(2)

TECHNICAL SUPPORT TASK REPORT FOR THE MODERNIZATION OF DEFENSE LOGISTICS STANDARD SYSTEMS

Volume II: Logistics Gateway Node Prototype Construction and Operation

Report DL702R1

April 1991

William T. James, III

With
Christo G. Andonyadis
John S. Doby
John Lycas

Prepared pursuant to Department of Defense Contract MDA903-90-C-0006.

LOGISTICS MANAGEMENT INSTITUTE
6400 Goldsboro Road
Bethesda, Maryland 20817-5886

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1991	Final

4. TITLE AND SUBTITLE

Technical Support Task Report for the Moderization of Defense Logistics Standard Systems – Volume II: Logistics Gateway Node Prototype Construction and Operation

5. FUNDING NUMBERS

C MDA903-90-C-0006
PE 0902198D

6. AUTHOR(S)

William T. James, III with Christo G. Andonyadis, John S. Doby, John Lycas

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Logistics Management Institute
6400 Goldsboro Road
Bethesda, MD 20817-5886

8. PERFORMING ORGANIZATION REPORT NUMBER

LMI-DL702R1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Logistics Standard Systems Division
6301 Little River Turnpike, Suite 210
Alexandria, VA 22312

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

A: Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This volume details the construction and operation of the Prototype Logistics Gateway Node (LGN) developed for the support task. The document is written from the perspective of software performance and is intended to assist in understanding and implementing the technical specification developed for the LGN.

14. SUBJECT TERMS

MODELS, EDI Translation, Logistics Gateway Node, LGN, CLGN, Prototype, MODELS Feasibility Test

15. NUMBER OF PAGES

170

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	18. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

PREFACE

This technical report, in three volumes, is the final report covering more than 2 years of technical activity supporting the Modernization of Defense Logistics Standard Systems (MODELS) project. The supporting activities included developing translation tables and the table-driven software for converting current fixed-length logistics data formats into new variable-length transaction equivalents. They also included designing and testing prototype hardware and software platforms that support transaction interchange between logistics sites.

This volume, *Logistics Gateway Node Prototype Construction and Operation*, is Volume II of the series. It details the construction and operation of the prototype logistics gateway nodes (LGNs) developed for the support task. The document is written from the perspective of software performance. It is provided to assist in understanding and implementing the technical specification developed for the LGN.

Volume I, *Prototype Test Report*, is an overview describing the task's purpose, results, conclusions, and recommendations from the viewpoint of four major support activities:

- Prototype LGN construction and testing
- Interconnection and control of telecommunicating LGNs
- Electronic data interchange transaction translation and testing
- Network performance simulation and cost modeling.

Volume III, *Logistics Gateway Node Technical Specification*, presents the performance requirements for an LGN and central LGN (CLGN) interconnected within a homogeneous network of LGNs under CLGN control. The specification's purpose is to describe the technical capabilities necessary for a network of deployed LGNs to meet the functional capability called for in OSD directives.

CONTENTS

CONTENTS (Continued)

CONTENTS (Continued)

TABLES

FIGURES

FIGURES (Continued)

CHAPTER 1

BACKGROUND

The mission of a logistics gateway node (LGN) is to facilitate the two-way translation of Defense Logistics Standard Systems (DLSS) and electronic data interchange (EDI) transactions at a particular site; each LGN serves as a communications interface, or gateway, between the local host computer and an X.25 wide area network (WAN), to which are attached the other LGNs.

During Phase I of the Modernization of Defense Logistics Standard Systems (MODELS) test, the LGN was strictly a translator invoked manually from the keyboard each time a translation was desired. In Phase II, the LGN ran unattended, except to simulate manually the download of a file from a host. Dial-up, modem-to-modem communication was added. The various tasks in the Phase II model ran serially. The Phase III prototype LGN also operates unattended and communicates with a local host and the WAN but is different in that it incorporates multitasking, enabling its component processes to run in parallel.

This volume describes the MODELS Phase III prototype LGN in considerable depth, as a supplement to another Logistics Management Institute (LMI) volume of this report, the *Logistics Gateway Node Technical Specification*.

CHAPTER 2

OPERATING REVIEW

The prototype LGN is a front-end processor for a host computer that transmits and receives DLSS transactions. It functions as an interface point between its host and an X.25 WAN. A specially configured central LGN (CLGN), eventually to be sited at the Defense Automatic Addressing System (DAAS) Office (DAASO), serves as an intermediate processing point for transactions that need to go through DAAS. During Phase III, a prototype CLGN was configured at LMI.

The path taken by a file of transactions via the LGN is as follows: the file is downloaded from the host, goes through translation at the LGN, is sent over the WAN, and arrives at the destination LGN, where it is retranslated. The next, and final, logical step of uploading to the receiving host was not implemented in the prototype LGN, as a result of restrictions on accessing the host computers at the test sites. For those transactions requiring processing by DAAS, two transmissions – an LGN-to-CLGN transmission, followed by a CLGN-to-LGN one – take place. For most of the test, all transactions were routed to the CLGN. Figure 2-1 depicts the overall information flow at this abstract level.

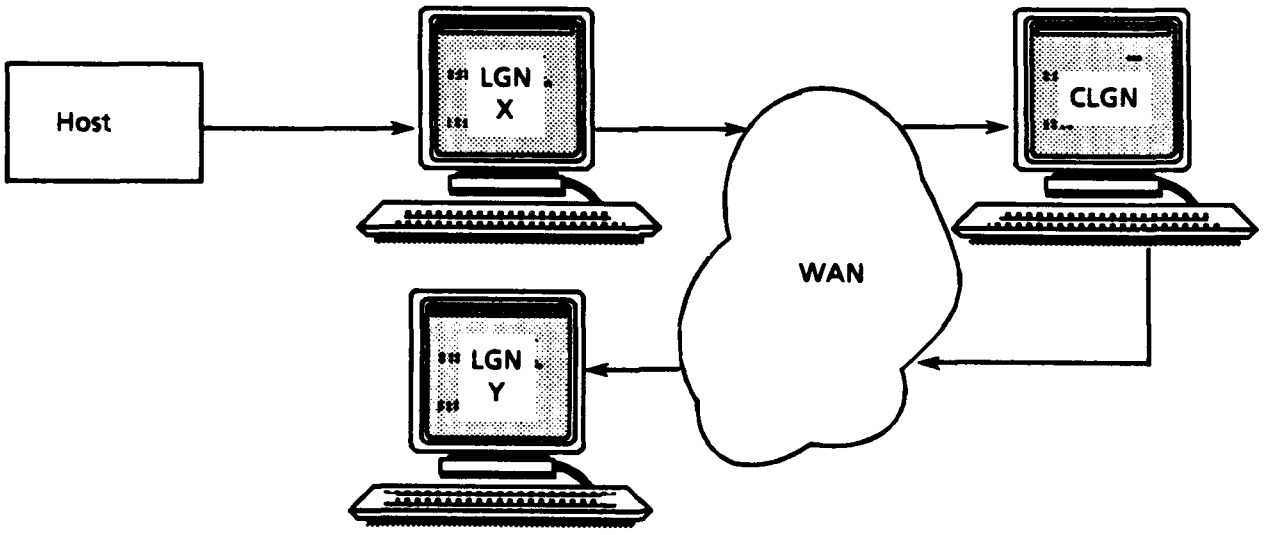


FIG. 2-1. OVERALL LGN INFORMATION FLOW

The rest of this chapter describes an end-to-end network transmission as implemented in Phase III. Although the flow of transactions through the system is described as though it were serial, at any time there may be simultaneous information flow in various directions and processing stages within an LGN.

Transaction flow begins when the LGN receives a file of outbound DLSS transactions from its host through a local communications interface tailored to the host. Figure 2-2 is a high-level view of the flow of information from the host through the LGN to the WAN. Connection to the host may be constant, or it may be initiated on a periodic basis by the LGN. LGN start-up parameters determine the frequency and, to some extent, the nature of the connection to and downloading of transactions from the host. The software for connecting and downloading is selected and/or customized to work with the host's hardware, software, and communications environment.

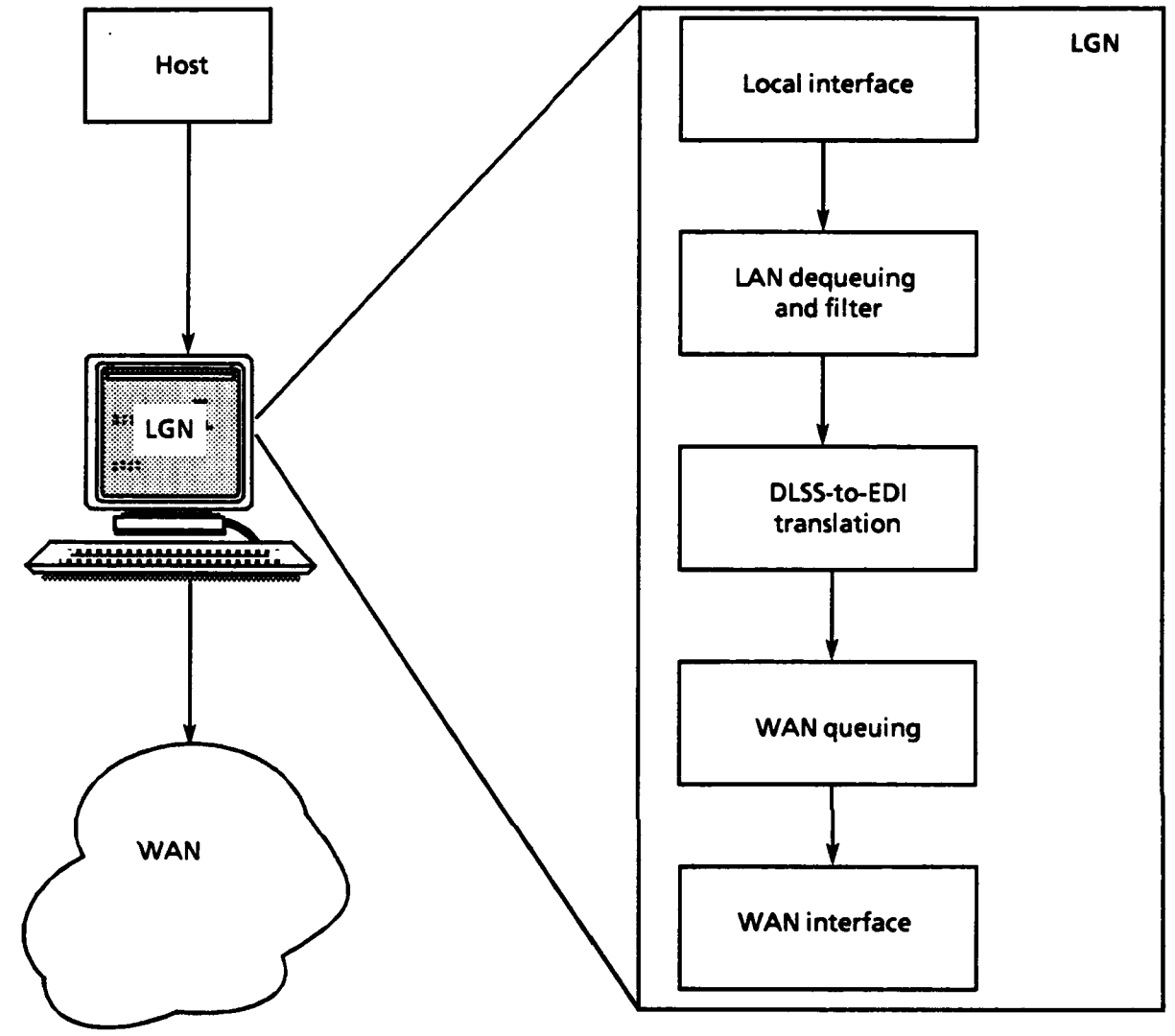
Upon successfully downloading a file of DLSS transactions from the host, the LGN assigns a unique 11-character[1] name to the file; an 8-character timestamp that will stay with the file through all phases of its processing (including retranslation at the receiving LGN), and a 3-character extension indicating the type of file (e.g., raw DLSS, filtered DLSS, error file). At each step of LGN processing, any intermediate or result files produced are assigned a name consisting of the file's 8-character timestamp plus a specific 3-character extension.

The LGN filters downloaded transactions to select only those appropriate to the test. The filtering mechanism is sufficiently flexible to handle an expanding set of filtering criteria. Each transaction that passes the filter is assigned a unique control number[2] for tracking and for functional source-to-destination verification. Transactions not passing the filter are written to a temporary log that is overwritten each time a batch of transactions is filtered, in order to conserve hard disk space (the prototype LGN does not use erasable media such as Write-Once-Read-Many (WORM) optical disks). In actual operation, all error log entries would be kept until no longer needed. Moreover, transactions from the host not passing the filter would be sent back to the host as well, where they would be corrected or discarded.

---

[1]Eleven-character file names are the maximum allowed in MS-DOS.

[2]The control number is a concatenation of the LGN's unique identifier (a 3-character mnemonic; e.g., DCS) with a hyphen and a 10-digit sequence number (e.g., DCS-1000000001), identifying any transaction in the system uniquely.



Note: LAN = local area network.

FIG. 2-2. INFORMATION FLOW FROM HOST THROUGH LGN

Using table-driven logic, the LGN translates the DLSS transactions into EDI transactions and packages them for transmission on an X.25 WAN. Currently, all transactions are sent to the CLGN, and an intermittent routing table based on LGN source is employed at the CLGN. In this simple prototype implementation, each LGN name is paired with another LGN name; thus, the source LGN for a file of transactions uniquely determines the file's ultimate destination. This rudimentary routing logic is used primarily to "complete the circuit" from the source LGN through the CLGN to the destination LGN. In contrast, an operational or production system's

routing logic would depend on the type of each transaction and on address-table entries based on a more complex set of system-based rules.

Once translated to EDI, the transaction file is compressed by the PKware data compression software (PKARC) and queued for transmission across the WAN. The compressed file is assigned a unique 11-character name consisting of an 8-character timestamp and the 3-letter LGN identifier. Embedded within the compressed file is the original file name, so its timestamp (assigned immediately after downloading from the host) is retained. As soon as the transaction file is bundled (and the WAN line is free), the LGN sends the compressed EDI transactions file, using an enhanced XMODEM file transfer over an X.25 protocol. The LGN looks up the destination routing identifier code (called the TORIC) in a file that maps the TORIC to the network address of the LGN serving the TORIC. Although in the prototype system all transactions go to the CLGN, a generalized mapping file not tailored specifically to the CLGN is used. If the call or transmission fails, it will be retried at prespecified intervals for a prespecified number of tries. The progress at each step of the X.25 session, including file transfer statistics, is logged either as an event or as an error, depending on its success.

The flow of information from the WAN to the receiving LGN, as implemented in the prototype system, is shown in Figure 2-3. As with sending files, the receiving LGN logs each step of the file reception, including the network address of the source LGN (this is a dial-up node if the source LGN is not directly connected to the WAN), the name of the file received, the elapsed file transfer time, and a breakdown of the X.25 packets exchanged. The source LGN name is derived from the file name, whose last three letters are the source LGN's unique mnemonic.[3]

Once the EDI file is received, the LGN expands the compressed file and translates each EDI transaction contained within it into the DLSS format. The result is a DLSS file, an error file, and a compare file that reside on the receiving LGN. The compare file is the result of a comparison made by the translator between the original DLSS transactions and the output of the EDI-to-DLSS translation. Because

[3]During the test, the original DLSS transaction was embedded in the EDI transaction (as one or more XXX segments), making a before-and-after comparison possible. The original transaction will not be carried in the production system.

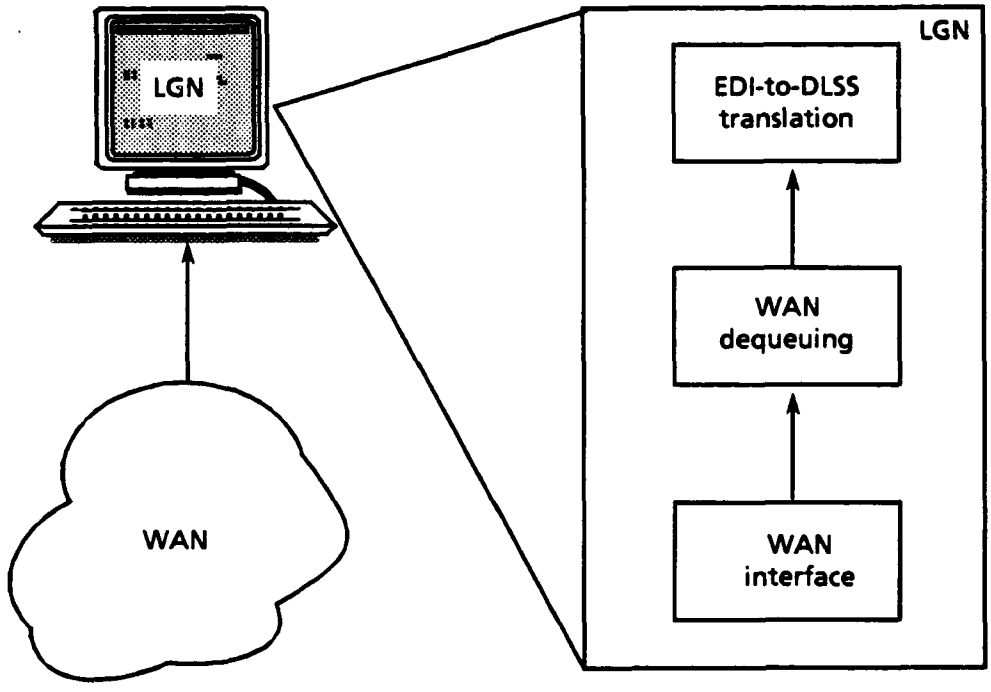


FIG. 2-3. INFORMATION FLOW FROM WAN THROUGH LGN

of security restrictions in effect during the prototype operation, as well as other practical considerations, the resultant DLSS file is not uploaded to the host.

