true
    If Sys_start Then
        Rec_buf_stat := True;
        Putln(W_dev,"DISPLAY INCOMING MESSAGE");
    End If;
    Sys_start := True;
    -- inner loop is used to indicate that a
    -- message has been sent and alert the operator
    -- that the terminal is waiting for a reply
    While (Mode = "X") Loop
        -- help menu consists of a display of term
        -- access level, and a display of possible
        -- modes
        Putstr(W_dev,"TERMINAL COMPROMISE LEVEL");
        Level := Init.Root_access.Compromise(1);
        Case Level Is
            When 0 =>

```

```

        Putln(W_dev,"UNCLASSIFIED");
    When 2 = >
        Putln(W_dev,"CONFIDENTIAL");
    When 4 = >
        Putln(W_dev,"SECRET");
    When 6 = >
        Putln(W_dev,"TOP SECRET");
End Case;
Putln(W_dev,"ENTER MODE DESIRED");
Putln(W_dev,"I = INPUT MESSAGE");
Putln(W_dev,"D = DISPLAY RECEIVED MESSAGE");
Putln(W_dev,"X = TRANSMIT MESSAGE");
Putln(W_dev,"E = LOGOFF");
Putln(W_dev,"");
Putstr(W_dev,"ENTER MODE HERE");
Getchar(R_dev,Mode);
If (Mode >= "a") And (Mode <= "z") Then
    Mode := Character Val(Type_name Pos(Mode)
        - Type_name Pos("a") + Type_name Pos("A"));
End If;
If Mode = "I" Then
    If Xmit_buf_stat = False Then
        -- enter message to be stored in
        -- outgoing message buffer
        Input_mess(Init.Resources.Min_class,
            Init.Initial_seg(Outbuf_offset),Xmit_buf_stat);
    Else
        Putln(W_dev,"MESSAGE WAITING TO BE TRANSMITTED");
    End If;
Elsif Mode = "D" Then
    If Rec_buf_stat = True Then
        Putln(W_dev,"ENTERING DISPLAY MODULE");
        -- display contents of incoming
        -- message buffer
        Disp_mess(Init.Initial_seg(Inbuf_offset),Rec_buf_stat);
    End If;

```

```

Else
    Putln(W_dev,"INCOMING BUFFER EMPTY");
End If;
Elsif Mode = "X" Then
    If Xmit_buf_stat = True Then
        Putln(W_dev,"SENDING MESSAGE");
        Xmit_mess(Init.Initial_seg(Inbuf_offset),
            Init.Initial_seg(Outbuf_offset),Inbuf_evc,Xmit_buf_stat);
    Else
        Putln(W_dev,"OUTGOING BUFFER EMPTY");
        Mode := "0";
    End If;
Elsif Mode = "E" Then
    Putln(W_dev,"LOGOFF PROCESS INITIATED");
    Logout(Init);
Else
    Putln(W_dev,"MODE ENTRY ERROR, TRY AGAIN");
End If; -- end of inner loop-exit after msg xmit
End Loop;
Putln(W_dev,"WAITING FOR INCOMING MESSAGE");
    -- reset mode selection value
Mode := "0";
End Loop;
Putln(W_dev,"END OF TDS TERMINAL PROCESS");
Detach(W_dev);
Detach(R_dev); -- infinite loop to avoid crash
While True Loop
End Loop;

End TDS1;

```

APPENDIX I

DEFS.LIB LIBRARY DEFINITION FILE

```

WITH agate, agatej, arl, alib, alibj, strlib, util;
PACKAGE DEFS IS
USE agate, agatej, arl, alib, alibj, strlib, util;

num_term : CONSTANT Integer := 2;
mess_buf_size : CONSTANT Integer := 4;
r_dev : CONSTANT Integer := 0;
w_dev : CONSTANT Integer := 1;
t_phys_dev : CONSTANT := 100;
pc_dev : CONSTANT := 101;
xmit_slit : CONSTANT Integer := 6;
recv_slit : CONSTANT Integer := 7;

wen_slit : CONSTANT Integer := 2;
ren_slit : CONSTANT Integer := 3;
wde_slit : CONSTANT Integer := 4;
rde_slit : CONSTANT Integer := 5;

stack_offset : CONSTANT Integer := 0;
code_offset : CONSTANT Integer := 1;
root_offset : CONSTANT Integer := 2;
pb_offset : CONSTANT Integer := 3;
outbuf_offset : CONSTANT Integer := 3;
inbuf_offset : CONSTANT Integer := 4;

TYPE sysmgr_rec IS record
    comm_port : array (1..2) of integer;
    b_size : integer;
    ch_size : integer;
    ch_access : array (1..num_term) of access_class;
    chd_ent : array (1..num_term) of integer;
    key : buf8;
END RECORD;

TYPE Table IS ARRAY (1..8) of CHARACTER;

TYPE buf_rec IS record
    num : integer;
    num_blk : integer;
    block : array (1..mess_buf_size) of Table;
END RECORD;

Procedure Jamps_setup;
Procedure Jamps_unhook;

PROCEDURE put_in (ldev : in integer; w_class : in access_class;
    str : in string );

PROCEDURE put_str (ldev : in integer; w_class : in access_class;
    str : in string );

PROCEDURE put_decd (ldev : in integer; w_class : in access_class;
    dval : in integer );

PROCEDURE put_succ (in_str : in string; decd_val : in integer;
    w_class : in access_class );

PROCEDURE get_in (ldev : in integer; r_class : out access_class;

```

```

        str : out string );

Procedure Input_mess (class : in access_class;
                     comm_buf : in Integer;
                     buf_stat : out boolean);

Procedure Xmit_mess (inbuf_slot : in Integer;
                    outbuf_slot : in Integer;
                    inbuf_evt : in Integer;
                    xmit_buf_stat : out Boolean);

Procedure Disp_mess (comm_buf : in Integer;
                    rec_buf_stat : out Boolean);

Procedure Logoff ( init : in rl_process_def);

Procedure Clr_screen (proc_suc : out Boolean);

END DEFS;

```

APPENDIX J

DEFS.PKG PACKAGE BODY FILE USED TO SUPPORT DEFS.LIB

```
pragma rangecheck(off); pragma debug(off); pragma arithcheck(off);
pragma enumtab(off);
```

```
WITH agate, agatej, ar1, alib, alibj, strlib, util;
PACKAGE BODY defs IS
USE agate, agatej, ar1, alib, alibj, strlib, util;
```

```
-- *****
```

Procedure Jamps_setup Is

```
Success : Integer;
Att_rec : attach_struct;
```

Begin

```
Att_rec.dev_name := sior;
Att_rec.sior_rec.dev_num := Pc_dev;
Att_rec.sior_rec.dev_type := lo;
Att_rec.sior_rec.dev_id := r_dev;
Att_rec.sior_rec.nr1 := Bvt( 16#04D# );
Att_rec.sior_rec.nr2 := Byte( 16#035# );
Att_rec.sior_rec.io_mode := rts_oflow;
Att_rec.sior_rec.maximum := 256;
Att_rec.sior_rec.delim_active := false;

Loop
    Detach_device(R_dev,Success);
    Put_succ("READ DEVICE DETACH ERROR ",Success,W_class);
    Exit When (Success = No_error);
End Loop; -- Repeat Loop
Loop
    Attach_device(Att_rec,Success);
    Put_succ("PC DEVICE ATTACH ERROR ",Success,W_class);
    Exit When (Success = No_error);
End Loop; -- Repeat Loop
Putln(W_dev,W_class,"PC DEVICE ATTACHED ");
End Jamps_setup;
```

```
-- *****
```

Procedure Jamps_unhook Is

```
Success : Integer;
Att_rec : attach_struct;
```

Begin

```
Att_rec.dev_name := sior;
Att_rec.sior_rec.dev_num := T_phys_dev;
Att_rec.sior_rec.dev_type := lo;
Att_rec.sior_rec.dev_id := R_dev;
Att_rec.sior_rec.nr1 := Byte( 16#04D# );
Att_rec.sior_rec.nr2 := Byte( 16#03E# );
Att_rec.sior_rec.io_mode := rts_oflow;
Att_rec.sior_rec.maximum := 256;
Att_rec.sior_rec.delim_active := false;

Loop
    Detach_device(R_dev,Success);
    Put_succ("PC DEVICE DETACH ERROR ",Success,W_class);
    Exit When (Success = No_error);
End Loop; -- Repeat Loop
```

```

    Loop
        Attach_device(Att_rec,Success);
        Put_suc("READ DEVICE ATTACH ERROR ",Success,W_class);
        Exit When (Success = No_error);
    End Loop; -- Repeat Loop
    Putln(W_dev,W_class,"READ DEVICE ATTACHED ");

End Jumps_unhook;

-- *****

PROCEDURE put_ln ( ldev : in integer; w_class : in access_class;
                  str : in string ) IS

-- put a string on device ldev with cr and lf

    out_buf : string( 82 );
    success : integer;
    wt_sio : wt_seq_struct;
    size_str : integer;
    CR : CONSTANT integer := 13;
    LF : CONSTANT integer := 10;
    BS : CONSTANT integer := 08;

BEGIN
    out_buf := str;
    size_str := length( str );
    out_buf := out_buf & char_to_str( character_val( CR ));
    out_buf := out_buf & char_to_str( character_val( LF ));
    wt_sio.device := ldev;
    wt_sio.data_off := out_buf ADDRESS + 1;
    wt_sio.data_seg := get_ss();
    wt_sio.count := size_str + 2;
    wt_sio.class := w_class;
    write_sequential( wt_sio, success );
END put_ln;

-- *****

PROCEDURE put_str ( ldev : in integer; w_class : in access_class;
                  str : in string ) IS

-- put a string on device ldev.

    out_buf : string;
    success : integer;
    wt_sio : wt_seq_struct;
    size_str : integer;

BEGIN
    out_buf := str;
    size_str := length( str );
    wt_sio.device := ldev;
    wt_sio.data_off := out_buf ADDRESS + 1;
    wt_sio.data_seg := get_ss();
    wt_sio.count := size_str;
    wt_sio.class := w_class;
    write_sequential( wt_sio, success );
END put_str;

-- *****

PROCEDURE put_dec( ldev : in integer; w_class : in access_class;
                  dval : in integer ) IS

-- put the string equivalent of a integer on the terminal screen.

    out_buf : string( 10 );

```

```

BEGIN
    out_buf := Int_to_str( dval );
    put_str( ldev, w_class, out_buf );
END put_dec;

-- *****

PROCEDURE put_succ( in_str : in string; dec_val : in integer;
                    w_class : in access_class ) IS

-- print a string and an integer on device attached in slot STDIO_W
-- (should be a serial terminal).