At each step within the LGN, the status of processing is written to an event and error log. There is one log per module; each log contains both event and error messages, which are distinguishable by their formats. This arrangement allows a file of transactions to be tracked from end to end and provides an audit trail for files, but not at the transaction level. (Such will not be the case, of course, in a production LGN; in it, end-to-end serialization and logging of all transactions will enable full auditing down to the transaction level.)

Periodically, the log files are sent to the CLGN and subsequently cleared; this operation is initiated via remote commands from the CLGN. Other ad hoc tasks carried out remotely from the CLGN include table and file updates, file requests, system commands (performed at the LGN), and module suspend and restart.

For the most part, the prototype CLGN,[4] located at LMI, operates like any other LGN, with these differences:

- The biggest difference is that, at the prototype CLGN (1) EDI transactions are retranslated to DLSS; (2) the resulting DLSS transactions are translated back to EDI, and then (3) the EDI transaction file is forwarded to its final destination LGN.[5] This process is carried out to simulate (roughly) the retranslation of certain transactions required of a production CLGN. Because of time restrictions on development, the prototype CLGN does not preserve transaction control numbers during retranslation from DLSS to EDI; rather, new control numbers are assigned to each transaction. Likewise, the prototype CLGN does not compare incoming EDI transactions with the retranslated EDI transactions.

- If the destination of a final EDI transaction file cannot be determined, the file resides on the hard disk at the CLGN; in this respect, the prototype CLGN acts as an entrepot for certain transactions.

- File names for intermediate and result files produced by the various modules follow a slightly different naming convention to preserve the source LGN name, which is embedded in all the CLGN intermediate and result file names. This method for associating a source LGN with a file is a stopgap; in the production CLGN, the unique transaction identifier and the more complete logs must provide for much more certain tracking of any transaction's source LGN.

- The prototype CLGN is able to perform certain remote operations, such as table updates, remote system commands, and module suspends and restarts, that an ordinary LGN cannot. Furthermore, remote LGNs are restricted to WAN communication between themselves and the CLGN, whereas the CLGN can communicate with any other LGN. This restriction may be less stringent in a production system: direct LGN-to-LGN communication may be allowable when specified conditions are matched in the LGN routing table and the destination LGN is connected directly to the WAN.

In the prototype system, one can reconfigure an LGN as a CLGN or vice versa with a minimum of program recompiling. Some minor design changes would allow an LGN's status to be switched between that of regular LGN and that of CLGN

---

[4]Actually, there are two essentially duplicate CLGNs located at LMI, to ensure a reasonable availability over the WAN and to ensure sufficient disk space. However, from the standpoint of an LGN, there exists only one CLGN – the one it communicates with – and thus for all practical purposes it makes sense to refer to *the* CLGN as though there were only one.

[5]In the prototype, this step is usually bypassed for convenience. Skipping this step, however, does not materially reduce the test's effectiveness, since all procedures performed at the destination LGN are performed at the prototype CLGN as well.

simply by changing one or more parameter values. An LGN can determine whether or not it is the CLGN via a parameter read-in at LGN start up. This flexibility will not be a factor in the production system, since the location of the CLGN will be fixed.

CHAPTER 3

OVERVIEW OF REQUIREMENTS

LOCAL CONNECTIVITY

The most difficult requirement to resolve in the LGN is connectivity to the host system. This is particularly true in the prototype environment, since the prototype LGN is basically "at the mercy of" the host environment. The host interfaces encountered during the development and operation of the prototype system are

- 3270 terminal emulation
- Asynchronous terminal emulation.

The asynchronous environment consists of various terminal emulations, depending on the host. A design goal of the prototype LGN is to insulate the site-specific connectivity requirements from the remainder of the LGN processing as much as possible. For the most part, this goal has been achieved, by having separate host-specific processes that are invoked by standard LGN processes. To the extent possible, host-specific operational considerations are table driven.

In general, the LGN periodically initiates a logical session with its host and polls it to determine whether there are transactions to download. While the frequency and nature of the polling vary somewhat from LGN to LGN, the following basic steps are always taken:

- *Step 1A* – If the connection to the host is hard wired, establish a host connection at LGN start up.
- *Step 1B* – If the connection to the host is not hard wired, establish a temporary host connection at the beginning of each poll.
- *Step 2* – By looking for a predefined file or set of files in a particular directory (e.g., UNIX, etc.) or data set, determine whether a file exists to download.
- *Step 3* – If a target file exists, download it and pass it to the next step in the LGN. Because of security requirements in place during the operation of the prototype system, files on the host cannot be deleted. Therefore, logic in the

LGN minimizes but does not eliminate the chance that the same data file is downloaded twice. This is a known limitation of the prototype LGN.

- *Step 4A* – If no file exists, do nothing.
- *Step 4B* – If the host connection is not hard wired, disconnect from the host.
- *Step 5* – Sleep until the next poll of the host.
- *Step 6* – Return to Step 1B or Step 2.

WIDE AREA NETWORK CONNECTIVITY

In the prototype system, all WAN connectivity is by way of a commercial X.25 WAN provider. Some LGNs have a direct connection to the WAN, making them directly addressable by the CLGN; others have a dial-up connection, in which case they poll the CLGN periodically to receive files or remote commands addressed to them. If an LGN is directly addressable, the sender initiates the transfer of data; for those LGNs with a dial-up connection, data transfers are initiated by the LGN, whether it is the sender or the receiver.

CLGN AND LOCAL LGNS

The CLGN, as indicated in Chapter 2, is essentially a superset of an LGN, with unique remote operations and maintenance capabilities. It also has some processing capabilities related to its (and DAASO's) role as a transaction entrepot. Generally, however, the translation, communications, and logging functions are common to both the LGNs and the CLGN. All LGN descriptions in this document refer to both the LGN and the CLGN, unless otherwise noted.

TRANSACTION PROCESSING

Since the prototype LGN handles files of transactions rather than individual transactions, it is not a transaction processor in the commonly understood meaning of the term. For the test, all transactions are considered to be of equal priority. Transactions are processed sequentially, in the order received from the host (or the WAN), except that some multiple card-image DLSS transactions are sorted during filtering and may be in a different order in the resultant filtered DLSS file. Guaranteed minimum processing times are not inherent in the prototype LGN's design.

THROUGHPUT

Average sustained throughput rates in a single LGN are as follows:

- Download: In the prototype system, download speeds differ significantly from LGN to LGN and depend almost entirely on available file transfer software at the site. In general, the highest download throughput rates have been achieved with file transfer software that had both a host and an LGN component and that executed in a cooperative manner. Of course, the line speed of the LGN-to-host connection is a limiting factor. For 3270 connections through a concentrator, competing communications activity from connected terminals is also a big factor. In other words, the download throughput rates given here are rough and should be viewed as an approximate baseline.
  - 3270 direct connect with intelligent host transfer software: 60,000 to 120,000 transactions per hour.
  - Asynchronous connection using Kermit: 9,000 transactions per hour.
- DLSS transaction filtering: 108,000 transactions per hour (1,800 per minute).
- DLSS-to-EDI translation: 7,200 transactions per hour.[1]
- File compression: 360,000 transactions per hour (6,000 per minute).
- WAN file transfer (2400 baud line; 128-character packet size, XMODEM file transfer; compressed file): 7,200 transactions per hour.
- File expansion: 540,000 transactions per hour (9,000 per minute).
- EDI-to-DLSS translation: 9,000 transactions per hour.

While the prototype system has been running, most of the DLSS transactions have been single card-image transactions. Multiple card-image transactions of only about 30 images were downloaded from the various hosts during the test, but DLSS card-image transactions on magnetic tape of up to 500 cards have been run manually through the prototype LGN. For this reason, throughput rates based on number of card images processed may be a more useful measure of performance than those

[1]Transaction processing rates for the Phase III LGN are considerably burdened by the multi-tasking software used in the prototype model in an MS-DOS environment. Phase I, stand-alone LGN speeds (without multi-tasking) were 21,000 transactions per hour (approximately 6 per second) for the slowest, most difficult DLSS-to-EDI translation. EDI-to-DLSS translations were even faster. Rates for both Phase I and Phase III LGNs reflect high levels of input/output (I/O) accesses per transaction. This is uncharacteristic of a production LGN, whose I/O activity should be tuned to a minimal level. Furthermore, in a production version, multi-tasking would not be based on a cycle-stealing process.

based on number of transactions processed. The prototype LGN throughput rates for card images processed are approximately 10 percent greater than those for transactions.

All processing times, including filtering, compression and expansion, and file transfer, are based on a Compaq 386/20 supporting a DESQview multitasking environment with approximately 12 tasks running concurrently. The overhead of this concurrency significantly degrades the throughput rates as opposed to those experienced in a stand-alone (single-tasking) environment.

LOCAL AND REMOTE OPERATIONS AND MAINTENANCE

Tables 3-1 and 3-2 list the various prototype LGN operations and maintenance tasks, respectively, and the extent to which each task can be performed remotely from the CLGN. All of the tasks can be performed locally.

A procedure developed for the prototype LGN, for synchronizing an LGN's clock with the CLGN, using Greenwich Mean Time (GMT) as a basis, has not been made operational. This inactive implementation is described in the WAN interface subsystem design alternatives in Chapter 9.

LOGGING ACTIVITY

Each LGN module records to a log every event of significance to the module, by time and date, as the event occurs. A separate log file is used for each module. While this approach makes it tedious to look at a historical window of the LGN as a whole, it makes it easier to examine the processing flow of a particular module. All analysis for the prototype LGN, including constructing ad hoc audit trails, is done off line.

TABLE 3-1

OPERATIONS TASKS

Operations task	Comments
System shutdown	Can be performed remotely.
System warm boot	Can only be performed locally; in some cases, particularly if the LGN is outfitted with an X.25 communications controller, it may be necessary to power down the LGN and then do a cold boot.
System restart	Can be performed remotely as a special system shutdown/system restart option; this differs from system warm boot, which starts the system from scratch, independently. Note: this operation cannot be executed on an LGN having an internal communications controller or other hardware that requires reinitialization at system start up.
Subsystem shutdown	Can be performed remotely.
Subsystem restart (after subsystem shutdown)	Can be performed remotely. Because of time constraints, this capability was not included for all prototype LGN subsystems.
Subsystem suspend/subsystem restart	Can be performed remotely.
Performance monitoring	Can be performed remotely by indirect means, e.g., requesting log files.
Disk maintenance tasks	Can be performed remotely by executing DOS commands on the remote LGN.

TABLE 3-2

MAINTENANCE TASKS

Maintenance task	Comments
Executable program update	Can be performed remotely via system shut-down/restart operations task for LGNs without internal hardware boards that require reinitialization at system start up. Note: this procedure has not been fully tested in the prototype LGN.
Translation table updating	Fully automated from CLGN, including module suspend and restart. As implemented in the prototype, this task operates on a file basis only, not on single records.
Examining create dates of files and other file characteristics	Can be performed remotely by executing a remote DOS command whose output is redirected to a file on the LGN and, then, requesting that file from the CLGN.

CHAPTER 4

BASIC DESIGN PRINCIPLES

HARDWARE/SOFTWARE PLATFORM

The prototype LGN runs on a Compaq 386/20 or 386/20e microprocessor. The 386/20 has 5 megabytes (MB) of main memory and a 60-MB hard disk; the 386/20e has 6 MB of main memory and a 40-MB hard disk. Each LGN has a dot-matrix printer attached, but the printer is used only on an ad hoc basis; no LGN operations depend on the printer's working reliably.[1] LGNs directly connected to the WAN are outfitted with an AdCom2-I X.25 communications board theoretically capable of managing 128 simultaneous virtual X.25 connections (however, in the prototype LGN environment, it manages only 1). LGNs with an IBM mainframe local host have an IRMA-2 3278/3279 terminal emulation board. All LGNs have a 2400-baud modem. LGNs with only a dial-up connection to the WAN use the modem as the sole means for interfacing with the WAN; LGNs with a direct connection to the WAN use the modem as a back-up communications device.

The CLGNs at LMI are connected to a Plantronics Micro Turbo Packet Assembler/Disassembler (PAD) connected to a General DataComm 4800-baud modem and a Rally Data Race 9600-baud modem. Each modem serves as a gateway for an addressable (direct) WAN line.

DESQview is the operating environment in which the LGN runs. DESQview provides multi-tasking by exploiting the 80386 chip's ability to simulate multiple virtual machines. Although the DESQview environment is not as robust as that of UNIX, virtually any DOS application can be run, unmodified, in the DESQview environment. This feature makes it possible to take advantage of the unequaled number and diversity of DOS-based development tools and products. American National Standards Institute (ANSI) C is the primary application language of the

[1]While relying on a printer for a console log (paper is a very reliable storage medium) is attractive, printers are too undependable for any essential processing functions. They jam, they run out of paper, and they encounter mechanical problems. Unless the systems using them explicitly recognize and deal with trouble in the printer connection (most software does not), a printer can render the system inoperational.

prototype LGN. Table 4-1 lists the complete array of software used in developing and operating the LGN.

TABLE 4-1

LGN DEVELOPMENT SOFTWARE

Software	Function
MS-DOS v3.3	Operating system
DESQview 386 v2.2	Multitasking environment and 386 memory manager
PolyAWK v1.3 (AWK programming language interpreter)	DLSS transaction filter
SuperSort v1.6	Used in filtering of multiple card-image transactions
CrossTalk Mk. 4 v1.02a	Used at most sites to automate a host session
Frontier Technologies Corporation (FTC) Super-X.25 run-time software v4.21	Used to initialize FTC AdCom2-I communication controller board. Used only at sites directly connected to the WAN.
PKARC/PKXARC	File compressor/expander
The following software is used for LGN software development only:	
Borland Turbo C v2.0	Primary application development language
DESQview API C library	C interface to DESQview
Greenleaf Comm Library	C-callable routines for communicating with the WAN
YACC and LEX	Parser and lexical analyzer for translation rules
AccSys function library	API functions for Paradox
Paradox 3	Data base management system for translation tables
FTC Super X.25 development software v4.21	C-callable routines and data for interfacing with the FTC AdCom2-I board

Note: API = Applications Programming Interface.

MODULAR DESIGN

The LGN is divided analytically into separate processing areas called subsystems; each consists of one or more modules. A module, which is the primary functional unit of the LGN, is made up of a number of processes. A process, which

roughly corresponds to an executable program, performs a more specialized function in support of the module to which it belongs. A process is perceived by the DESQview multi-tasking environment as the basic operational unit.

Each module is defined in terms of its processing and its interfaces with other modules. Generally, interfaces are defined as interprocess communications (IPCs). In theory, since the modules are closed except for their interfaces, each can be developed in any language that can (1) execute in the overall operating environment and (2) interface with the IPC facility. In practice, it is impractical or impossible for any language other than C to interface with the DESQview IPC. However, through the use of additional C-utility programs, it is possible to use any other language as a main programming platform. Accordingly, in deciding which programming language to use for each module, we sought a balance between applicability to the task and ease in interfacing with the DESQview IPC. In general, higher level, fourth-generation languages have been used where possible to reduce development time. An example is the use of AWK in the filter module.

Generally, a module is implemented as one or more C-language processes, each using the DESQview Application Programming Interface (API). Each process waits for events indicating that action is required. An event is defined as one of several possible occurrences, listed here in order of their priority:

- Input from the local keyboard
- Message from the operations subsystem
- Expiration of a software-based timer
- Message from any other subsystem
- Communications signal from the local host or the WAN.

After making an appropriate response to an event, the process awaits the next event. This loop continues until stopped by a message from the operations subsystem to shut down processing.

Those subsystems in which a language other than C is appropriate are handled as follows: A C-language event manager process runs as described above. When an

event is detected for an action involving a non-C process, the C-process executes a batch file process made up of at least these two components:

- One or more non-C programs
- A trailer C-program that sends a "process complete" message to the event-manager C-process.

In a batch process, each program is run sequentially; thus, the trailer C-program is executed automatically right after the last non-C program. In this way, a non-C process communicates indirectly with the main event manager process, by using the trailer C-program as a messenger. The batch process can also contain header C-programs, which are run prior to the non-C programs and perform additional IPC-related tasks. The batch process terminates after its last program is run.

For example, the AWK programming language is used for the input filtering process. The C event-handling program creates a batch process to run AWK and accompanying C-programs each time a file requires filtering.

All processes in the LGN are essentially peers of one another. Thus, the non-C process described, although invoked by another process, is an independent entity, as opposed to being a child process completely subordinate to the process that created it.

A significant benefit of a highly modular approach is that each module can employ a different software developer. Provided the interfaces between modules are well defined, the developers can code to the interface specifications, taking preferential liberties in programming without affecting the overall system. This strategy puts the largest share of responsibility on the interface design, since it will affect the whole system, its development, and its operation. During the development of the prototype LGN, the best results were achieved by establishing overall structures and practices that defined the IPC interface, and then assigning complete subsystems to individual developers. The benefits of a common programming style and approach were gained from a cohesiveness within subsystem modules, whereas adherence to the IPC interface rules was the chief factor in the successful integration of the disparate subsystems.

PARAMETERIZED SUBSYSTEM START-UP

A number of parameters are used to tune an LGN to its particular environment. These generally relate to the frequency of communication with the host and the WAN. Other parameters specify file and naming conventions, time-out periods, site-specific processes (generally called by the local interface subsystem), communication settings, and overall LGN characteristics. Each module has a separate bootstrap process and its own configuration file (parameter table) containing parameter settings. In addition, there is a configuration file of system-wide parameters read by all modules in the LGN. Maintenance of the parameter files is achieved by remotely controlled file replacement.

Parallel Processing

With few exceptions, all modules in the prototype LGN run in parallel, i.e., concurrently. However, since the system processes one file of transactions at a time, the actual degree of parallelism is usually low at any one time. Nevertheless, there are exceptions: for example, an LGN can be translating transactions in the DLSS-to-EDI direction while it is processing transactions in the EDI-to-DLSS direction. Likewise, a pipe-lining effect occurs if two transaction files from the host download in rapid succession. In this case, the first file might be translated while the second file is being filtered. However, if the filter finishes before the translator, the second file will not begin translation until the first file is finished.

For the prototype model, all processes run at equal priority; no attempt is made to optimize processing in this respect. In the production LGN, a design objective should be (1) to accommodate priority traffic and (2) otherwise to minimize throughput bottlenecks by adjusting module processing priorities during run time.