BEGIN
    put_str( STDIO_W, w_class, in_str );
    put_dec( STDIO_W, w_class, dec_val );
    put_ln( STDIO_W, w_class, "" );
END put_succ;

-- *****

PROCEDURE get_ln( ldev : in integer; r_class : out access_class;
                 str : out string ) IS

-- Assumes ldev attached for single character reads.
-- Reads a string one character at a time.
-- Automatically echoes out STDIO_W.
-- Handles backspaces. Terminates on CR or after 80 input chars.
-- Ignores control chars other than BS and CR.

CR : CONSTANT integer := 13;
LF : CONSTANT integer := 10;
BS : CONSTANT integer := 08;

?in_buf : string( 80 );
success : integer;
rd_sio : rd_seq_struct;
rd_ret : rd_seq_return;
str_size : integer;
in_char : integer;
temps : string( 3 );

BEGIN
    str_size := 0;
    in_char := 0;
    rd_sio.data_off := in_char ADDRESS;
    rd_sio.device := ldev;
    rd_sio.data_seg := get_ss();
    LOOP
        read_sequential( rd_sio, rd_ret, success );
        IF ( success = no_error ) THEN
            in_char := Land( in_char, 16#7F# ); -- turn-off high bit.
            IF ( in_char = BS ) THEN
                IF ( str_size > 0 ) THEN
                    str_size := str_size - 1;
                    temps( 0 ) := character_val( 3 );
                    temps( 1 ) := character_val( BS );
                    temps( 2 ) := character_val( BS );
                    temps( 3 ) := character_val( BS );
                    put_str( STDIO_W, rd_ret.class, temps );
                END IF;
            ELSIF ( in_char = CR ) THEN
                temps( 0 ) := character_val( 2 );
                temps( 1 ) := character_val( CR );
                temps( 2 ) := character_val( LF );
                put_str( STDIO_W, rd_ret.class, temps );
            ELSIF ( character_val( in_char ) IN " " ) THEN
                str_size := str_size + 1;
                in_buf( str_size ) := character_val( in_char );
            END IF;
        END IF;
    END LOOP;

```

```

                                temps( 0 ) := character'val( 1 );
                                temps( 1 ) := character'val( in_char );
                                put_str( STDIO_W, rd_ret.class, temps );
                                END IF;
                                END IF;
                                EXIT WHEN (( str_size >= 80 ) OR ( in_char = CR ) OR
                                ( success /= no_error ));
                                END LOOP;

                                in_buf( 0 ) := character'val( str_size );
                                str := in_buf;
                                r_class := rd_ret.class;
                                END get_ln;

-- *****
--
-- Procedure: input_mess
--
-- Purpose: This procedure allows the operator
-- to input a message into the outgoing message buffer.
-- Characters are input into 8 byte blocks so that they
-- will be compatible with the data encryption device.
-- The character 'S' is used to indicate the end of the
-- message. The initial block is reserved for the
-- address header. Format of the header is as follows:
--
-- blk[1][1]: source
-- blk[1][2]: destination
-- blk[1][3-5]: message number
-- blk[1][6]: classification
-- blk[1][7-8]: number of blocks in msg
--
-- Source, destination, and number of blocks are entered
-- in this procedure. Remaining entries are filled in
-- by the sysmgr process prior to transmission.
-- *****
Procedure Input_mess (class : in access_class;
                                comm_buf : in Integer;
                                buf_stat : out boolean) Is

    Mess_rec : Buf_rec;
    Blk_cnt : String;
    Temp_ptr, Input_ptr : Pointer;
    Charin : Character;
    i,j,k,Success : Integer;
    Count : Integer;
    Proc_suc : Boolean;

Begin
    -- *****input mess*****
    -- create pointer to start of input buffer
    Input_ptr := Lib_mk_ptr(Ldt_table,Comm_buf,1);
    Temp_ptr := Input_ptr;
    Clr_screen(Proc_suc);
    Putln(W_dev,W_class,"BEGIN JINTACCS AUTOMATED
                                MESSAGE PREPARATION PROCESS.");
    Putln(W_dev,W_class,""); -- initialize msg block counter
    i := 1;
    Charin := Character'Val(0);
    Jamps_setup(Class); -- begin character entry loop
    While (Charin /= Character'Val(12)) And (i /= Mess_buf_size) Loop
-- block 1 is addr, block 2 is strt of msg
        i := i + 1; -- begin loop to read 8 char for each block
        For j In 1..8 Loop
            If (Charin /= Character'Val(12)) Then
                Getchar(R_dev,R_class,Charin);

```

```

Mess_rec.Block(i)(j) := Charin;
-- echo character input to terminal screen
Putchar(W_dev,Charin);
Else
Mess_rec.Block(i)(j) := "S";
End If;
End Loop; -- For Loop -- for
End Loop;
-- while
-- insert sentinel at end of buffer in case input
-- buffer size was exceeded
Mess_rec.Block(Mess_buf_size)(8) := "S";

-- count keeps track of number of blocks input
Count := i;

-- fill in address block source,dest,and num_blk
Mess_rec.Num_blk := Count;
Mess_rec.Block(1)(1) := Term_num;
Jump$ unhook;
Putln(W_dev,W_class,"");
Putln(W_dev,W_class,"ENTER DESTINATION TERMINAL NUMBER");
Getchar(R_dev,R_class,Mess_rec.Block(1)(2));
While (Mess_rec.Block(1)(2) = Term_num) Loop
Putln(W_dev,W_class,"IMPROPER DESTINATION TRY AGAIN");
Getchar(R_dev,R_class,Mess_rec.Block(1)(2));
End Loop; -- while
Putchar(W_dev,Mess_rec.Block(1)(2));
Putln(W_dev,W_class,"");
Binascii(Count,3,Blk_cnt,"0");
For i In 1..2 Loop
Mess_rec.Block(1)(i + 6) := Blk_cnt(i);
End Loop; -- For Loop
-- place mess_rec in outgoing message buffer to
-- await transmission
Move(Mess_rec,Temp_ptr,Sizeof(Mess_rec));
Buf_stat := True;
Putln(W_dev,W_class,"MESSAGE INPUT COMPLETE");
End Input_mess;

-- *****
--
-- Procedure: xmit_mess
--
-- Purpose: This procedure alerts the system manager
-- process that the operator desires to transmit the message
-- stored in the outgoing message buffer. This is done by
-- advancing the outbuf eventcount. The system manager process
-- notifies the terminal process that the message has been sent
-- by advancing the input buffer eventcount.
-- *****

Procedure Xmit_mess (inbuf_slot : in Integer;
                    outbuf_slot : in Integer;
                    inbuf_evc : in Integer;
                    xmit_buf_stat : out Boolean) Is

    Success : Integer;

Begin
    -- xmit_mess
    -- notify svsmgr, msg ready to xmit
    Advance(Outbuf_slot,Success);
    Put_succ("OUTPUT ADVANCE ERROR",Success,W_class);

    -- await sysmgr xmit complete notification
    Inbuf_evc := Inbuf_evc + 1;
    Await(Inbuf_slot,Inbuf_evc,Success);

```

```

Put_succ("AWAIT INPUT BUFFER ERROR",Success,W_class);
Putln(W_dev,W_class,"MESSAGE TRANSMISSION COMPLETE");
Xmit_buf_stat := False;
End Xmit_mess;

```

```

-- *****
--
-- Procedure: disp_mess
--
-- Purpose: This procedure displays the
-- message stored in the incoming buffer segment. It
-- is similar in structure to input_mess.
--
-- *****

```

```

Procedure Disp_mess (comn_buf : in Integer;
                    rec_buf_stat : out Boolean) Is

```

```

    Disp_rec : Buf_rec;
    Disp_ptr : Pointer;
    D_char : Character;
    i,j : Integer;
    Proc_suc : Boolean;

```

```

Begin

```

```

    -- disp_mess
    -- create pointer to incoming message buffer segment
    Disp_ptr := Lib_mk_ptr(Ldt_table,Comn_buf,1);
    Clr_screen(Proc_suc);
    Putln(W_dev,W_class,"BEGIN DISPLAY OF RECEIVED MESSAGE");
    -- place contents of incoming message buffer into
    -- disp_rec
    Move(Disp_ptr,Disp_rec,sizeof(Disp_rec));
    -- check incoming message for error message which
    -- is indicated by a source of '0'
    -- if no error message then begin display
    If Disp_rec.Block(1)(1) /= "0" Then
        Putstr(W_dev,"MESSAGE FROM TERMINAL ");
        Putchar(W_dev,Disp_rec.Block(1)(1));
        Putln(W_dev,W_class,"");
        Putstr(W_dev,"MESSAGE NUMBER: ");
        Putchar(W_dev,Disp_rec.Block(1)(3));
        Putchar(W_dev,Disp_rec.Block(1)(4));
        Putchar(W_dev,Disp_rec.Block(1)(5));
        Putln(W_dev,W_class,"");
        Putln(W_dev,W_class,"MESSAGE FOLLOWS --");
        Putln(W_dev,W_class,"");
        i := 2; -- output inbuf contents to terminal
        While (D_char /= "S") And (i /= Mess_buf_size) Loop
            For j In 1..8 Loop
                If D_char = "S" Then
                    Putchar(W_dev,Disp_rec.Block(i)(j));
                    D_char := Disp_rec.Block(i)(j);
                End If; -- if
            End Loop; -- For Loop -- for
            i := i + 1;
        End Loop; -- while
    Else
        Putln(W_dev,W_class,"MESSAGE FROM SYSTEM MANAGER");
        Putln(W_dev,W_class,"SECURITY VIOLATION");
        Putln(W_dev,W_class,"IMPROPER DESTINATION ACCESS");
        Putln(W_dev,W_class,"MESSAGE NOT DELIVERED");
    End If; -- if
    Putln(W_dev,W_class,"");
    Putln(W_dev,W_class,"END OF MESSAGE");
    Rec_buf_stat := False;
End Disp_mess;

```

```

-- *****
-- Procedure: logoff
-- Purpose: This procedure disables the TDS terminal
-- and makes the resources assigned to that terminal process
-- available. No new terminal process is created to replace it.
-- *****
Procedure Logoff ( init : in rl_process_def) Is
    Suc,Success : Integer;
Begin -- logoff
    Putln(W_dev,W_class,"TERMINATING CHILD SEGMENTS");
    -- to reinitialize a terminal process at this terminal,
    -- process segments would have to be terminated prior
    -- to the self delete call
    Putln(W_dev,W_class,"SELF DELETING CHILD PROCESS NOW");
    Putln(W_dev,W_class,"TERMINAL OFF LINE");
    Detach(W_dev);
    Detach(R_dev);
    Self delete(Init.Initial_seg(Stack_offset),Success);
    If (Success /= No_error) Then
        Attach(T_phys_dev,W_dev,False,Suc);
        Attach(T_phys_dev,R_dev,True,Suc);
        Put_succ("child self delete error",Success,W_class);
    End If; -- if
End Logoff;