There are critical instances in which a process requires exclusive use of the LGN. At such times, all other processing is suspended. For example, when a mailbox (used for IPC messages) is to be created, a check is made to see whether or not it already exists; if it does not, one is created. To ensure that another process does not create the same mailbox during the time between checking and creation, all other processes are put on hold. As soon as the mailbox is created, the suspension is lifted, and LGN processing resumes normally.

Interprocess Communication

Each process in the LGN communicates via IPC, using the DESQview API facilities. DESQview uses a mailbox protocol for IPC: each process has mailboxes into which other processes put messages. A process needs only the name of the mailbox to send a message; once it is sent, the sender can forget it. This asynchronous approach works well for the peer-to-peer relationship among processes. For the prototype, acknowledgment is seldom required. However, a production system will require (1) more checks to ensure that the receiving process receives the message intact and (2) logic specifying a number of retries and required actions if unsuccessful. The present design does not preclude an ordered message dialog between two processes; it is just not performed automatically as part of the DESQview IPC.

Synchronization in the prototype model is emulated in two ways. First, an IPC semaphore is used to synchronize certain processes; in this way, one process can infer whether or not another is active by the availability or nonavailability of the semaphore.

Second, certain messages signal that a requested process has been completed. For example, Process 1 can send a message to Process 2, requesting performance of a certain task, and then wait a designated period to receive notification of task completion and error status from Process 2.

Interprocess messages are structured with a header followed by the message. Exceptions are (1) simple messages in which the header contains all the information necessary and (2) others, unique to each message type, that use one or more additional information fields. The message header is constructed as follows:

```
typedef struct modelshead

{
WORD wType;
char   cPriority;
char   sQuerymod[SY_MODMNEMLN+1];
char   sQuerylgn[SY_LGNNMLN+1];
word   wId;
} MODELSHEAD;
```

where:

- "wType" is a non-negative integer specifying the message type. Each message type has a unique number and an associated mnemonic identifying it. For example, the mnemonic for the message to initiate a filter is FILTERDLSS (message type 201).
- "cPriority" is the priority code of the message. E stands for expedited, L stands for low, and S stands for system message. System messages are ones sent by the system boot module; they have the highest priority. Low-priority (L) messages are not functional in the prototype LGN, except in the WAN interface module, where they are used for a specialized purpose.
- "sQuerymod" is the 2-letter name of the module sending the message. Each module in the LGN has a unique 2-letter identifier defined as a constant, along with a mnemonic name for the constant. For example, SY_D2EMNEM equals "DE", which is the 2-letter code for the DLSS-to-EDI translator module.
- "sQuerylgn" is the 3-letter identifier of the LGN sending the message. Normally, this is the same as the LGN on which the message is received. However, in some cases, the message originates from another LGN. The chief example of this is CLGN-to-LGN remote commands.
- "wId" is a non-negative integer that serves as an additional qualifier of the message. It is primarily used for sequencing messages, which is to say that it is not used often in the prototype LGN.

The simplest messages contain the message header and no additional fields. For instance, the SUSPEND_READY message consists solely of a message header, with wId equal to SUSPEND_READY (48). Most messages, however, have additional fields built into their structure. For example, the STARTED message contains one additional field, tTime_stamp, as illustrated:

```
typedef struct msg_started

{
MODELSHEAD  mMsg_head;      /*MODELS message header*/
time_t      tTime_stamp;    /*time of day*/
} MSG_STARTED;
```

A design that relies on the concept of messages sent between modules and action based on the message is called object-oriented. The key to an object-oriented design is defining the objects (modules) in terms of message handling. A complete listing of message structures, defined in the prototype LGN from the perspective of a

receiving module and the processing that the message triggers, is included in the MODELS LGN system specification and is a prime component of the system's detailed design.

Priority Management

Messages are one class of events that are acted upon or rejected by a module. A module becomes aware of an event by monitoring its object queue, through which all events are funneled. (The actual message is not sent to the object queue but, rather, a marker pointing to the event.) For mail message events, each module has one mailbox set aside to receive system messages from the system boot module. Messages sent to this mailbox are handled before messages received in any other mailbox; messages concerning critical actions such as suspensions or shutdowns are received here.

Peer Relationships Among LGNs

In initiating communications, all LGNs are peers; each can initiate a connection with another at any time. As implemented in the test system, two restrictions are placed on this prerogative. First, remote LGNs can initiate a connection only with the CLGN, not with other LGNs. Second, a connection between the CLGN and a dial-up LGN has to be initiated by the LGN; there is no outbound dialing from the WAN. This raises a question of how two dial-up LGNs (in the production system) might communicate. Two possible solutions might use (1) a WAN out-dialing capability, if available, or (2) the CLGN as a store-and-forward repository into which LGNs poll periodically.

Remote Operations

The CLGN can send any message to a remote LGN, by embedding it inside a SENDMSG message sent to the WAN interface module of the receiving LGN. When the SENDMSG outer layer is stripped off, the embedded message is sent to the appropriate module. This procedure applies for responses the remote LGN sends back to the CLGN.

One type of remote message sent by the CLGN is the utility request. When an LGN receives a utility request, it spawns a temporary process to execute the DOS command specified in the message. In most cases, the DOS command includes redirection of output to a file. The CLGN can subsequently request this file so that

file creation dates and other directory information about an LGN can be analyzed remotely.

CHAPTER 5

PROCESSING ENVIRONMENT

All software and data files required for operating the prototype LGN are located on the LGN's C-drive hard disk. An A-drive (floppy-disk drive) is required for manual file updates and emergency boots. Forty megabytes of hard-disk storage is the minimum required for smooth operation of the LGN; having any less capacity greatly increases the chances of running out of disk space during processing.

Appendix A contains a listing of the contents of the prototype LGN disk directories. The more important ones are covered here. The \CONFIG directory contains the configuration files for all modules. Most of the configuration files pertain to a particular module and are named by a concatenation of a two-letter module mnemonic with a .CFG extension. Others relate to a subsystem or to the entire LGN. All configuration files are American Standard Code for Information Interchange (ASCII) text files with entries of the form

<param> = <value>.

Although the software can interpret <value> as a list of values, in the prototype implementation, <value> is single-valued.

By convention, two-letter mnemonics are used to identify the top-level directories. For example, the local interface module is under directory \LI. The system boot module determines the directory for a module's executable programs by reading from its corresponding module configuration file the optional OP_DRIVE and OP_DIRECTORY parameters. If neither of those parameters is found, the drive and directory default to the ones listed in the program's Program Information Files (PIFs) (see Chapter 6).

The organization of subdirectories under the main module directories is listed with each module in Appendix A. In general, application programs reside in a subdirectory called BIN. Beyond that, subdirectory naming conventions are specific to the module to which they pertain.

No random access memory (RAM) disk is used in the prototype LGN, since all available memory is used for running programs. Any temporary files are written to the C-drive in specified directories. In a production model, the single maneuver most likely to affect translation throughput speed would be a maximum use of virtual storage.

CHAPTER 6

NAMING CONVENTIONS

With few exceptions, names for configuration parameters, application programs, and callable application functions follow accepted coding conventions. This chapter lists the naming conventions used for each type of named entity in the system.

CONFIGURATION PARAMETERS

In general, configuration parameters are of the form

<module>_<name>

where <module> is a two-letter module mnemonic and <name> is a descriptive parameter name. Table 6-1 shows the module mnemonics.

TABLE 6-1

MODULE MNEMONICS

Mnemonic	Module	Mnemonic	Module
CL	CLGN polling	SB	System boot
D2E	DLSS-to-EDI translation	SM	System monitor
E2D	EDI-to-DLSS translation	SY	Systemwide (all modules)
LD	LAN dequeuing	WD	WAN dequeuing
LI	Local interface	WI	WAN interface
OP	Operations (subsystem)	WQ	WAN queuing

PROGRAM NAMES

Executable and batch programs in the prototype were named at the discretion of their individual developers. Generally, all programs belonging to a particular

module start with the two-letter module mnemonic. This is not so, of course, for off-the-shelf programs incorporated in a module.

MESSAGES

Message mnemonics are all named (via #define) constants that convey the message content but follow no convention other than the standard C-language practice of naming constants via upper-case letters. Message structures (see the Interprocess Communication section of Chapter 4) are formed by a concatenation of MSG__ and the message mnemonic.

C-PROGRAM IDENTIFIERS

For C-program identifiers, Hungarian notation[1] is used. It allows programmers to overcome the lax enforcement of data typing rules in the C-language. Table 6-2 shows the standard set of Hungarian prefixes as implemented in the prototype environment. Table 6-3 shows prefixes to the prefixes that further describe the identifier.

Several exceptions to Hungarian notation naming rules exist in the prototype system. Some frequently used structures are assigned their own Hungarian prefixes. Function name identifiers and constants do not use Hungarian notation. Furthermore, much of the translation program code pre-dates Phase III, so a slightly different Hungarian notation is used in that area.

C-PROGRAM NAMED CONSTANTS

Names of constants follow the standard C-language practice of using all upper case. Constants defined in include files, which are accessible by all subsystems, begin with an appropriate two-letter module mnemonic. Constants whose domain is only one subsystem do not follow any prescribed naming convention.

---

[1]Hungarian notation is an increasingly popular naming methodology that prefixes every identifier with letters suggesting the identifier's type. In a Hungarian notation variable, all letters up to the first capital letter are part of the Hungarian prefix.

TABLE 6-2

HUNGARIAN NOTATION TYPE PREFIXES

Notation	Type
ac	non-AsciiZ string (array of characters)
b	BOOL (int)
by	BYTE (unsigned char)
c	char
dp	DATA_PTR (void *)
h	DV_API_HANDLE (unsigned long)
i	int
l	long
m	message structure
s	AsciiZ string (char *)
st	structure
t	time_t (defined as long in Turbo C)
v	void or variable argument list
w	WORD (unsigned int)

TABLE 6-3

ADDITIONAL HUNGARIAN TYPE PREFIXES

Notation	Type
a	array
aa	two-dimensional array
g	global
p	pointer
u	unsigned
x	static

C-APPLICATION FUNCTIONS

While systemwide functions are usually named su__<descriptive-name>, there are a few exceptions: for instance, names for functions contained within a module are up to the programmer's discretion.

PROGRAM INFORMATION FILES

All DESQview processes are invoked via a program information file (PIF) that must be of the form <2-char-id>-PIF.DVP. In the prototype LGN, <2-char-id> is the two-letter module identifier for main module processes; otherwise, it is a descriptive identifier of the process. For example, the main local interface process PIF name is LI-PIF.DVP; the download script process PIF name is DS-PIF.DVP.

DESQVIEW MAILBOXES

Two mailboxes are used consistently by LGN modules: the system mailbox and the expedited mailbox. Modules use the default DESQview mailbox, which is unnamed, as the system mailbox. The expedited mailbox has the value "<module>__E". A suite of named constants is declared and set to each expedited mailbox value. Other mailboxes, including temporary intramodule mailboxes, do not follow a naming convention.

CHAPTER 7

PROCESSING SUBSYSTEMS AND MODULES

The design of the LGN divides processing into four subsystems:

- Local interface subsystem
- WAN interface subsystem
- Transaction processing subsystem
- Operations subsystem.

These are further broken down into several major modules, each consisting of one or more processes. Chapters 8 through 11 describe these major subsystems in terms of their modules.

THE THREE STATES OF MODULE PROCESSING

A module is always in one of three possible states: the Initialization State, the Operational State, or the Shutdown State. The LGN itself can be considered to be in one of these three states. When the LGN starts up, it is in the Initialization State, as are each of its modules. When every module has initialized and confirmed that fact to the system boot (SB) module, the SB module instructs all others to enter the Operational State. Most of the time is spent in this Operational State, in which all transactions are processed and all host and WAN communications take place. The LGN enters the Shutdown State as the result of a command or a fatal error. Normally, the Shutdown State ends with termination of LGN execution, requiring the LGN to be rebooted to the Initialization State. There is a special shutdown/restart message that enables shutdown to be immediately followed by restart, i.e., a return to the Initialization State. It is possible for an individual module to be in the Shutdown or Initialization State while the rest of the LGN is in the Operational State.[1]

[1]This occurs when a module receives a SHUTDOWN message or a RESTART message (following a SUSPEND or SHUTDOWN message).

COMMON MODULE PROCEDURES

A module boots when the SB (see Chapter 11) creates a process and invokes a canonically named PIF:

<PIF-Drive>:\<PIF-Directory>\<ModMnem>-PIF.DVP

The PIF, in turn, starts the module's main executable program. The <PIF-Drive> and <PIF-Directory> are retrieved from the LGN.CFG file. The file BOOT.TAB lists modules to boot; each module in the list is referenced by its two-letter mnemonic (<ModMnem>). Each module first performs common and module-specific initialization procedures and then sends SB a STARTUP message, indicating it is ready to proceed into the Operational State, or a CANTSTART message, indicating a failure during module initialization.

As the SB receives a STARTUP or CANTSTART message from each module, it displays the status in its window on the local console.

Figure 7-1 shows pseudocode[2] for the standard module start-up procedure making up the bulk of the Initialization State. Except for SB, this same approach is used by each module in the system. SB follows a different path during the LGN Initialization State, functioning as a central clearinghouse for notification messages. The SB module, before invoking other modules, creates mailboxes to be used by them; this procedure allows SB to retain ownership of the mailboxes if a module is shut down. Individual module start-up procedure opens the mailboxes, reads values for parameters, and performs all other initialization necessary to proceed to the Operational State. A STARTUP message sent to SB indicates successful completion of initialization; a CANTSTART message indicates an error, and SB triggers a transition of the LGN to the Shutdown State. After the STARTUP message is sent, the start-up procedure waits SY_WAIT4GO seconds for an OPGO message from SB, which means that all modules have completed initialization and that the LGN can advance to the Operational State. For any message other than OPGO, or if SY_WAIT4GO seconds elapse, the module sends a SHUTDOWN_READY message to SB, triggering the transition to the Shutdown State.

---

[2]Pseudocode, also known as Structured English, uses a small subset of English words in conjunction with symbolic names representing abstract entities, to describe the sequence of events making up a process. It is more precise than a flowchart but is less rigorously structured than a format programming language.

```
/* Psuedocode for module Initialization State processing */

BEGIN
    OPEN mailboxes previously created by System Boot
    READ LGN parameters
    READ module parameters
    INITIALIZE resources
    IF ERROR during INITIALIZE
       SEND CANTSTART message to System Boot
       PROCEED to Shutdown State
    ENDIF
    SEND STARTUP message to System Boot
    CREATE time and set for SY_WAIT4GO seconds

    WAIT for EVENT from Event-Queue
    IF EVENT = OPGO message
       PROCEED to Operational State
    ELSE
       PROCEED to Shutdown State
    ENDIF
END
```

FIG. 7-1. MODULE INITIALIZATION STATE PROCESSING

Figure 7-2 shows pseudocode describing a module's Operational State, in which modules spend most of their processing time. Upon entering the Operational State, each module goes into a loop, (1) awaiting events, (2) acting on events it recognizes, and (3) treating as an error those it does not. Specifically, the module waits for the types of messages it serves or for any of the special messages: SUSPEND, RESTART, or SHUTDOWN. The processing of messages, other than special messages, is module-specific, although consistency is strived for through use of common message-handling procedures. Special messages are handled more uniformly, except that each module's SHUTDOWN and RESTART activities differ. The module waits in this loop until receiving a SHUTDOWN message, which initiates transition to the Shutdown State.

```
./* Psuedocode for module Operational State processing */

BEGIN
    suspended = FALSE
    DO FOREVER
      WAIT for EVENT from Event-Queue
      IF EVENT = SHUTDOWN message
        PROCEED to Shutdown State
      ENDIF

      IF suspended = TRUE
        IF EVENT = RESTART message
          READ LGN parameters
          READ module parameters
          RE-INITIALIZE resources
          MOVE (Suspended) EVENTS from Holding-Queue to Event Queue
          suspended = FALSE
        ELSE
          MOVE EVENT to Holding-Queue
        ENDIF

      ELSE                          /* not suspended */
        DO CASE (EVENT)
        CASE SUSPEND message
          SEND SUSPEND-READY message to System Boot
          suspended = TRUE
        CASE expected message
          Act on message
        OTHERWISE
          LOG ERR
        END CASE
      ENDIF
    END DO
END
```

FIG. 7-2. MODULE OPERATIONAL STATE PROCESSING

The LGN and its component modules enter the Shutdown State by one of five events:

- A module sends a CANTSTART message to SB during the Initialization State.
- A module fails to send a STARTUP message to SB in the time allotted during the Initialization State.
- A module sends a CANTSTART message to SB during its re-initialization following reception of a RESTART message from SB.

- The CLGN sends an LGN_SHUTDOWN message to SB remotely, via the WAN.
- An LGN_SHUTDOWN message is sent to SB from the local keyboard.

A need is recognized for additional means, not implemented in the prototype system, of entering the Shutdown State, as part of the error procedure for certain fatal errors. More practicable methods for entering the Shutdown State in response to a nonrecoverable LGN error should be built into the operational system.

As noted above, the transition to the Shutdown State is always through the SB module. Figure 7-3 displays the pseudocode for the common module logic for the Shutdown State.

```
/* Psuedocode for module Shutdown State processing */

/* Entered upon receiving a MOD_SHUTDOWN message from System Boot */

BEGIN
   SEND SHUTDOWN_READY message to System Boot
   CLOSE all mailboxes
   KILL any temporary mailboxes (not created by System Boot)
   FREE other resources as needed
   QUIT
END
```

FIG. 7-3. MODULE SHUTDOWN STATE PROCESSING

HOW A DESQVIEW PROCESS IS INVOKED IN THE PROTOTYPE LGN

Although the SB module is the module chiefly responsible for starting up DESQview processes, other modules such as the local interface module and the local area network (LAN) dequeuing module perform this task as well. Therefore, a general discussion of the mechanics of starting up a DESQview process is presented here.

Each DESQview process has an associated PIF containing information needed by DESQview to execute the process properly. The parent process (the one starting a new process) reads this PIF into its own memory. It can then edit selected fields of the PIF image in memory. For example, the PIF field specifying the directory where the process is located can be overridden by the OP_DIRECTORY parameter, which is optionally included in every module's configuration parameter file. Some LGN

processes use a simplified PIF structure, shown in Figure 7-4, that clumps together ignored fields; other processes simply refer to PIF fields as an offset within a buffer. The address of the PIF image is passed as an argument to the DESQview API function app_start, which starts the process designated in the PIF.

How does the first DESQview process get started? In the prototype LGN, this is accomplished via DESQview's primitive scripting facility, which allows a process to be invoked automatically upon starting DESQview from DOS, in which case that is the main process of the SB module.

```
typedef  struct pif                          /* PIF (.DVP) structure          */
{
 char     acFiller1 [2];                     /* not used by LGN               */
 char     acPgmtitle [32-2];                 /* descriptive program title     */
 char     acFiller2 [36-32];                 /* not used by LGN               */
 char     sPgmstartcmd [100-36];             /* command to start program      */
 char     cDrive;                            /* default drive (e.g., C)       */
 char     sDirname [165-101];                /* default directory name        */
 char     sPgmparams [229-165];              /* program parameters            */
 char     acFiller3 [233-229];               /* not used by LGN               */
 BYTE     byNlogrows;                        /* # rows in logical window      */
 BYTE     byNlogcols;                        /* # cols. in logical window     */
 BYTE     byRow;                             /* initial row location          */
 BYTE     byCol;                             /* initial column location       */
 int      iSysmem;                           /* system memory size (KB)       */
 char     acFiller4 [367-239];               /* not used by LGN               */
 BYTE     byCtrlbyte1;                       /* bit-mapped                    */
 BYTE     byCtrlbyte2;                       /* bit-mapped                    */
 char     acKeys [2];                        /* keys to start from menu       */
 char     acFiller5 [380-371];               /* not used by LGN               */
 TBOOL    cClose_on_exit;                    /* auto-close on exit? (T/F)     */
 char     acFiller6 [384-381];               /* not used by LGN               */
 BYTE     byNphysrows;                       /* # rows in physical window     */
 BYTE     byNphyscols;                       /* # cols. in physical window    */
 char     acFiller7 [388-386];               /* not used by LGN               */
 BYTE     byCtrlbyte3;                       /* bit-mapped                    */
 char     acFiller8 [SY_PIFLN-389 + 1];      /* fill to end                   */
} PIF;
```

FIG. 7-4. PIF STRUCTURE

CHAPTER 8

LOCAL INTERFACE SUBSYSTEM

The local interface subsystem is responsible for transferring all transactions between the host and the LGN. One of the prototype test goals is to make use of existing host-to-terminal file transfer capabilities, rather than modifying the host. Thus, a flexible design was called for that could segregate site-dependent communication processes with only minimum impact on the rest of the local interface subsystem. This segregation is accomplished in the prototype LGN via a separate download script invoked by the main local interface process. The download script is a batch process that, in turn, calls a site-specific program to maintain the host connection optionally and perform the data transfer. Figure 8-1 displays the relationships among the components of the local interface subsystem. Parameters read at boot time specify site details, including download scheduling information and the site-dependent program to be invoked. The communications interfaces handled by the prototype system are

- 3270 terminal emulation
- Asynchronous teletype and terminal emulation.

LOCAL INTERFACE MODULE

Purpose and Description

The local interface module oversees the transfer of all data between the host and the LGN. Since the actual connection to the host is maintained by programs called by the download script process, the main local interface process is host-independent.

The module attempts to initiate a download from the host during "download windows," which are simply time periods. There may be one or more download windows. Within each download window, the host is polled at intervals until a download is completed successfully or the end of the download window is reached.

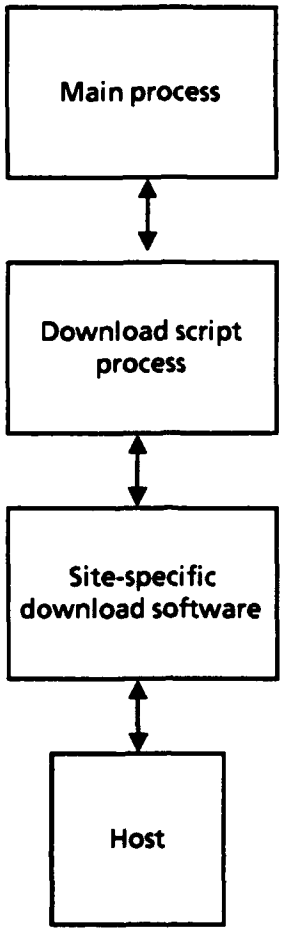


FIG. 8-1. LOCAL INTERFACE SUBSYSTEM

The values for the download windows and polling interval are read during subsystem start up.

Start-Up Procedures

The local interface module performs the standard module start-up procedure, plus additional steps related to invoking the local interface subsystem's host-dependent process. As part of the start-up procedure, the module parameters listed in Table 8-1 are read in. Additional parameters concerning uploading and multiple physical and logical host connections were defined but are not used in the prototype system.

Another local interface module start-up task is reading the download window parameters. The download window parameters are kept in a separate file called

TABLE 8-1

LOCAL INTERFACE START-UP PARAMETERS

Parameter	Description
LI_SESSIONINTERVAL	Number of seconds between host polls
LI_DNLDMNEM	Mnemonic used in determining PIF that executes the download script process
LI_DNLDSCRIPT	Name and optional parameters of site-dependent program invoked by the download script process
LI_DNLDTIMEOUT	Number of seconds to wait before receiving a "completed" message from the download script process

DLWINDOW.DAT and are read in during start up and after each host session as well. The DLWINDOW.DAT file contains one line for each window; each line consists of the following fields:

- Window start time
- Window end time
- Last download date (YYMM)
- Last download time (HHMM)
- LGN target file name
- Host source file name.

The local interface module uses four mailboxes:

- Default DESQview mailbox
- Expedited mailbox SY_LANINTMXEXP
- Low-priority mailbox SY_LANINTMXLOW (inactive in the prototype system)
- A holding or deferred mailbox used for storing certain messages to be acted on later.

The default mailbox is used to receive messages from SB. The expedited mailbox is used to receive messages from all other modules. The local interface module receives no expedited messages from any outside module, only from its component processes.

In the prototype, the low-priority mailbox is not used. The deferred mailbox provides a temporary holding facility when the module is unable to service its expedited mailbox – during downloads, for example.

If 3270 or asynchronous terminal emulation is involved, Terminate-and-Stay-Resident (TSR) software is installed when the LGN is booted, either from the CONFIG.SYS file or the AUTOEXEC.BAT file; TSR software controlling a device must be loaded before DESQview is invoked.

A timer, created as part of the initialization process, is used to keep track of the time elapsed since a download script process was started. During the Operational State, if enough time (determined by a parameter) accumulates since the last download process, the local interface process assumes an error has occurred that has caused the host connection software to hang.

Processing

Frequency of Operation

The main event-handling process in the local interface module is always running. Mostly, it is parked on the object queue awaiting a timer[1] elapse, which indicates that it is time to attempt a host download. The download script process is invoked whenever a download attempt is made. Usually, the download script process oversees the entire session with the host and quits after the download attempt. The download script process, a batch file, calls other programs serially. These programs are not DESQview processes and do not run continuously.

Flow of Processing

The frequency of host download polls is governed by a timer. If the local interface module is not in a download window, the timer is set to the start time of the next window. If the module is in a download window, the timer is set to expire after LI_SESSIONINTERVAL seconds, which is the delay time between polls. Figure 8-2 shows the logic used to determine whether or not the module is currently in a download window. This logic is performed right after start up and after every download attempt. Note that once a file is downloaded from the host, the module is

[1]A timer is a DESQview object that can be set to "expire," or send a time message, at a prescribed time of day or after a certain number of seconds.

no longer considered to be in the window in which the download was made. In other words, for a module to be in a download window, two conditions have to be met:

- The current time must fall between the download window's start and end time.
- A download must not already have occurred within this window today.

```
/* Psuedocode for determining if currently in a download window */

BEGIN
   next-start-time = 9999        /* impossibly high value */
   current-time = current time
   current-date = current date
   dnld-window-ct = # of download windows in configuration file
   in-window = 0
   i = 0

   DO WHILE i < dnld-window-ct
     AND in-window = 0
        i = i + 1
        last-dnld-date = date of last download in ith window
        window-start-time = start time of ith window
        window-end-time = end time of ith window
        IF current-time > = window-start-time
         AND current-time > =  window-end-time
         AND current-date NOT = last-dnld-date
             in-window = i
             next-start-time = window-start-time
        ELSE
            /* This is always true when i = 1: */
            IF window-start-time < next-start-time
               next-start-time = window-start-time
            ENDIF
         ENDIF
     END DO
END
```

FIG. 8-2. DOWNLOAD WINDOW DETERMINATION LOGIC

During each download poll, the local interface module invokes the download process, which in turn calls one or more site-dependent programs to perform the following functions:

- Establish a connection with the host (if the connection is not permanent).
- Log on to the host.

- Download all available designated files.
- Log off the host.
- Update file DLWINDOW.DAT.
- Disconnect from the host (if the connection is not permanent).

In most cases, CrossTalk Mk.4 is the program that performs all of these functions. The main module waits for LI_SESSIONINTERVAL seconds for a message from the download script process indicating that the download attempt is finished. If it does not receive this message in the time allotted, it assumes that a nonrecoverable error occurred during the host session; the timer is reset, and processing continues.

Because host downloads may take a relatively long time, an additional check should be made before initiating a host poll, to ensure that a download is not already in progress. Code for this function was not completed for the prototype system. For the prototype test, the situation was avoided simply by scheduling download windows far enough apart to ensure download completion before the start of the next download window.

The scripts controlling host sessions are located in the directory \LI\DNLD\SCRIPTS and are highly tailored[2] to each site. The production system should employ a sender-driven download design. But the following description of the flow of processing in the script program is applicable to almost all of the sites in the Phase III test, and the script program is necessarily described, as an integral part of the prototype LGN design.

Even though the script is invoked only if the module is currently in a download window, the script reads in the download window parameters from DLWINDOW.DAT. This reading is done primarily because a script language such as CrossTalk Mk.4 can more likely read a text file than handle command-line parameters. Therefore, to find out which download window is in effect, the script program needs to read the parameters from DLWINDOW.DAT.

After the script reads the download window parameters and performs other program initializations, it attempts to log on. If successful log-on is possible, an

[2]This was especially a challenge during the prototype test, since the host environment was subject to change without notice.

appropriate return code is passed; if not, the script logs off the host and returns control to the download script batch process.

Once logged on to the host, the script downloads the host file associated with the current download window (read in via DLWINDOW.DAT) to a uniquely named file in the LGN's \LI\DNLD\STAGING directory. The exact software and methodology used to download the file vary from site to site in the prototype environment. Whenever possible, file-transfer software with both a host and an LGN component working in tandem is used (e.g., FTTSO). Otherwise, an asynchronous transfer using XMODEM or Kermit is used.

After the download, the script checks for the existence of the downloaded file and the return code sent by the download script process. Usually this code is zero, since most communications packages' script languages cannot send a return code. Note that these checks do not guarantee that the download has been flawless; they prove only that something is there. At this point, the DLWINDOW.DAT file is updated to show that a file was downloaded during the window.

A RCVDDLSS message is sent to the LAN dequeuing module, indicating the name of the downloaded file. The file is uniquely named via the concatenation of a base-36 timestamp, a one-digit qualifier, and the extension .DLD. The timestamp represents the number of seconds since 00:00:00 GMT, 1 January 1970. Finally, the local interface module re-reads the download window parameters and resets the timer accordingly.

Data on all pertinent events are logged, including the values to which the timer is set, the number of download windows read, the current download window number, the name of the download script process, the start and end time of the download script process, and the size and create date of the downloaded file.

Data Structures

The PIF structure (Figure 7-4) is used to start the download file process. No other data structures of note are required by the module.

Differences Between LGN and CLGN Implementations

Although the production system's local interface subsystem is likely to be greatly different, in the prototype system, the CLGN can emulate all LGN functions. The CLGN's local interface module is virtually the same as the LGN's.

Shutdown Procedures

Upon receipt of a MOD_SHUTDOWN message, the standard procedures for module shutdown are followed and the module terminates.

Serialization

The local interface module must be running all the time. Since it is the module that initiates the DLSS-to-EDI flow through the LGN, it has no prerequisite process. On the other hand, the LAN dequeuing module can be serialized to execute after the local interface module; this process is described under the topic of Serialization in Chapter 10.

Files

The format of the host files differs from site to site; this factor is one of those involved in deciding on the appropriate file transfer software. The main local interface process, however, is not affected by the format of the host file. It passes host files along, transparently, to the LAN dequeuing module, which then has to filter out any extraneous header and/or communication data mixed in with the DLSS transactions.

Alternative Designs

The prototype LGN local interface subsystem is designed with a severe limitation: no modification of the host software is permitted. Below are design considerations for a production system free of this restriction:

- Employ sender-driven file transfers
- Couple local interface module and host operating environment more tightly (e.g., create direct access to host file from C-program or direct access to LGN IPC from the host).

CHAPTER 9

WAN INTERFACE SUBSYSTEM

X.25 MODULE

Purpose and Description

The X.25 or WAN interface module provides for either direct or dial-up service with the commercial WAN used during the prototype test, as shown in Figure 9-1. It is responsible for connecting with the WAN and maintaining a session with a remote LGN. All transactions and information exchanged between an LGN and the CLGN are managed by the WAN interface. During the test, remote LGNs have communicated with the CLGN only; direct LGN-to-LGN communication has not taken place. Unless otherwise noted, this restriction on remote LGN communications is implied for the rest of this section.

Start-Up Procedures

Because the communications link must be initialized before the WAN interface module can proceed to the Operational State, this module performs a number of tasks in addition to the usual start-up chores. Also, for LGNs that are directly connected to the WAN, initialization code for the AdCom2-I X.25 board is loaded beforehand in the AUTOEXEC.BAT file, as per DESQview's restrictions with device driver software.

IPC Start-Up Procedures

The module first opens its expedited and low-priority mailboxes (recall that these were created previously by the SB module). Actually, the low-priority mailbox is used for storing previously deferred messages,[1] rather than ones of low priority.

---

[1]After the expedited and low-priority mailboxes are opened, a deferred mailbox is created and opened; it is used as a holding area for messages involving unsuccessful WAN transmissions. During the Operational State, after an unsuccessful attempt to process a WAN transmission message, the message is retained in the deferred mailbox for ulDefertimeout seconds, and then transferred to the low-priority mailbox. When no expedited messages precede it in the queue, the message is processed again. If the WAN transmission is again unsuccessful, the message is placed back in the deferred mailbox and the cycle starts over. A message can be deferred and subsequently retried up to iDeferLimit times. ulDefertimeout and iDeferLimit are read in as configuration parameters.

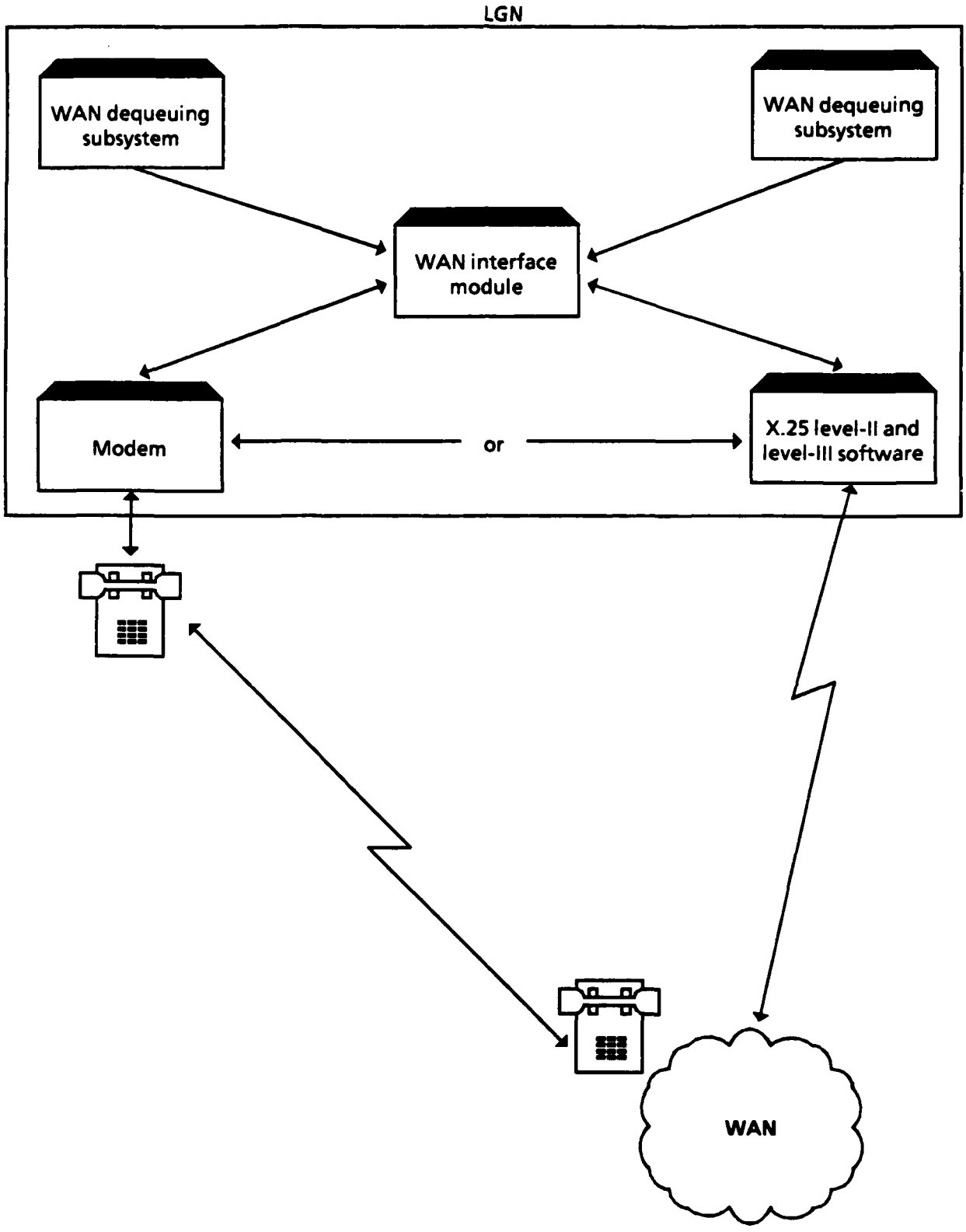


FIG. 9-1. WAN INTERFACE SUBSYSTEM

The WAN interface module creates four timers used for scheduling and synchronizing different tasks during the Operational State:

- *hCheckx25_timer.* The CheckX25 timer determines the time allowed between checks for status changes in the communications interface.
- *hCheckipc_timer.* The CheckIPC timer regulates the time spent between instances of polling the IPC object queue. It works in tandem with the CheckX25 timer. Note that the CheckX25 timer signals X.25 polling to end and IPC queue (event queue) monitoring to start, while the CheckIPC timer signals IPC queue monitoring to end and X.25 polling to start. The module spends most of its Operational State time checking alternately for IPC and communications events. This processing flow, along with the role played by the two timers, will be explained in more detail under the topic of Flow of Processing.
- *hDeferred_timer.* The Deferred timer expires every ulDefertimeout seconds, and all deferred messages are moved to the low-priority mailbox. As a consequence, they are back into the object queue and eligible once again for processing.
- *hIPCStat_timer.* The IPCStat timer expires every ulIPCStatinterval seconds (typically, the expiration interval is 24 hours). The timer's expiration signals the module to write an entry to the log and the screen summarizing IPC activity over the timer interval.