-- *****
-- Procedure: clr_screen
-- Purpose: Clears display screen.
-- *****
Procedure Clr_screen (proc_suc : out Boolean) Is
    i : Integer;
Begin -- clr_screen
    For i In 1..25 Loop
        Putln(W_dev,W_class," ");
    End Loop; -- For Loop
End Clr_screen;
End DEFS;

```

APPENDIX K

TDS2 TERMINAL UTILITY PROGRAM LISTING

The TDS terminal utility module is compiled and prepared for execution in almost the same manner as the system manager module. By modifying certain parameters which are identified in the program listing, the system manager may specify the specific port and terminal number of each remote terminal process. TDS.CMD files are created for each terminal attached and are essentially identical except for assignment of specific ports and terminal numbers, therefore, only the source code for one terminal utility is provided. A TDS2.CMD file is submitted with the GUARD.SSB file for each remote terminal attached. To attach additional terminals, (as more ports become available), entry numbers would have to be specified in the GUARD.SSB file. A listing of the TDS2.KMD file is included as Appendix L.

type

```
{ common type include files }

{ Si gate-typ.zli}
{ Si lib-typ.zli}
{ Si rlp-typ.zli}
{ Si guard-typ.zli}

{ library procedure include files }

{ Si io-dec.zli}
{ Si io-hex.zli}
{ Si seg-mgr.zli}
{ Si gate.zli}
{ Si lib.zli}
{ Si io-str.zli}
```

{ ***** }

Procedure: input_mess

Purpose: This procedure allows the operator to input a message into the outgoing message buffer. Characters are input into 8 byte blocks so that they will be compatible with the data encryption device. The character 'S' is used to indicate the end of the message. The initial block is reserved for the address header. Format of the header is as follows:

```
blk[1][1]: source
blk[1][2]: destination
blk[1][3-5]: message number
blk[1][6]: classification
blk[1][7-8]: number of blocks in msg
```

Source, destination, and number of blocks are entered in this procedure. Remaining entries are filled in

by the sysmgr process prior to transmission.

```
procedure input_mess(comn_buf:integer;
                    var buf_stat:boolean);
```

```
var
```

```
    mess_rec: buf_rec;
    blk_cnt: string;
    temp_ptr,input_ptr: pointer;
    charin:char;
    i,j,k,succes:integer;
    count: integer;
    proc_suc: boolean;
```

```
begin {input_mess}
```

```
    { create pointer to start of input buffer }
```

```
input_ptr:= lib_mk_pntr(ldt_table,comn_buf,1);
```

```
temp_ptr:= input_ptr;
```

```
clr_screen(proc_suc);
```

```
putln(w_dev,'ENTER MESSAGE TO BE TRANSMITTED');
```

```
putln(w_dev,'ENTER A $ TO INDICATE END OF MESSAGE');
```

```
putln(w_dev,'');
```

```
    { initialize msg block counter }
```

```
    i:= 1;
```

```
    { begin character entry loop }
```

```
while(charin <> '$') and
```

```
    (i <> mess_buf_size + 1) do begin
```

```
    { block 1 is addr, block 2 is strt of msg }
```

```
    i:= i + 1;
```

```
    { begin loop to read 8 char for each block }
```

```

    for j:= 1 to 8 do begin
        if charin < > 'S' then begin
            getchar(r_dev,charin);
            mess_rec.block[i][j]:= charin;

            { echo character input }

            putchar(w_dev,charin);

        end else

            { if charin='S' then pad the remaining
              entries with 'S' to avoid sending an
              incomplete block }

            mess_rec.block[i][j]:= 'S';

    end; {for}

end; {while}

    { insert sentinel at end of buffer in case input
      buffer size was exceeded }

mess_rec.block[mess_buf_size][8]:= 'S';

    { count keeps track of number of blocks input }

count:= i;

    { fillin address block source,dest,and num_blk }

mess_rec.num_blk:= count;
mess_rec.block[1][1]:= term_num;

putln(w_dev, ' ');
putln(w_dev, 'ENTER DESTINATION TERMINAL NUMBER');
getchar(r_dev,mess_rec.block[1][2]);
putchar(w_dev,mess_rec.block[1][2]);
putln(w_dev, ' ');

```

```
binascii(count,3,blk_cnt,'0');
```

```
for i:= 1 to 2 do
```

```
    mess_rec.block[1][i+6] := blk_cnt[i];
```

```
    { place mess_rec in outgoing message buffer to  
      await transmission }
```

```
move(mess_rec,temp_ptr, sizeof(mess_rec));
```

```
buf_stat:= true;
```

```
putln(w_dev,'MESSAGE INPUT COMPLETE');
```

```
end; {input_mess}
```

```
{ *****
```

```
    Procedure: xmit_mess
```

Purpose: This procedure alerts the system manager process that the operator desires to transmit the message stored in the outgoing message buffer. This is done by advancing the outbuf eventcount. The system manager process notifies the terminal process that the message has been sent by advancing the input buffer eventcount.

```
***** }
```

```
procedure xmit_mess(inbuf_slot:integer;outbuf_slot:integer;
```

```
    var inbuf_evc:integer;
```

```
    var xmit_buf_stat:boolean);
```

```
var
```

```
    success:integer;
```

```
begin {xmit_mess}
```

```
    { notify sysmgr, msg ready to xmit }
```

```
    advance(outbuf_slot,success);
```

```

show_err('OUTPUT ADVANCE ERROR',success);

    { await sysmgr xmit complete notification }
await(inbuf_slot,inbuf_evc + 1,success);
show_err('AWAIT INPUT BUFFER ERROR',success);

inbuf_evc:= inbuf_evc + 1;

putln(w_dev,'MESSAGE TRANSMISSION COMPLETE');

xmit_buf_stat:= false;

end; {xmit_mess}

{ *****

    Procedure: disp_mess

    Purpose: This procedure displays the
message stored in the incoming buffer segment. It
is similar in structure to input_mess.