These timers are created during module initialization but are not set until the Operational State begins. The IPCStat timer is an exception; it is set during start up, enabling the messages exchanged as part of the Initialization State protocol to be included in the module's next IPC status report.

Once the mailboxes and timers are created, the configuration parameters are read in. Setting many of the WAN interface parameters is optional; they have default values. Table 9-1 shows the complete list of module-specific configuration parameters; where a default value is shown, entry is optional for the parameter in the WAN interface configuration file (WI.CFG). Two parameters included in the table are not specific to the WAN interface module but are very important to its operation. The SY_LGNCOMM parameter indicates whether the LGN is configured as a direct-connect site or a dial-up site. If the LGN is direct-connect, the SY_NETADDR parameter contains the LGN's network address.

TABLE 9-1

WAN INTERFACE BOOT PARAMETERS

Parameter	Description	Default
WI_CHECKX25QUEUE	Wait time between X.25 polls	5 seconds
WI_CHECKIPCQUEUE	Wait time between IPC checks	5 seconds
WI_DEFERTIMEOUT	Wait time between retrying messages	60 minutes
WI_DEFERLIMIT	Maximum number of retries	5
WI_FTCWINT	FTC AdCom 2-I software interrupt number	105 (0 x 69)
WI_DEFAULTDIR	Top-level directory under which subdirectories are found	
WY_LGNCOMM	Type of WAN interface (direct or dial-up)	
If the LGN is directly connected to the WAN:		
SY_NETADDR	Network address of the LGN	
If the LGN has a dial-up connection to the WAN:		
WI_NETPHONE	Network phone number	
WI_BAUD	Baud rate	2400
WI_PORT	LGN comm port number (typically 1 or 2)	
WI_PARITY	Parity ("E", "O", or "N")	"N"
WI_STOPBITS	Stop bits per character	1
WI_DATABITS	Data bits per character	8
WI_ATTEMPTS	Maximum number of times to try dialing the network	5

Note: FTC = Frontier Technologies Corporation.

Communications Start-Up Procedures

When the LGN is direct-connect configured, after IPC initialization is completed, the WAN interface module initializes the X.25 communications hardware and software and the X.25 level-II link and level-III link.

All direct WAN communications go through the AdCom2-I communications controller board. The module interfaces with the AdCom2-I board via API function calls and global data structures. Before invoking any AdCom2-I API functions, the module initializes a data structure that holds packet and frame parameters used by the AdCom2-I board during X.25 communications. The prototype LGN maintains parameter values for several network environments, including commercial WAN defaults, Defense Data Network (DDN), direct (AdCom2-I to AdCom2-I, bypassing the WAN), and loop-back (for testing). All environments except DDN have been encountered at some time during the Phase III test. In the prototype, only one of these conditions can be in effect at an LGN at any given time; only recompiling the WAN interface software can change the WAN environment setting. In a production LGN, this configuration management procedure must be more flexible. Table 9-2 shows the values of the more significant parameters for a default commercial WAN connection used during the test.

Once the X.25 parameters are set to starting values, the module initializes the X.25 level-II link and level-III link to the WAN, using the AdCom2-I API functions. The LGN uses only one line for an X.25 session; only one line (line 0), the first line available on the AdCom2-I board, is initialized.

Clock Synchronization

In a production LGN, calibration with the CLGN clock will need to be established before the LGN can proceed to the Operational State. In the prototype LGN, code was written to calibrate the LGN clock with the CLGN but was not activated. The clock synchronization algorithm coded is described later under the topic of Algorithms.

Processing

Frequency of Operation

The application software for the WAN interface module is always operating. If the LGN has a dial-up connection to the WAN, it remains parked in the IPC queue waiting for events; if directly connected, the module seesaws between the IPC queue and the X.25 line, monitoring each area for activity.

TABLE 9-2

SELECTED X.25 INITIALIZATION PARAMETERS

Parameter	Value	Meaning
Packet parameters:		
line	0	Line to which the initialization applies
term_type	1	Specifies DTE (1) or DCE (3)
ccittyear	0	Specifies CCITT 1980 (0) or 1984 (1)
passive	0	Send restart packet onlink initialization
modulo	1	Pending packet modulus
mbit	1	Allow m-bit use
mbitmode	5	Enable data splitting and re-assembling
qbit	0	Do not distinguish between X.25 control packets and data packets in packet header
dbit	0	Disable remote DTE acknowledgment of data packets
ddn	0	Disable DDN compatibility
ddnstand	0	Do not use DDN standard service on call setup
sdu	1024	Maximum data buffer size to be delivered to or from X.25
tx_max_packet_size	1024	Maximum packet size to receive
rx_max_packet_size	1024	Maximum packet size to send
tx_packet_size	128	Level 3 send packet size
rx_packet_size	128	Level 3 receive packet size
num_pvc	0	Number of permanent virtual circuits
timers and retry counters	a	CCITT and ISO timers T20 – T28
CCITT 1980 facilities used on a per-call basis:		
fcp_negotiate	1	Enable flow control parameter negotiation
Frame-level parameters:		
mx_frame_size	1028	Maximum size data passed to level 3
t1_time	60	Wait time before retransmitting a frame (value is in units of 50ms)
n2_cnt	3	Maximum number of retransmission attempts
frame_modulo	8	Frame sending modulus
frm_window_size	4	Maximum number of outstanding frames

[a] Values recommended in FTC *Super-X.25 DOS Programming Manual*, Version 4.1, Appendix A are used.

Flow of Processing

The main processing loop logic and the handling of messages by this module depend on whether the LGN's WAN configuration is direct or dial-up. For a direct-connect LGN, the module services DESQview events and incoming X.25 packets as well.

For DESQview IPC service, the module parks in the DESQview object queue until a message is received or a timer expires. If the CheckX25 timer expires, the module switches to the X.25 service and checks for incoming packets. If no packets are received, processing switches back to the IPC queue. Packets received are processed until time expires for the X.25 service interval; when the time is up, the module returns to monitoring the IPC queue. This loop continues indefinitely until a SHUTDOWN message arrives in the object queue. Figure 9-2 shows pseudo-code for the WAN interface module Operational State processing for a direct-connect LGN. The processing is the same for a dial-up LGN, except that the X.25 step is skipped.

A direct-connect LGN monitors the X.25 WAN connection directly by using Frontier Technology Corporation's (FTC's) Super-X.25 API function calls to communicate with the AdCom2-I hardware. When servicing the X.25 connection during the Operational State, the WAN interface module listens for a CALL REQUEST packet on line 0 (the X.25 line initialized during start up). If a CALL REQUEST packet is not detected within ulCheckIPC seconds, the module switches back to IPC processing.

When a CALL REQUEST packet is received over the WAN line within ulCheckIPC seconds, the module responds by sending a CALL ACCEPT packet. Embedded in the first DATA packet following the CALL REQUEST packet is a code indicating the type of message being sent by the remote LGN. The coding scheme used is termed the MODELS Network Protocol (MNP). It begins with a one-character identifier (in the first DATA packet) representing the action requested by the remote (calling) LGN; if required, supplemental data (e.g., a file name) are also included in the packet. Two MNP codes – A (Acknowledge) and N (Negative acknowledge) – sent by the called LGN provide an immediate response to an MNP. The MNP is summarized in Table 9-3. When two codes are used to indicate the same action, one refers to a more streamlined version of a request than does the other.

```
/*      Psuedocode for WAN interface module
        Operational State processing */

BEGIN
    X25-seconds     = 5
    IPC-seconds     = 5

    DO FOREVER
        Set CheckX25 timer to IPC-seconds
        check-X25 = FALSE
        DO WHILE check-X25 = FALSE
            WAIT for EVENT from IPC-Queue
            DO CASE (EVENT)
            CASE SHUTDOWN message
                PROCEED to Shutdown State
            CASE CheckX25 timer expiration
                check-X25 = TRUE
            CASE expected message
                Act on message
            OTHERWISE
                LONG ERROR
            END CASE
        END DO
        Set CheckIPC timer to X25-seconds
        seconds-left = X25-seconds
        DO WHILE seconds-left > 0
            WAIT for incoming packet for seconds-
                left seconds
            IF incoming packet received
                PROCESS packet
                CHECK CheckIPC timer
                SUBTRACT seconds remaining on
                    CheckIPC timer from seconds-
                    left
            ENDIF
        END DO
    END DO
END
```

FIG. 9-2. WAN INTERFACE MODULE OPERATIONAL STATE PROCESSING FOR DIRECT-CONNECT LGN

Once the MNP code is validated, the module services the remote LGN's request. As long as time remains on the CheckIPC timer, the module monitors the X.25 line for another DATA packet, until it is time to switch over to IPC processing.

Many of the IPC messages processed by the WAN interface module are the counterparts to remote messages received via the WAN. The LGN, when processing

TABLE 9-3

MODELS NETWORK PROTOCOL SUMMARY

MNP Code	Meaning
A	Acknowledge (ACK); used to acknowledge requests
C	Request to receive any file being stored in outbound directory destined for calling LGN
E	Request to receive file that matches a prototype
F or P	Request to send a file
H	Remote message
K	Request for confirmation that a table is ready to be received by calling LGN; generally followed by T
N	Negative Acknowledge (NAK); opposite of A
Q	Request to receive any message being stored in outbound directory destined for calling LGN
R	Request to send an EDI file
S or U	Request to receive a particular file
T	Request to receive table specified in reply to previous K request
X	Request to receive any file request being stored in outbound directory destined for calling LGN

IPC messages, sends the same MNP commands that it waits for when monitoring the WAN line. At the start of IPC processing, the CheckX25 timer is set for ulCheckX25 seconds. The module awaits an IPC message until the CheckX25 timer elapses,processing those messages that arrive during the timer interval and switching back to monitoring the X.25 line[2] when it expires.

Except for SUSPEND and SHUTDOWN messages, all IPC messages sent to the WAN interface module prompt a WAN transmission to a remote LGN. The data transmitted can be an EDI file, a table, a message, a request, or a response to a request (which can be a file, message, or table). The most prevalent message processed is SENDEDI, which instructs the WAN interface module to transmit an

[2]For a dial-up LGN, IPC event processing is interrupted by the CheckX25 timer every ulCheckX25 seconds, but the CheckX25 processing is not executed. Instead, every ulCheckX25 seconds, the module takes a momentary break from checking and servicing the IPC queue.

EDI file to a remote LGN. The SENDEDI message contains the full path name of the EDI file, both as it exists on the local LGN and as it will exist on the receiving LGN. It also includes the TORIC, which is an index into the file DODAAC.DAT.[3] This file maps the TORIC to a remote LGN, its network address, and an indicator of whether the remote LGN is dial-up or direct-connect to the WAN. All messages sending data to a remote LGN use the DODAAC.DAT file.

Another IPC message, GETEDIFILE, is sent at regular intervals by the CLGN polling module. This message is used by dial-up sites to poll the CLGN for the following information:

- EDI files
- Messages
- Files other than EDI files
- File requests
- Table updates.

Since a dial-up LGN cannot be called directly, polling the CLGN is the only way that LGN can receive EDI files and other data. The WAN interface attempts to do the poll in one telephone call. After completing each of the above steps, the module proceeds to the next step; if necessary, it redials the WAN.

To initiate a network call, the destination LGN's communications profile first is extracted from the DODAAC.DAT file, as described above. (If LGN-to-LGN communications are in operation and the destination is a dial-up LGN, which cannot be called directly, the destination will default to the CLGN. The CLGN holds data for dial-up LGNs in a staging directory until the remote LGN polls the CLGN for data.) For a direct-connect LGN, the X.25 level-II and level-III links are re-initialized. Next, a CALL REQUEST packet including the addresses of both the calling LGN (only the CLGN in the test) and the called LGN is assembled and delivered across the WAN. The module then awaits a packet on the WAN line. If it is anything other than a CALL ACCEPTED packet, the line is closed and the call attempt is deemed unsuccessful.

---

[3]The DODAAC.DAT file was originally indexed by DoD Activity Address Code (DODAAC), not TORIC; hence the misnomer.

If the LGN has a dial-up connection to the WAN, it communicates with the WAN via a public PAD; it also requires more steps to establish a connection. First, the serial communications port is initialized (once this is done, the serial port can be used indefinitely for communications until the LGN is shut down). Initialization is accomplished with the help of the Greenleaf Comm Library suite of communications functions. In fact, on a dial-up LGN, all interfacing with the serial port (and, thus, with the modem and the WAN) is done via the Greenleaf functions. Once the serial port is initialized, the WAN interface module telephones the network via the serial port. Specifically, it calls a modem (provided by the WAN), which is connected to a PAD, wakes up the PAD, sets some initial PAD parameter values, establishes a WAN session with the remote LGN, and sets some additional PAD parameters. At this point, it is ready to begin an MNP dialog.[4]

For both direct-connect and dial-up LGNs, the WAN interface module makes multiple attempts to call over the WAN. The number of retries for various steps in the WAN calling process is determined by a mixture of module start-up control parameters and hard-coded values. Any message involving data transfer that exceeds a maximum number of tries or that fails during file transfer is put into the holding mailbox, where it will be retried when the Deferred timer expires. After a message has been retried WI_DEFERLIMIT times, it is discarded and an error entry is written to the log.

Every transmission of files or other data between two LGNs is preceded by one or more MNP codes indicating the action to follow. In the case of a CLGN poll, the entire session could consist solely of MNP codes sent back and forth. In cases (such as the processing of a SENDEDI message) where a file transfer ensues, the file transfer mode used is XMODEM/CRC with a buffer size of 896 bytes.

At the completion of a WAN session, the WAN interface module closes the communications line. In the case of a direct connection, it sends a CLEAR REQUEST packet to the remote LGN and waits for a CLEAR CONFIRMED packet in response. If the connection is dial-up, the module hangs up the telephone line.

---

[4]The specific characters exchanged to connect with the PAD and establish a network session are highly tailored to the WAN being used. By the same token, the values for the PAD parameters will probably vary from network to network. In the prototype LGN, the PAD parameters were set to make the WAN as transparent as possible (i.e., no echo, no editing, no flow control, etc.).

In addition to handling IPC messages, the WAN interface takes appropriate action on the expiration of either the IPCStat timer or the Deferred timer. (Recall from the IPC start-up procedure that expiration of the Deferred timer results in the transfer of all deferred messages from the deferred mailbox to the low-priority mailbox.) Since messages in the low-priority mailbox can get bumped in the IPC queue by any other messages, they may never reach the top of the queue. However, in practice, the waiting line of messages in the queue is short, and the low-priority messages usually are processed immediately. In the production LGN, a more sophisticated method will be needed to guarantee that the maximum wait times for messages is not exceeded.

For the IPCStat timer, each time a message is (1) received, (2) sent, (3) deferred, or (4) retried during normal processing, the module updates counters that keep track of those four message activities. When the IPCStat timer expires, the module writes entries to the log and to the local console showing the count for each of those message activities since the previous IPCStat timer expiration. For each message activity, subtotals are listed by message type (e.g., SENDEDI) occurring during the designated time interval.

Data Structures

Global Information Structure (WAN_GLOBAL). The WAN_GLOBAL structure contains parameter values, mailbox and timer handles, and other information reflecting the LGN's current state. Its component variables are used frequently throughout the module. The WAN_GLOBAL structure is shown in Figure 9-3. Nested within the WAN_GLOBAL structure is the DIAL_INFO structure, which contains variables describing the asynchronous communications connection of a dial-up LGN. It is shown in Figure 9-4.

FTC Global Structures (INIT_GLOB and DATA_DESC). Data passed between the Super-X.25 API functions are in the form of globally accessible data structures. The INIT_GLOB structure holds the X.25 level-2 and level-3 initialization parameters. The DATA_DESC structure consists of a pointer to the data area of an X.25 packet plus additional fields that further describe the packet.

Inter-LGN Message Structure (MSG_SENDMSG). Although the MSG_SENDMSG structure is relatively simple, it is important in the sense that

```
typedef  struct   wan_global
{
BOOL              bRestart;              /*  Restart Flag                    */
BOOL              bSuspended;            /*  Suspended Flag                  */
BOOL              bDirect;               /*  Direct or Dial-Up               */
char      sDefaultdir[SY_FILEMLN + 1];   /*  1st Level Directory             */
char      sLgn[SY_FILEMLN + 1]           /*  LGN Identifier                  */
char      sMod_name[SY_MODMNEMLN + 1];   /*  Module Mnemonic                 */
char      sMetaddr[SY_X25ADDRLN + 1];    /*  LGN's Network Address           */
DIAL_INFO         stDial_info;           /*  Dial-Up Information             */
DV_ADIHANDLE      hExp_mailbox;          /*  Expedited Mailbox Handle        */
DV_ADIHANDLE      hLow_mailbox;          /*  Low-Priority Mailbox Handle     */
DV_ADIHANDLE      hDef_mailbox;          /*  Deferred Mailbox Handle         */
DV_ADIHANDLE      hSys_mailbox;          /*  System Mailbox Handle           */
DV_ADIHANDLE      hCheckipc_timer;       /*  CheckIPC Timer Handle           */
DV_ADIHANDLE      hCheckX25_timer;       /*  CheckX25 Timer Handle           */
DV_ADIHANDLE      hDeferred_timer;       /*  Deferred Timer Handle           */
DV_ADIHANDLE      hIPCStats_timer;       /*  IPCStat Timer Handle            */
int               iDeferLimit;           /*  Max # of Message Retries        */
unsigned long     ulCheckipcqueue;       /*  CheckIPC Timer Interval         */
unsigned long     ulCheckX25queue;       /*  CheckX25 Timer Interval         */
unsigned long     ulDefertimeout;        /*  Deferred Timer Interval         */
unsigned long     ulQuiettimeout;        /*  Not Used                        */
unsigned long     ulTeardowndelay;       /*  Not Used                        */
unsigned long     ulIpcstatinterval;     /*  IPCStat Timer Interval          */
} WAN_GLOBAL;
```

FIG. 9-3. WAN INTERFACE GLOBAL INFORMATION STRUCTURE

```
/* Global dial-up information (ignored for direct-connect LGN) */

typedef   struct   dialup
{
int       iPort;          /*  Com Port Number (0 = COM1, 1 = COM2)    */
int       iBaud;          /*  Baud Rate                               */
int       iParity;        /*  Parity (Odd, Even, or None)             */
int       iStopbitss;     /*  Number of Stop Bits                     */
int       iDatabits;      /*  Number of Data Bits                     */
int       iAttempts;      /*  Max Number of Dial Attempts             */
int       iSpeaker;       /*  Modem Speaker Status (0 = Off, 1 = On)  */
char      sNetphone[21];  /*  WAN Telephone Number                    */
} DIAL_INFO;
```

FIG. 9-4. DIAL_INFO GLOBAL STRUCTURE

every message sent over the WAN is passed in this structure. It is shown in Figure 9-5.

```
/* Message structure used when sending a message over the WAN                    */

typedef   struct    msg_sendmsg
{
MODELSHEAD                      mMsg_head;          /*  MODELS message header    */
char      sDestmod[SY_MODDMNEMLN + 1];              /*  Destination module       */
char      sLgn[SY_LGNMLN + 1];                      /*  Destination LGN          */
BYTE                            bPriority;          /*  Message priority         */
int                             iMsgLen;            /*  Length of embedded msg.  */
char      acMsgtest[SY_MAXMSGLN];                   /*  Message text and data    */
} MSG_SENDMSG;
```

FIG. 9-5. INTER-LGN MESSAGE STRUCTURE

Algorithms

Queue Service. The means by which the WAN interface module equitably services both incoming X.25 packets and IPC events was described earlier under Flow of Processing.

Clock Synchronization. A clock synchronization algorithm was incorporated into the prototype system but has not been activated during the test. Activating the code is a trivial operation. But, since the algorithm is relevant to the design of a production LGN, it is described here nonetheless. Because all transaction files are timestamped as they move through the MODELS process – from the source LGN, through the WAN, to the destination LGN – the system clocks in each LGN in the system must be closely synchronized. The clock synchronization algorithm has the WAN interface module at each LGN set up a session with the CLGN during its Initialization State. The LGN then sends a series of WI_CLKREQCNT messages of type CLKREQ to the CLGN, each separated by a 1-second delay. Each message contains the system time according to the local LGN. The CLGN responds to each CLKREQ message with a CLKRESP message containing the time originally entered by the local LGN and the time (from the CLGN's viewpoint) that the CLGN received the message. To this information, the local LGN adds the time the response was

received (according to the local LGN). After WI_CLKREQCNT query-response pairs, three sets of numbers will have been developed:

{QL(i) : 1 <= i <= WI_CLKREQCNT} Time (according to local LGN) when query i was sent.

{RY(i) : 1 <= i <= WI_CLKREQCNT} Time (according to CLGN) when query i was received.

{RX(i) : 1 <= i <= WI_CLKREQCNT} Time (according to local LGN) when response i was received.

The average round-trip propagation delay for a query-response is:

$$eeee2\mu - \frac{\sum \left[R_L(i) - Q_L(i) \right]}{10}$$

Assuming the delay in each direction is equal,[5] then the average delay in each direction is:

$$\mu - \frac{\sum \left[R_L(i) - Q_L(i) \right]}{20}$$

Adjusting for the delay, the LGN's clock is set to:

$$\frac{\sum \left[R_c(i) - Q_L(i) \right]}{10} + \mu$$

Differences Between LGN and CLGN Implementations

The WAN interface module makes no distinction between the LGN and CLGN environment, even though only the CLGN is capable of executing such tasks as remote module suspensions or communication with another LGN on the WAN. These differences are all resolved by either (1) input tables or (2) the makeup of messages sent to the WAN interface module by other modules in the LGN.

[5]If a systematic bias in directional propagation delay exists and is constant, a small refinement in the algorithm can correct it.

Shutdown Procedures

The WAN interface module acts when it receives a SHUTDOWN or SUSPEND message. In addition to following the normal procedure of suspending message processing, it disconnects from the direct or dial-up communications line and stops its array of timers.

Serialization

It is not feasible to serialize the WAN interface module with any other processes; it must be constantly monitoring the X.25 line for incoming packets. In addition, other tasks such as IPC statistics updating and CLGN polling (for a dial-up LGN) depend on continuously running timers.

Files

Two types of files – EDI files and general files – are sent and received by the WAN interface module. The primary processing difference between them is that the EDI file is deleted after successful transmission. The WAN interface module of the sending LGN is responsible for names and locations of the files.

Alternative Designs

The WAN interface module of the prototype LGN suitably performs its task of providing a WAN communications link. However, as a by-product of running the test in a number of different environments, some desirable alternative design features for the production LGN have been uncovered:

- The ability to handle more than one logical WAN session. This capability is already inherent in the X.25 protocol and in the FTC hardware and supporting software. Additional logic would need to be programmed into the module.
- It would be beneficial, especially in a high-volume, multiple virtual session environment, for the module to be able to recognize and take action in response to other types of packets, those it currently ignores. The X.25 supervisory packets in particular are helpful in determining the causes of communications errors.

CLGN POLLING MODULE

Purpose and Description

The CLGN polling module is a small, single-purpose module activated on dial-up LGNs only. It regularly sends messages to the WAN interface module directing it to poll the CLGN for EDI and other files, file requests, and remote messages.

Start-Up Procedures

The CLGN generally performs a subset of the standard LGN module Initialization State sequence. The module receives only system messages from SB. Therefore only the default, system mailbox is used. In addition to standard systemwide parameters covering general LGN information, the following parameters are obtained from configuration files during start-up:

CL_POLLINTERVAL	CLGN polling interval (number of seconds)
WQ_QUEUEDIR	Outbound directory of EDI files on the CLGN
WQ_STAGE1DIR	Destination directory for files received from the CLGN

The module creates the CLGN Polling timer, which is used during the Operational State to regulate the sending of messages to the WAN interface module.

Processing

The CLGN polling module consists of one program, which responds to the usual system messages (SUSPEND, SHUTDOWN, and RESTART) and only one other event: expiration of the CLGN Polling timer.

Frequency of Operation

The module runs continuously. Most of its time is spent waiting in the IPC queue for the expiration of the CLGN Polling timer.

Flow of Processing

After receiving an OPGO message from SB signaling transition to the Operational State, the CLGN polling module sets the CLGN Polling timer to CL_POLLINTERVAL seconds. Subsequently, the module enters its main processing loop of parking in the IPC queue awaiting a system message

(i.e., SUSPEND, SHUTDOWN, or RESTART) or the expiration of the CLGN Polling timer. When the timer expires, the module forms a GETEDI message and sends it to the WAN interface module.

To accommodate the testing environment, the polling interval has been changed frequently. Suppose five new files were to be sent to a dial-up CLGN with the polling interval set at 24 hours; it would take 5 days to update the LGN. To overcome this problem, the polling module reads the CL_POLLINTERVAL parameter with every timer expiration, setting the timer to the most recently acquired CL_POLLINTERVAL value. In the example, the CLGN first sends a new polling configuration file specifying a smaller polling interval, typically 30 minutes. From then on, the LGN polls the CLGN every 30 minutes instead of once a day. After all the files are sent, the CLGN sends the LGN another configuration file restoring the original polling interval of 24 hours.

Shutdown Procedures

Upon receipt of a SHUTDOWN message, the CLGN polling module follows standard LGN module Shutdown State procedures.

Serialization

This module does not run in a serial fashion; its main processing task is dependent on a continuously running timer that is independent of any other LGN activity.

Alternative Designs

In a production system, the tasks performed by the CLGN polling module could be incorporated into the WAN interface module. For purposes of developing the prototype, making the CLGN polling function a separate module was a simpler design, providing more modularity.

CHAPTER 10

TRANSACTION PROCESSING SUBSYSTEM

Figure 10-1 shows the transaction processing subsystem in terms of

- Its component modules
- Tables used by the translation modules
- The flow of information between modules
- The interfaces between the transaction processing subsystem and other subsystems with which it communicates.

The prototype LGN cannot recognize EDI transactions or Service-specific transactions that fall outside of the DLSS format (as understood by the LGN). In the operational system, both of these types of transactions will be passed directly to the WAN queuing module, bypassing the translator.

LAN DEQUEUING MODULE

Purpose and Description

The LAN dequeuing module (shown in Figure 10-2) transforms files downloaded by the local interface module into the DLSS-to-EDI (D2E) translator format. It is able to distinguish between single-card and multiple-card DLSS transactions. Cards belonging to multiple-card transactions are grouped together and sorted. The module assigns a unique transaction ID to each transaction processed. Data that are not part of any transaction are discarded.

The module is made up of two DESQview processes (two programs): the main event-handling process and an AWK filter process. To support a single translation, there may be up to five instances of the AWK filter process.

Start-Up Procedures

The LAN dequeuing module is created when the system boots. It is initialized via standard module initialization code. As usual, it does not open or use the low-priority mailbox. Two other mailboxes are created: the expedited mailbox and a

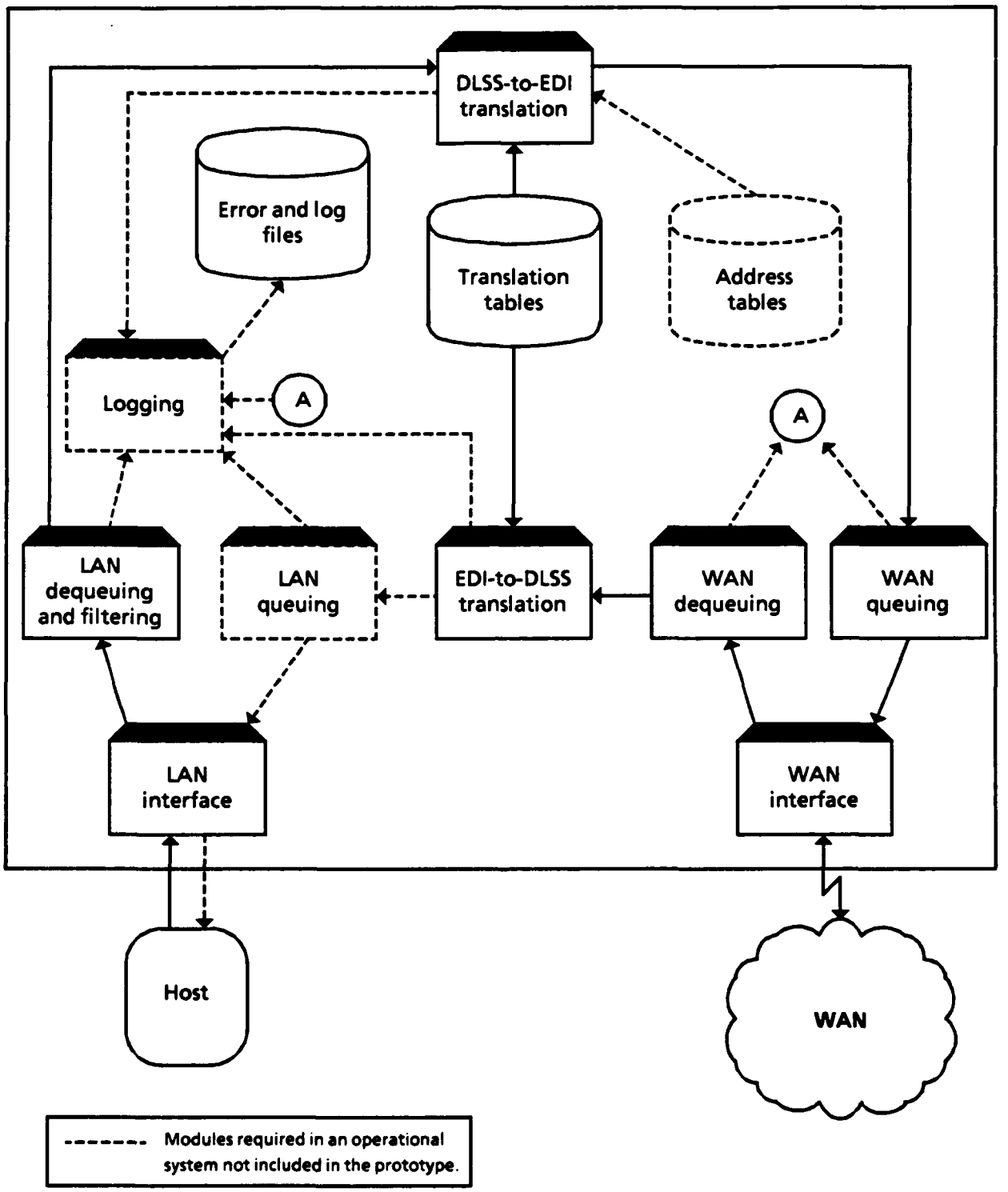


FIG. 10-1. TRANSACTION PROCESSING SUBSYSTEM

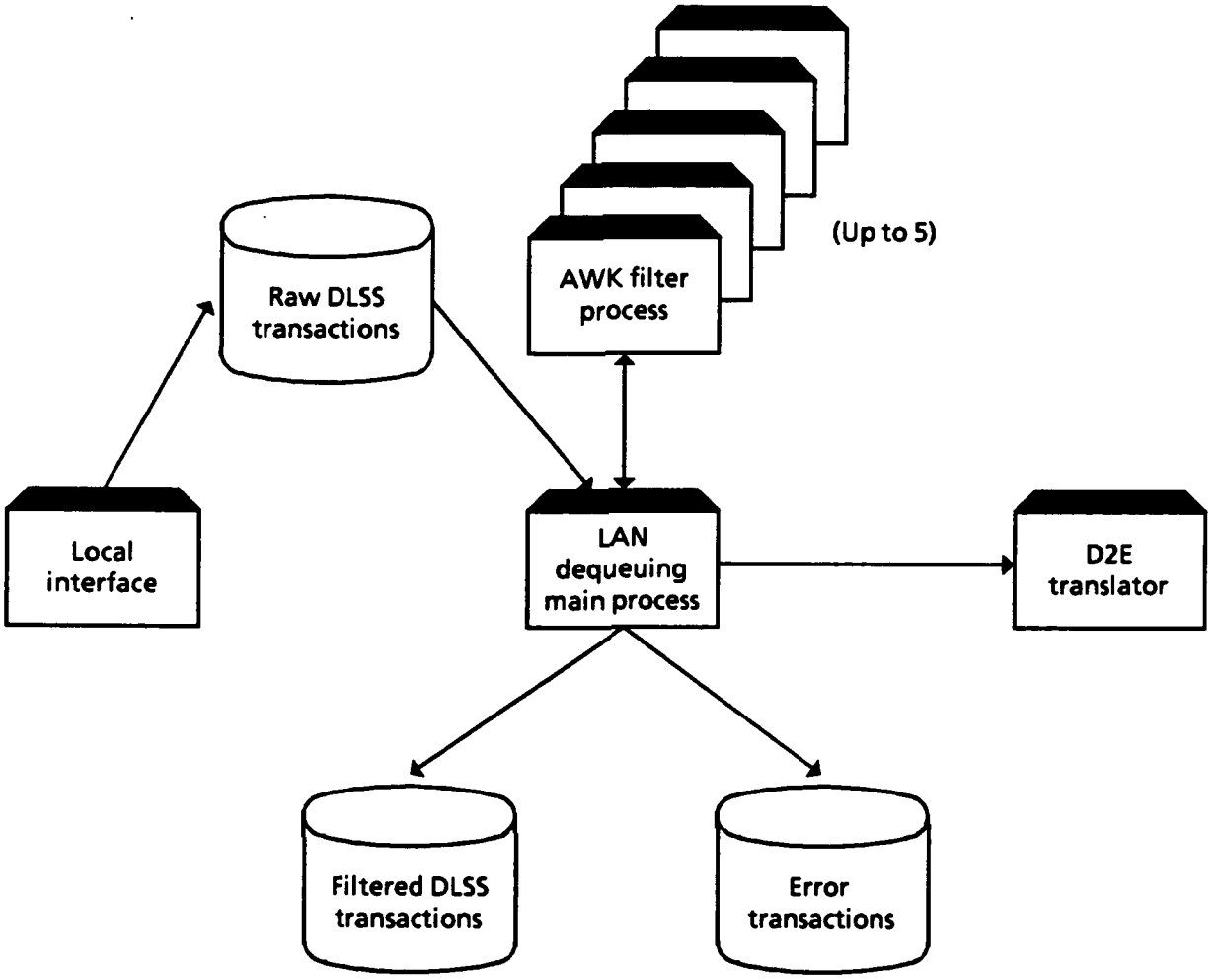


FIG. 10-2. LAN DEQUEUING MODULE

"semaphore mailbox." While, according to DESQview, the semaphore mailbox is technically a mailbox, in reality it is a semaphore. Only one process can have possession of the semaphore at any given time. A process can determine whether or not it currently owns the semaphore; by this method synchronization among processes can be achieved.