***** }

procedure disp_mess(comn_buf:integer;
                    var rec_buf_stat:boolean);

var

    disp_rec:buf_rec;
    disp_ptr:pointer;
    d_char:char;
    i,j:integer;
    proc_suc: boolean;

begin {disp_mess}

    { create pointer to incoming message buffer segment}

disp_ptr:= lib_mk_pntr(ldt_table,comn_buf,1);

clr_screen(proc_suc);

```

```

putln(w_dev,'BEGIN DISPLAY OF RECEIVED MESSAGE');

    { place contents of incoming message buffer into
      disp_rec }

move(disp_ptr                                ,disp_rec,sizeof(disp_rec));

    { check incoming message for error message which
      is indicated by a source of '0' }

    { if no error message then begin display }

if disp_rec.block[1][1] < > '0' then begin

    putstr(w_dev,'MESSAGE FROM TERMINAL');
    putchar(w_dev,disp_rec.block[1][2]);
    putln(w_dev,' ');
    putln(w_dev,'MESSAGE FOLLOWS --');
    putln(w_dev,' ');
    i:= 1;

    { output inbuf contents to terminal }
    while (d_char < > 'S') and
        (i < > mess_buf_size + 1) do begin

        for j:= 1 to 8 do begin

            if d_char < > 'S' then begin

                putchar(w_dev,disp_rec.block[i][j]);
                d_char:= disp_rec.block[i][j];

            end; {if}

        end; {for}

        i:= i + 1;

    end; {while}

end else begin

    putln(w_dev,'MESSAGE FROM SYSTEM MANAGER');

```

```

        putln(w_dev,'SECURITY VIOLATION');
        putln(w_dev,'IMPROPER DESTINATION ACCESS');
        putln(w_dev,'MESSAGE NOT DELIVERED');

end; {if}

putln(w_dev,'END OF MESSAGE');

rec_buf_stat:= false;

end; {disp_mess}

{*****}

        Procedure: logoff

        Purpose: This procedure disables the TDS terminal
        and makes the resources assigned to that terminal process
        available. No new terminal process is created to replace it.

        *****}

procedure logoff(init:r1_process_def);

var

        suc.success:integer;

begin {logoff}

        putln(w_dev,'TERMINATING CHILD SEGMENTS');

                { to reinitialize a terminal process at this terminal,
                  process segments would have to be terminated prior
                  to the self_delete call }

        putln(w_dev,'SELF DELETING CHILD PROCESS NOW');
        putln(w_dev,'TERMINAL OFF LINE');

        detach(w_dev);
        detach(r_dev);

        self_delete(init.initial_seg[stack_offset],success);

```

```

        if (success < > no_error) then begin
            attach(5,w_dev,false,suc);
            attach(5,r_dev,true,suc);
            show_err('child self delete error',success);
        end; {if}

end; {logoff}

{*****}

    Procedure: show_err

    Purpose: This procedure is called to
display the success code of the resource mngmnt
call if it is other than zero. If the success code
indicates no_error then no message is output.

*****}

procedure show_err(str:string; code:integer);

begin {show_err}

    if code < > no_error then begin

        putstr(w_dev,str);
        putstr(w_dev,' ');
        putdec(w_dev,code);
        putln(w_dev,' ');

    end;

end; {show_err}

{*****}

```

Procedure: clr_screen

Purpose: Clears display screen.

```
*****}
```

```
procedure clr_screen(proc_suc:boolean);
```

```
var
```

```
    i:integer;
```

```
begin {clr_screen}
```

```
for i:= 1 to 25 do
```

```
    putln(w_dev,' ');
```

```
end; {clr_screen}
```

```
{*****}
```

Procedure: main

Purpose: This procedure provides a menu for the terminal operator. It monitors buffer status and calls the appropriate procedure dependent on the mode selection.

```
*****}
```

```
procedure main( var init : rl_process_def );
```

```
var
```

```
    DEBUG : INTEGER;      { Used for debugging }
```

```
    DEBUGPTR : POINTER;   { Used for debugging }
```

```
    success : integer;
```

```
    seg_num:integer;
```

```
    mode:char;
```

```
    xmit_buf_stat,rec_buf_stat:boolean;
```

```
    temp_str:string[1];
```

```
    i,level:integer;
```

```
    inbuf_evc: integer;
```

```
    stk_evc:integer;
```

```
    sys_start:boolean;
```

```

begin {main}

    { initialize terminal process parameters }

xmit_buf_stat:= false;
mode:= '0';

    { sys_start= false for twterminal 1 only all
      other terminals should have sys_start= true}

sys_start:= false;
inbuf_evc:= 0;

    { attach terminal as read/write device }

attach(5,w_dev,false,success);
if (success < > no_error) then
begin

    DEBUG := success;
    swapin_segment(init.initial_seg[outbuf_offset], success);
    DEBUGPTR := lib_mk_pntr(ldt_table,
                           init.initial_seg[outbuf_offset], 1);
    DEBUGPTR      := DEBUG;

end; {if}

attach(5,r_dev,true,success);
if (success < > no_error) then
begin

    show_err('ATTACH R_DEV FAILED ', success);

end; {if}

putln(w_dev,' ');
putln(w_dev,'TERMINAL ACTIVE');
putchar(w_dev,term_num);
putln(w_dev,' ');