Only one module-specific parameter, LD_FILTERPIFDIR, is read during directory initialization; it contains the PIF for the AWK filter process. The main LAN dequeuing program requires the location of the filter process' PIF in order to read and use the file in starting the filter process during the Operational State. Another parameter, the current serial number, used during the Operational State is

read on an as-needed basis during processing. The value is kept as an ASCII string in a separate file called SERIALNM.CFG. This file is updated each time a batch of transactions is filtered.

Processing

As is the case with other modules in the LGN, the main processing role of the LAN dequeuing module is to await specific IPC events. Critical IPC messages to this module are RCVDDLSS, from the local interface module, and FILTERDONE, an intra-module message from the filter process.

Frequency of Operation

The main event-handling process of the LAN dequeuing module is always running. When not processing a specific message, it is parked in the IPC queue. The filter process is executed when needed and is terminated once it has completed its filtering tasks.

Flow of Processing

The main loop checks for RCVDDLSS messages in the expedited mailbox; when one is in the queue, it is removed and processed. The sFilenm field in the RCVDDLSS message contains the name of the file to be filtered. The file name consists of a timestamp, assigned previously by the local interface, with the extension .DLD. The file is in the directory \LI\DNLD\QUEUE. Another field in the RCVDDLSS message specifies the host name. The LAN dequeuing module is designed to invoke a separate filter process for each host connected to the LGN. During the prototype test, the LGN has been connected to only one host, so only one file filter has been developed.

When the RCVDLSS message is read, a separate process is started to filter the file sFilenm. The parameters sent to the filter process are: (1) the host name; (2) the file name, without the extension (i.e., the file's timestamp only); and (3) the LGN name. The filter process uses the LGN name as a component of the unique transaction ID assigned to each DLSS transaction.

Up to five filter processes at a time can be active, although the module runs only one active filter at a time; the other four are dormant, awaiting the semaphore LD_SEMAPHORE. The filter process is a batch file consisting of (1) a header C-language program that gains ownership of the semaphore; (2) the filter programs,

which are written in AWK and SuperSort; and (3) a trailer C-language program that releases the semaphore and sends a FILTERDONE message to the main C-language event-handling process.

The actual filtering and sorting are done by a series of AWK programs and SuperSort scripts. The SuperSort scripts inform the SuperSort software of the file names, sort keys, and collating sequence. Generally, successive AWK programs, upon recognizing a type of DLSS transaction, process and write it to an output file. The main output file created by the filter process is C:\LD\VAL\<time-stamp>.VAL, where <time-stamp> is the timestamp of the transaction file being filtered. An error file, \LD\FILTER\ERROR.TXT, is also produced. Because of space limitations and lack of high-capacity, removable media in the prototype LGN, this file is written over each time the filter process is run.

For each transaction, the filter writes to the output file a unique ID, which precedes the transaction. The transaction ID is a concatenation of the tilde character (~), the three-letter LGN name, and a sequence number. The sequence number is incremented by one each time a transaction ID is formed and written. At the end of each filter program, the current sequence number, which is the one applied to the next transaction, is written to the file C:\CFG\SERIALNM.CFG.

Generally, each transaction is distinguished by its Document Identifier Code (DIC) field, located in the first three columns. If the transaction is determined to be a single-card transaction, a transaction ID is written to the output file, followed by the transaction. If it is a multiple-card transaction, more pre-processing, and consequently more programs, are involved. For these transactions, the values of other fields are used as qualifiers and keys to separate transactions and to sort multiple card images within transactions for processing by the D2E translator.

The main event-handling process keeps track of outstanding filter processes. Each is identified by the timestamp of the filtered file; the FILTERDONE message includes this identifier to identify which process has been completed. At this point, a TRANSDLSS message is formed and sent to the D2E translator module; also, the unfiltered DLSS file is deleted.

Shutdown Procedures

For the most part, the LAN dequeuing module follows the standard prototype LGN module shutdown procedures. Its one additional task is to cancel the semaphore mailbox.

Serialization

Since the prototype LGN operates on a file of transactions, rather than on a single transaction at a time, serialization would impose less severe decreases in performance than it would if the LGN were truly a transaction-processing system. Under the prototype conditions, the LAN dequeuing module is a good candidate for serialization.

The LAN dequeuing module could be invoked following the download of a DLSS file, and managing multiple filter processes would be unnecessary. If two or more downloads occurred in rapid succession (which happens), the design should allow more than one instance of the module to run simultaneously. The logic for assigning serial numbers would have to be modified accordingly, to ensure that duplicate numbers are not assigned.

Files

Two main files are processed by the LAN dequeuing module: the input file of unfiltered DLSS transactions and the output file of filtered DLSS transactions. The file ERROR.TXT contains card images considered by the filter process not to be part of a transaction. In addition, several temporary files are used during the filtering of multiple-card transactions.

DLSS-TO-EDI TRANSLATION MODULE

Purpose and Description

The DLSS-to-EDI translation module (D2E translator) performs the core LGN function of translating transactions from DLSS into EDI format. The operating context for this module is shown in Figure 10-3.

The translation module receives a TRANSDLSS message from the LAN dequeuing module that specifies a DLSS file to be translated into EDI format and triggers the module to process all transactions in that file. The translation logic that

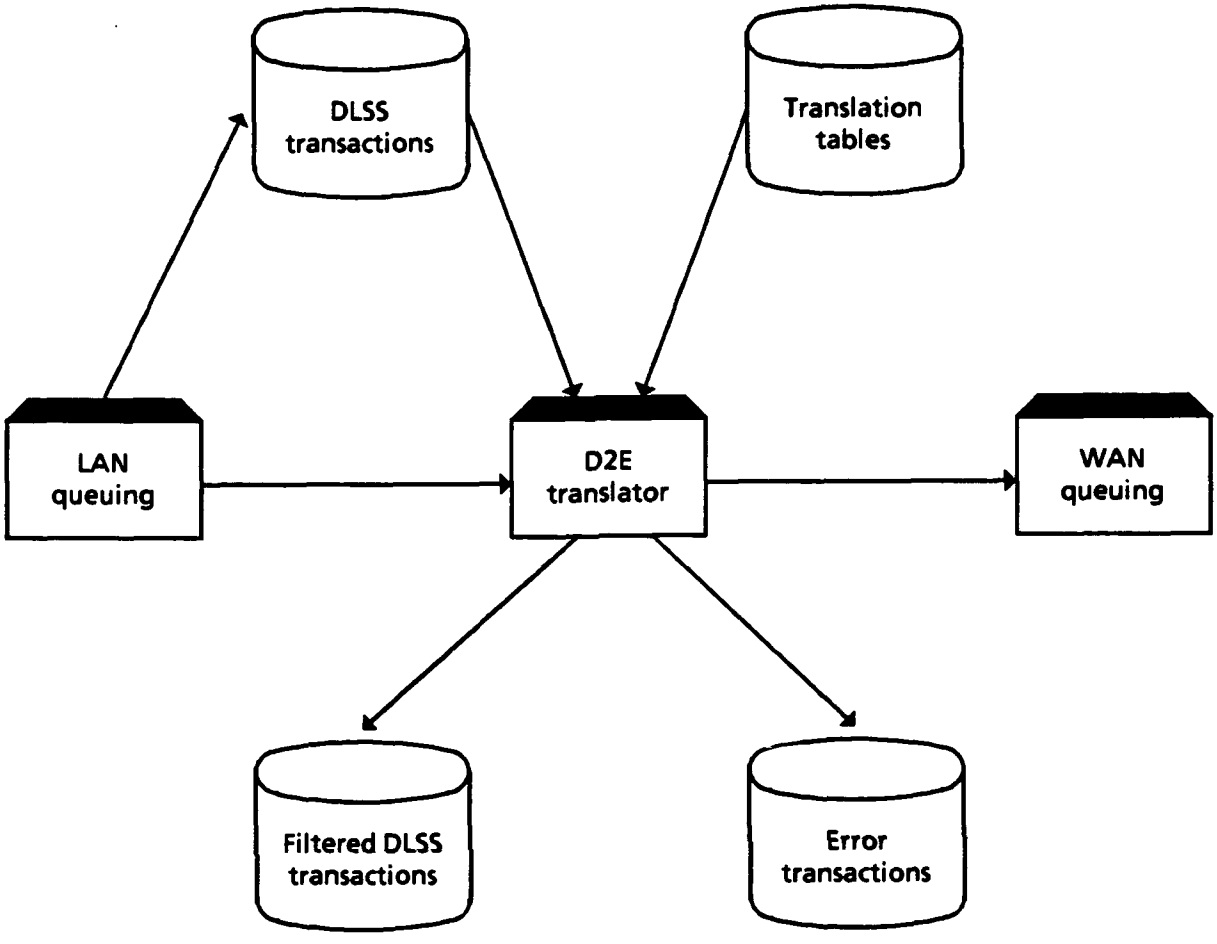


FIG. 10-3. MODULE FOR DLSS-TO-EDI TRANSLATION

drives the module resides primarily in tables designed by LMI, using Paradox structures and syntax. Some supplemental logic, in the form of frequently used utility routines, is incorporated in the translator module itself.

The translator is a C-language program that reads the translation rules from the tables at module start-up and compiles them into an internal "portable-code" (P-Code) abstract stack machine format. An abstract stack machine uses a set of pseudo-machine-language instructions that can be evaluated by an abstract stack machine. The language constructs of the translation tables are simple compared to those of a conventional programming language (like C, for example), keeping the abstract stack machine fairly uncomplicated. However, the syntax is not trivial; as the apparent data interrelationships and cross-references used to specify the translation process become more complicated and numerous, the sophistication of the

table logic increases proportionately. The translation P-Code is stored entirely in memory, greatly increasing the performance of the translator in comparison to that which would be obtained by accessing the translation logic directly from the tables. The utilities YACC and LEX are used to build the compiler.

The translation tables are converted to P-Code by using an incremental compiler. In theory, this process provides the capability to revise one portion of the logic or add a new rule without having to recompile the entire set of tables. During the prototype test, however, the mechanisms for notifying the program of the particular translation logic revisions have not been developed. Therefore, when a change to the translation tables takes place, the translator reloads the entire set of tables and recompiles all the logic rules.

Start-Up Procedures

The D2E module is a single C-language executable process. At start-up, the module performs the standard module start-up procedures. Once those procedures are accomplished, the translation tables are loaded into memory and converted to the internal P-Code format. Because of the added workload shouldered by the translation module during the Initialization State, its start-up time is significantly longer than that of other modules (except the EDI-to-DLSS translation module): up to 3 minutes on a Compaq 386/20 with a disk cache.[1] The length of time to complete start-up procedures is determined by the number of records in the translation tables.

As usual, the module uses its expedited mailbox as the vehicle for receiving messages. The low-priority mailbox is not opened and consequently is not used. In contrast to the other modules in the LGN, the D2E module does not read any module-specific parameters during start-up. All information required by the module is either hard-coded, passed through parameters in messages, or implied.

---

[1]These times are based on a "fully loaded" LGN; i.e., an LGN with 11 or 12 processes all initializing at the same time. A stand-alone version of the translator, built expressly for LMI translation table development and analysis, yields much quicker times: approximately 30 seconds with all other conditions equal.

Processing

The D2E module waits in the IPC queue for (1) a system message from SB or (2) a TRANSDLSS message triggering a DLSS file translation. The translation itself is driven by the translation tables.

Frequency of Operation

This module is always running, but it is dormant until a message arrives for processing.

Flow of Processing

The receipt of a TRANSDLSS message in the IPC queue starts a DLSS-to-EDI translation of the file denoted in the message. The file is read one transaction (one or more DLSS card images) at a time. The translation is driven mainly by the tables DIC2TID and DLSS2EDI, with the EVALDLSS table playing an important supporting role. The relationships among the translation tables are shown in Figure 10-4.

The DIC2TID table provides the first step in translating a transaction. One or more records in the DIC2TID table referencing a unique transaction identification (TID) form a TID section. The first TID section record to be translated corresponds to the DIC of the DLSS transaction found in the DIC2TID. Internal tables (in program memory) map the TID section to the first DLSS2EDI record to use for translation of the TID section.[2] The DLSS2EDI table is essentially a step-by-step sequence of instructions describing how to build an EDI transaction in terms of its component segments and fields. The table is grouped logically by TID section. Each record entry in the DLSS2EDI table corresponds to a conditionally[3] created EDI field in the target transaction.

The DLSS2EDI table also contains records that do not correspond to any particular EDI element. These records are used to generate "side effects," such as

---

[2]Remember that the tables are loaded into memory compiled, so all references to table records are actually to internal representations of table records; likewise, all record positions resolve to memory offsets.

[3]The [condition] field determines whether or not the EDI element will be created in the current EDI segment; when condition is true, the EDI element created follows the rules in the [eval] field.

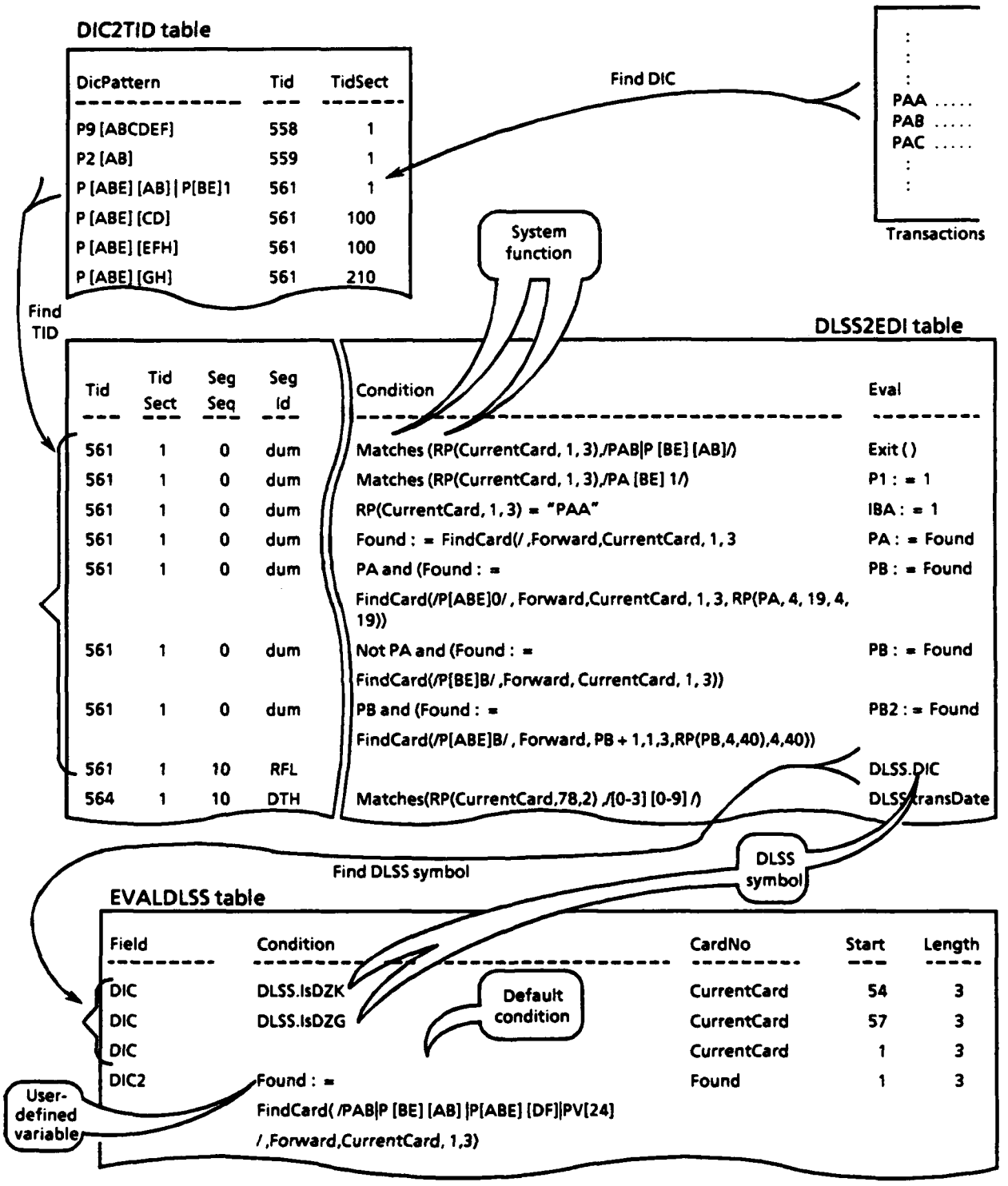


FIG. 10-4. TABLE INTERACTION FOR DLSS-TO-EDI TRANSLATION

assigning a value to a user-defined variable or exiting from the current section. The segment sequence number in these records is always zero.

The translation proceeds through the DLSS2EDI rules, building EDI segments according to the DLSS2EDI rules specified for that section an element at a time, until it reaches the last DLSS2EDI record in the section. At that point, the program returns to the DIC2TID table for another TID section to translate, repeating the translation loop until no unprocessed TID sections remain.

How does the EVALDLSS table fit in? In the [condition] and [eval] fields of the DLSS2EDI table, references are made to logical names, which roughly equate to fields in a DLSS transaction or to EDI elements. These logical names are essentially the atomic units of the translation rules, and they are defined in the EVALDLSS table. The logical names themselves may be defined in terms of conditions and further transformations, referring as necessary to other logical names. The translation rules for EDI elements can be quite complicated.

If the translation of a transaction fails, the segments produced up to that point for the transaction are written to an error file.

If at least one transaction from the file is translated correctly, a DISPATCHEDI message is sent to the WAN queuing module.

SUSPEND messages are handled in the usual way. A RESTART message signals a transition out of the suspended state, and before re-initializing, the module frees all memory associated with internal representations of the translation tables.

Data Structures

Several aggregate data types associated with the translation module relate to the compiler and the execution engine for compiled code: symbol tables, parse trees, the evaluation environment, and page tables for input and output data.