    { stack eventcount is advanced to notify

```

```

    sysmgr that terminal is activated }
    advance(init.initial_seg[code_offset],success);
    advance(init.initial_seg[stack_offset],success);
    show_err('STACK ADVANCE ERROR',success);

    { loop until operator enters 'e' to indicate logoff }

while mode < > 'e' do begin

    { inbuf_evc is used to have the terminal wait after
      transmitting a message until a reply is received
      from the dest term. It is initially advanced for
      terminal 1 to start the system and then is advanced
      upon receipt of an incoming message }

    await(init.initial_seg[inbuf_offset],inbuf_evc + 1,success);
    show_err('AWAIT INCOMING MESSAGE',success);

    inbuf_evc:= inbuf_evc + 1;

    { sys_start is used to avoid the 'display incoming
      message' prompt at terminal 1 when the system is
      started. Once the system is operating it will
      always be true }

    if sys_start = true then begin

        rec_buf_stat:= true;
        putln(w_dev,'DISPLAY INCOMING MESSAGE');

    end; {if}

    sys_start:= true;

    { inner loop is used to indicate that a
      message has been sent and alert the operator
      that the terminal is waiting for a reply }

    while mode < > 'x' do begin

```

```

{ help menu consists of a display of term
  access level, and a display of possible
  modes }

  putstr(w_dev,'TERMINAL COMPROMISE LEVEL');
  level:= init.root_access.compromise[1];

  case level of

    0: putln(w_dev,'UNCLASSIFIED');
    2: putln(w_dev,'CONFIDENTIAL');
    4: putln(w_dev,'SECRET');
    6: putln(w_dev,'TOP SECRET');

  end; {case}

  putln(w_dev,'ENTER MODE DESIRED');
  putln(w_dev,'I = INPUT MESSAGE');
  putln(w_dev,'D = DISPLAY RECEIVED MESSAGE');
  putln(w_dev,'X = TRANSMIT MESSAGE');
  putln(w_dev,'E = LOGOFF');
  putln(w_dev,' ');
  putstr(w_dev,'ENTER MODE HERE');

  getchar(r_dev,mode);

  if mode= 'I' then begin
    if xmit_buf_stat= false then begin
      { enter message to be stored in
        outgoing message buffer }

      input_mess(init.initial_seg[outbuf_offset],
        xmit_buf_stat);

    end else begin

      putln(w_dev,'MESSAGE WAITING TO BE TRANSMITTED');

    end;
  end;

```

```

end else if mode = 'D' then begin
    if rec_buf_stat = true then begin
        putln(w_dev, 'ENTERING DISPLAY MODULE');
        { display contents of incoming
          message buffer }
        disp_mess(init.initial_seg[inbuf_offset],
                  rec_buf_stat);
    end else begin
        putln(w_dev, 'INCOMING BUFFER EMPTY');
    end;
end else if mode = 'X' then begin
    if xmit_buf_stat = true then begin
        putln(w_dev, 'SENDING MESSAGE');
        xmit_mess(init.initial_seg[inbuf_offset],
                  init.initial_seg[outbuf_offset],
                  inbuf_evc, xmit_buf_stat);
    end else begin
        putln(w_dev, 'OUTGOING BUFFER EMPTY');
    end;
end else if mode = 'E' then begin
    putln(w_dev, 'LOGOFF PROCESS INITIATED');
    logoff(init);
end else
    putln(w_dev, 'MODE ENTRY ERROR. TRY AGAIN');
{ end of inner loop-exit after msg xmit ;

```

```

        end; {while}

putln(w_dev, 'WAITING FOR INCOMING MESSAGE');

        { reset mode selection value }

mode:= '0';

end; {while}

        putln(w_dev, 'END OF TDS TERMINAL PROCESS');
        detach(w_dev);
        detach(r_dev);

        {infinite loop to avoid crash}

while true do;

end; {main}

modend.

```

APPENDIX L
TDS2.KMD LINKING FILE FOR TDS2.PAS

{ *****

Program: TDS2.KMD

Date: 7 February 1987

Author: G. E. RECTOR, JR., CAPT/USMC

Advisor: T. J. BROWN, MAJ/USAF

Purpose: This program is used when linking the
TDS2 terminal utility program after it is compiled. It
eliminates the need to manually enter each of the file
names each time a new version or update of the program
is compiled.

*****}

tds2 = r1-init, tds2, rllib/s, paslib/s/p:80

APPENDIX M

GUARD.SSB SYSTEM GENERATION SUBMIT FILE

This file is used to sysgen the system. Each file is a command file used to create the initial segments.

```
bs:ld3.cmd
ks:k0.cmd
ks:k1.cmd
ks:nv.cmd
ks:k2.cmd
ks:nv.cmd
ks:k3.cmd
ks:k4.cmd
ks:nv.cmd
cs:vloader.cmd;2;
ds:vlogin.cmd;2,10;
ds:rltrap.cmd;4;
ds:nv.ds;2,5;
ds:nv.ds;5;
ds:guard.cmd;5,0;
ds:tds1.cmd;5,2;
ds:tds2.cmd;5,3;
end
```

LIST OF REFERENCES

1. Klein, M. H., "Computer Security," *Issues in C3I Program Management*, Ed. Jon L. Boyes, AFCEA International Press, 1984.
2. Peterson, J. L., and Siiberschatz, A., *Operating System Concepts*, Massachusetts: Addison-Wesley, 1985.
3. Department of Defense Computer Security Center, *DoD Trusted Computer System Evaluation Criteria* CSC-STD-001-83, Fort Meade, Maryland, 15 August 1983.
4. Headquarters, United States Marine Corps, Washington, D. C., *U. S. Marine Corps Command and Control Master Plan (C2MP)*, (*Draft*), December 1986.
5. Marine Corps Tactical Systems Support Activity, *Tactical Combat Operations System Description Document*, Marine Corps Base, Camp Pendleton, California, 1981.
6. Marine Corps Tactical Systems Support Activity, *Technical Interface Concept for Marine Tactical Systems*, Marine Corps Base, Camp Pendleton, California, 12 September 1978.
7. Anderson, J. P., "Computer Security Technology Planning Study," *ESD-TR-73-51, Volume I and II*, AD758206, AD772806, James P. Anderson & Co., Fort Washington, Pennsylvania, October 1972.
8. Ames, S. R., Jr., "Security Kernels: A Solution or a Problem," *Proceedings of the 1981 Symposium on Security and Privacy*, IEEE Computer Society Press, Los Angeles, California, April 1981.
9. Woodward, J. P. L., "ACCAT Guard System Specification (Type A)," *MTR-3634*, The MITRE Corporation, Bedford, Massachusetts, August 1978.
10. Woodward, J. P. L., "Applications for Multilevel Secure Operating Systems," *1979 National Computer Conference*, New York, New York, June 1979.
11. Anderson, J. P., *On The Feasibility of Connecting RECON to an External Network*, Fort Washington, Pennsylvania, March 1981.
12. Diffie, W., and Hellman, M. E., "New Directions in Cryptology," *IEEE Transactions on Information Theory*, IT-22, 6 November 1976.
13. National Bureau of Standards, Federal Information Processing Standard, FIPS Publication 46, *Data Encryption Standard*, January 1977.
14. Davio, M., and others, "Analytical Characteristics of the DES," *Advances in Cryptology - Proceedings of Crypto 83*, ed D. Chaum, Plenum Press, Inc., 1983.

15. National Bureau of Standards, Federal Information Processing Standard, FIPS Publication 81, *DES Modes of Operation*, 2 December 1980.
16. Voydock, V., and Kent, S., "Security in High-Level Network Protocols," *Computing Surveys*, Vol. 15, No. 2, June 1983.
17. Gemini Computers Inc., Monterey, California, *System Overview*, Gemini Trusted Multiple Microcomputer Base, Version 0, May 1984.
18. Boebert, E., Kain, R., and Young, B., "Trojan Horse Rolls Up to DP Gate," *Computerworld*, 2 December 1985.
19. Gemini Computers Inc., Monterey, California, *Sysgen User's Manual*, June, 1986.
20. Gemini Computers Inc., Monterey, California, *GEMSOS Ring 0 User's Manual for the Pascal MT+86 Language*, July 1986.
21. Reed, D. P., and Kanodia, R. K., "Synchronization with Eventcounts and Sequencers," *Communications of the ACM*, Vol. 22, No. 2, February 1979.
22. Brewer, D. J., *A Real-Time Executive for Multiple Computer Clusters*, Masters Thesis, Naval Postgraduate School, Monterey, California, December 1984.
23. Digital Research, Inc., Monterey, California, *CPM-86 Operating Manual*, 1983.

BIBLIOGRAPHY

Barnes, J. G. P., *Programming In ADA*, Addison-Wesley Publishers Limited, 1984.

Brand, S., "Environmental Guidelines For Using DOD Trusted Computer System Evaluation Criteria", *Proceedings of the 7th DOD/NBS Computer Security Conference*, September 1984.

McMillan, B., *DTST REPORT: Volume 5 "Battle Management, Command, Control, Communication and Data Processing"*, October 1983.

Morgan, B., "Considerations in Applying an Encryption Device to a Communications Network," *1977 Conference on Computer Security and the Data Encryption Standard*, U. S. Department of Commerce, Springfield, Virginia, February 1978.

Peterson, J. L., and Siiberschatz, A., *Operating System Concepts*. Massachusetts: Addison-Wesley, 1985.

Schell, R., and Tao, T. F., "Microcomputer Based Trusted Systems For Communication and Workstation Applications", *Proceedings 7th DOD/NBS Computer Security Conference*, September 1984.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2. Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3. Thomas J. Brown, Code 39 Naval Postgraduate School Monterey, CA 93943-5000	2
4. George E. RECTOR, Jr. 10013 Laurant Place Nokesville, Va 22123	2
5. Joseph S. Stewart, Code 55ST Naval Postgraduate School Monterey, CA 93943-5000	2
6. C3 Academic Group, Code 74 Prof. M. K. Sovereign Naval Postgraduate School Monterey, CA 93943-5000	2
7. Commandant of the Marine Corps (Code C4,CCA) Headquarters Marine Corps Washington, D.C. 20380	2
8. Commandant of the Marine Corps (Code INTM) Headquarters Marine Corps Washington, D.C. 20380	2
9. Chief, Command, Control and Communications Division (D103FT) Marine Corps Development Center Marine Corps Development and Education Command Quantico, VA 22134-5080	2
10. Chief, Intelligence Division Marine Corps Development Center Marine Corps Development and Education Command Quantico, VA 22134-5080	2
11. Dr. Tien Tao Gemini Computers Inc. P.O. Box 222417 Carmel, CA 93922	2
12. Dr. Roger Schell Gemini Computers Inc. P.O. Box 222417 Carmel, CA 93922	2

END

9-87

Dtic