Two symbol tables are maintained for the translator: a DLSS evaluation table and a system symbol table. The DLSS evaluation table is the internal representation of the logic defined in the translation table EVALDLSS. The C-structure of this entity is shown in Figure 10-5. The table contains information derived directly or indirectly from the translation table: the symbol name (pszName), the symbol's internal type (tType) and type as presented in the Paradox table (cType), a pointer to

the P-Code that evaluates it (codEval), and the length of the P-Code (wCodeLen). Since the DLSS symbol is re-evaluated every time it is referenced, its most recent value is not stored in the symbol table; rather, the rules for its evaluation (i.e., the pointer to its P-Code) are stored.

```
typedef  enum tagTYPE
(
eVoid                   /*  No type                    */
,eInteger               /*  16-bit integer             */
,eString                /*  String                     */
,eBoolean               /*  Boolean (integer)          */
,eRegex                 /*  Regular expression         */
,eTypeCount             /*  # of types                 */

typedef  struct tagDLSS_SYMBOL
{
char     *pszName;      /*  Symbol name                */
TYPE     tType;         /*  Data type                  */
char     cType;         /*  Type as coded in data base */
PCODE    codEval;       /*  Evaluation P-Code          */
WORD     WCodeLen;      /*  Length of P-Code           */
} DLSS_SYMBOL;
```

FIG. 10-5. EVALDLSS SYMBOL TABLE ENTRY

Similarly, the translator maintains an array of entries representing records in the DLSS2EDI table. Each D2ETRANS entry corresponds to a rule for conditionally including a particular EDI element. The fields in the D2ETRANS table correspond closely to field counterparts in the DLSS2EDI table. The D2ETRANS entries are grouped together by TID section into a larger DLSS2EDI structure. The structure for these entities is shown in Figure 10-6.

The language that captures translation logic (as contained in the fields [EVALDLSS -> Condition], [EVALDLSS -> Transformation], [DLSS2EDI -> Condition], and [DLSS2EDI -> Eval]) is dubbed TransLog. As computer languages go, TransLog is uncomplicated, but not trivially so: (1) it has few data types and few operators; (2) symbols scope either to a TID section or are global; (3) it allows for a maximum of 20 run-time declarable variables, all of which are of type integer; and (4) it is strongly typed – each symbol has exactly one type, which is declared explicitly. The type of each DLSS symbol is declared in the EVALDLSS data base and is mapped to an internal type during start-up. The types supported in the data

```
typedef  enum tagD2ETRANS
{
WORD   wSegSeq;          /*  Segment seq number      */
char   aszSegId[kwTR__MaxSegIDSz + 1];
                         /*  Segment id              */
WORD   wSegUse;          /*  Segment use #           */
WORD   wEltSeg;          /*  Element use #           */
WORD   wOutputSection;
                         /*  Output section          */
PCODE  codEval;          /*  Evaluation P-Code       */
WORD   wCodeLen;         /*  Length of P-Code        */
} D2ETRANS;

EXTERN  struct tagDLSS2EDI
{
WORD   wTid;             /*  Transaction ID          */
WORD   wTidSect;         /*  TID section             */
WORD   wNumrecs;         /*  Number of D2ETRANS      */
D2ETRANS       *recTranslation;
                         /*  Trans. records          */
} garecDlss2Edi{kwTR__MaxNumTidSects];
```

FIG. 10-6. DLSS2EDI TABLE ENTRY

base and their internal equivalents are shown in Table 10-1. The compiler enforces strict typing.

TABLE 10-1

DATA TYPES IN EVALDLSS DATA BASE

Data base type flag	Data base type	Symbol table type
A	Alpha	eString
U	Untrimmed alpha	eString
L	Logical	eBoolean
D	Date	eString
N	Numeric	eString

The DLSS evaluation table is an array of objects such as those in Figure 10-5. It is loaded at start-up from the EVALDLSS table. The load process is actually a two-pass compiler and code generator that builds the P-Code for each function.

Where a field representing different conditions contains multiple translation table records, they are collapsed into a single table entry, and multiple table records are represented as different blocks of code within the P-Code. The subfield codEval, within the table entry, points to an array of P-Code entries associated with the EVALDLSS symbol. The D2ETRANS table is loaded in similar fashion.

The P-Code is generated during the compiler's second pass. (Other subfields of the EVALDLSS symbol table entries are resolved during its first pass.) The generic structure of the code is:

```
typedef struct tag<structure_name>

{
BYTE    opCode;
<some-type>  <hungarian_prefix>Operand
} <structure-name>;
```

where:

- <structure-name> is the mnemonic representing the structure for a particular type of P-Code instruction (e.g., STRING_OPERAND_INSTRUCTION).
- <some-type> is either char, BYTE, int, WORD, REGEX_CODE (regular expression code, also compiled), STRING, or FUNC (pointer to a system function).
- <hungarian-prefix> is the Hungarian notation prefix corresponding to the variable type named in <some-type>.

Generally, a line of code consists of an instruction and an operand. From the structure, it can be seen that the operand can be any one of seven types, depending on the nature of the instruction. The uses of these operands are illustrated below.

Table 10-2 contains a list of the instruction set supported by the abstract machine,[4] along with a brief functional description. P-Code for an EVALDLSS symbol consists of streams of these instructions generated by the code generation pass of compilation during translator initialization. codEval points to the beginning of this stream of P-Code instructions.

[4]A P-Code entry for an abstract stack machine is analogous to a machine instruction for a true machine.

TABLE 10-2

P-CODE INSTRUCTION SET

Instruction	Operand type	Description
Pop	none	Discard the top value on the stack and decrement the stack pointer.
Push	WORD	Push the operand value onto the stack.
PushString	STRING	Push the length and address of the operand onto the stack.
PushRegex	REGEX	Push the address of regular expression P-Code onto the stack.
Copy	none	Copy top value of stack onto the position "1 beyond the top," which becomes the new top.
Calls	FUNC	Call the system function pointed to by the operand.
CallU	WORD	Recursively evaluate the P-Code entry for the DLSS value whose index is equal to the operand value.
LVal	WORD	Pop the value on the top of the stack to the system variable whose index is equal to the operand value.
Rval	WORD	Push the value of the system variable whose index is equal to the operand value onto the stack.
EQString	none	Compare the string at the top of the stack with the string below it for equality, pop the stack, and set the result (<0, 0, or >0).
WEString	none	Like EQString, except check for inequality.
EQ	none	Compare the integer at the top of the stack with the integer below it for equality, pop the stack, and put the result on top of the stack.
NE	none	Like EQ, except check for inequality.
And	none	Perform a logical AND with the integer on the top of the stack and the integer below it, pop the stack, and put the result on top of the stack.
Or	none	Perform a logical OR with the integer on the top of the stack and the integer below it, pop the stack, and put the result on top of the stack.
SBPJump	WORD	Jump (advance the P-Code pointer) forward the number of bytes specified in the operand. Also set the base pointer (used in jumps) to the location in the P-Code following the SBPJump instruction.
Jump	BYTE	Jump forward the number of bytes specified in the operand.
JumpT	BYTE	If the top of the stack is true (non-zero), jump forward the number of bytes specified in the operand.
JumpF	BYTE	If the top of the stack is false, jump forward the number of bytes specified in the operand.
JumpI	BYTE	Jump forward the number of bytes specified by the local variable whose index in the local variable array matches the operand value.
JumpIT	BYTE	Like JumpI, except the jump is conditional on the top of the stack being true.
JumpIF	BYTE	Like JumpI, except the jump is conditional on the top of the stack being false.
Not	none	Perform a logical NOT on the integer on the top of the stack.
NOP	none	No operation.
Inc	WORD	Increment the DLSS symbol whose index is equal to the operand value.
Plus	none	Add the integer on the top of the stack to the integer below it, pop the stack, and put the result on the top of the stack.
Minus	none	Add the integer on the top of the stack to the integer below it, pop the stack, and put the result on the top of the stack.

Expressions in TransLog are built up from references to EVALDLSS functions, user-defined variables, and system primitives. User-defined variables are simply identifiers in a TransLog expression that are not a reference to an EVALDLSS entry or a system function. The compiler considers them to be integers, implying that they can be also be used as Boolean. Their values are cleared at the start of a new TID section translation.

System primitives are functions or variables; they are stored in the system symbol table and are initialized during start-up. The number of system functions has grown gradually throughout the prototype test; the complete list of system functions at test end is listed in Appendix B. There are two kinds of system symbols: (1) variables, including those whose value is constant and those whose value can change during a translation, and (2) functions that transform or retrieve some value. The functions can be thought of as similar to EVALDLSS symbols, except for two factors:

- System functions can have parameters.
- System functions have actual compiled C-language code underlying them, not P-Code.

The system function table is needed for the compiler to recognize valid system function names, validate the numbers and types of parameters for a system function call in TransLog, and connect the call to the run-time routine at code-generation time.

The system symbols stored in the system table have their own structures. Figure 10-7 shows the C-language structure for system-table symbol entries. Symbols in class eFunction are evaluated by executing the function pointed to by fnFunction. Symbols in class eVariable store their current value at vValue. The symbol table consists of an array of symbol table entries.

A second significant data structure for the compiler is for parse trees. Parse trees are generated by the syntax checker of the compiler. They represent derivations of expressions from the syntax rules of TransLog. TransLog statement parsing will be handled by the inherent capabilities of YACC, the compiler generator.

```
typedef enum tagSYM_CLASS
{
eFunction                          /*  Symbol is function          */
,eVariable                         /*  Symbol is variable          */
}SYM_CLASS

typedef struct tagSYS_SYMBOL
{
char          *pszName;            /*  Symbol name                 */
TYPE          tType;               /*  Data type                   */
SYM_CLASS     scClass;             /*  Function or variable        */
BYTE          byNumParams;         /*  Number of fixed params      */
BYTE          byMaxNumParams:      /*  Max number of params        */
BYTE          byTupLeSize;         /*  Repeat param group size     */
BOOL          bIsPrimitive;        /*  Always Yes in prototype     */
BOOL          bIsScopedToTransaction;
                                   /*  Reset to zero?              */
TYPE          *ptParamType;        /*  Array of param types        */
VALUE         uValue;              /*  Current value               */
FUNC          fnFunction;          /*  Code for function call      */
}SYS_SYMBOL
```

FIG. 10-7. SYSTEM TABLE ENTRY AND RELATED STRUCTURES

A third major set of data structures is the execution environment, containing

- A global execution stack
- A stack pointer
- An instruction pointer
- A call walk-back.

The global execution stack is simply an array of type integer. The stack pointer is the current position in the stack; it is initialized to zero. The instruction pointer references the P-Code instruction under current evaluation; it is initialized to the symbol's codEval field at the start of P-Code evaluation for a symbol. The call walk-back is an array holding a pointer to the current symbol table entry, as well as pointers to all DLSS symbols referenced recursively during P-Code evaluation. By way of the DLSS symbol entry codEval subfield, the call walk-back array points to all P-Code entry points in the call stack.

The translation module also uses a set of data arrays to map DLSS images, page table entries, and file locations to each other. A similar array maps EDI segments to

file locations. Because DLSS transactions can be hundreds of card images in length and the amount of memory to store a DLSS transaction is limited, images pertaining to the current transaction are brought in as needed; the program allocates space for the 100 most recently used images. The following data items and structures are used in DLSS card-image mapping:

gaiDlssSwap2CardMap[ ]	maps page table entry to card number
garecDlssCard2File[ ]	maps card number to page table entry and to input file location
gapszDlssCardSwap[ ]	array of buffers, each of which holds the card image associated with a particular page table entry.

The structure garecEdiSeg2FileMap maps EDI segments to output file locations. Actually, the EDI segments are written temporarily to a sort file before being written to the EDI output file.

Algorithms

The translator module makes significant use of algorithms. These are discussed as they apply to the EVALDLSS symbol table, in particular, and, in much the same way, to the D2ETRANS array entries.

Overview of Compiler Processing. The compiler loads the EVALDLSS table into memory in two passes. The first pass populates the internal EVALDLSS symbol table with all the field names in the data base. All the subfields for each symbol table entry are filled in except for codEval. In the second pass, code is generated for each entry; the codEval subfield for the entry points to the allocated space for the code.

In the second pass, each symbol is compiled in two phases. First, a lexical and syntactical analysis is performed to generate a parse tree from the TransLog code for the symbol. If any symbol fails lexically or syntactically, the compile fails and the program aborts. For symbols that have more than one record in the translation EVALDLSS table, the records are collapsed as follows:

[Condition]1:\n[Transform]1;\n[Condition]2:\n[Transform]2;\n...
...[Condition]n:\n[Transform]n.

where different conditions and transforms represent successive data base records for the same symbol[5] and \n represents the "new line" character. Such a statement is interpreted as executing the transform that follows the first true condition. A null condition is interpreted as true, while a null transform is interpreted as no transform.

Lexical Analysis. Lexical analysis is performed by a finite stack machine in C-language generated by the LEX utility. The TransLog grammar used for LEX is shown in Figure 10-8 in Backus-Naur form. Note that the grammar is not case-sensitive; all names are converted to upper case by the lexical analyzer. Also note that EVALDLSS name references are of the form DLSS.name, but only name is stored as the identifier name in the EVALDLSS symbol table.

Syntax Analysis. The syntax analysis phase generates a parse tree using the facilities of YACC. Each major production in Figure 10-8 is augmented with semantic rules[6] for generating a parse tree node. The resulting grammar is processed by YACC to produce a compiler that generates a parse tree, which subsequently is the input to the code generator.

Additional semantic checking is also done during the syntax analysis phase. As code is generated to put system function parameters on the stack, their type is compared with the required type for that position's parameter. Similarly, the type of each Boolean or comparison operator is validated to ensure Boolean or string types. If type checking fails, the compile fails and the program aborts. In addition, the syntax checking phase makes sure that limits are not exceeded for the number of instructions on the stack or the number of local variables; if the limits are exceeded, the compilation aborts.

The semicolon following a condition expression is treated as a label for P-Code branching on true, and the semicolon after a transform as a branching label for false. In this way, during evaluation of the P-Code, the translator executes the transform code if the condition is true but branches to the next condition or the end of the P-Code if the condition is false.

---

[5]This collapsing does not take place for D2TRANS entries. Each D2TRANS codEval subfield points to one condition-transformation pair (each member of which can be null).

[6]The parse tree is generated in post order so that infix expressions (e.g., 9–5 + 2) are converted to postfix (e.g., 952 + –).

```
program                 ::=  case_block '.'
case_block              ::=  case_expr | case_vlock ';' case_expr
case_expr               ::=  condition ':' expr_list
expr_list               ::=  expr | expr_list ',' expr
expr                    ::=  'NOT' expr | '(' expr ')' | variable | INTEGER |
func_call
                             | user_call|REGEX|STRING_LITERAL|'INC' '(' expr ','

                             | compare_expr | bool_expr | arith_expr | assign_expr
```

[Note: In the previous line, the expressions are interpreted as a single term within a larger expression.]

```
STRING_LITERAL          ::=  '"' (\. | [^\"])* '"'
REGEX                   ::=  '/' (\. | [^/])*
user_call               ::=  USER_NAME maybeparen
maybeparen              ::=  '(' ')' | ϵ
USER_NAME               ::=  'DLSS.' letter (letter | digit | national)*
national                ::=  [#$%?_]
digit                   ::=  [0-9]
Letter                  ::=  [a-zA-Z]
condition               ::=  'CASE' expr_list | 'DEFAULT'
param_list              ::=  nonull_param_list | ϵ
nonull_param_list       ::=  expr | nonull_param_list ',' expr
compare_expr            ::=  expr '=' expr | expr '<>' expr
bool_expr               ::=  expr 'OR' expr | expr 'AND' expr
arith_expr              ::=  exp '+' expr | expr '-' expr
assign_expr             ::=  expr ':=' expr
func_call               ::=  SYSTEM_NAME '(' param_list ')'
variable                ::=  SYSTEM_NAME
SYSTEM_NAME             ::=  letter (letter | digit | national)*
```

FIG. 10-8. TRANSLOG GRAMMAR

Code Generator. The parse tree generated by the syntax analyzer converts to P-Code on the basis of the semantic action directives added to the TransLog productions. An array of buffers is built to hold the resulting code for all symbols. When code generation for a symbol is complete, the code is copied to a buffer, referenced by the symbol's codEval bucket.

The code generator passes the parse tree in a post-order traversal, emitting code for the major productions defined in the TransLog grammar. Generally, string or integer expressions determine whether (1) parameters for a system function call, (2) operands of a comparison, or (3) assignment operators are pushed onto the stack.

Expressions cause labels to be back-fitted and operator code to be generated. A feature of the compiled code is that a comma in an expression list pops all stack values off the stack. Thus, expression values appearing before a comma are completely independent of those appearing after it. The execution engine is responsible for popping values off the stack when a system function is called.

Execution Engine. The execution engine evaluates condition-transformation pairs in the EVALDLSS and DLSS2EDI tables. It is called with the C function

void vExecute (register PCODE pcCode).

It may be called (1) directly from the translator, when the translator is navigating through the DLSS2EDI table or (2) recursively from P-Code, when an expression refers to an EVALDLSS symbol. It returns (via the execution stack) integer, string, or pointer to regular expression code, depending on the nature of the expression evaluated.

At the top-level call to vExecute, there is a global execution stack (as explained earlier under the topic of Data Structures) that is visible to all recursive functions of vExecute and to the code associated with system symbols. The evaluation uses this global execution stack to execute string compares and Boolean operations according to the P-Code. Code for calls to system functions is generated so that the correct number and types of parameters for the function are placed on the stack.

Producing Interleaved Output. The correct order of segments in an output EDI transaction is not necessarily the same as the order of the segments in the DLSS2EDI table. Information about each EDI segment produced is kept in the structure garecEdiSeg2FileMap:

```
struct  tagEDI_FILE_MAP
{
WORD            wCreationOrder;         /*  Order in which created          */
WORD            wOutputSection;         /*  Composite loop indicator        */
WORD            wL1;                    /*  Outer loop counter              */
WORD            wL2;                    /*  Inner loop counter              */
long            lLoc;                   /*  Offset position in file         */
} *garecEdiSeg2FileMap;
```

(The fields wL1 and wL2 are not used in the current system.) Each EDI segment created during translation of a transaction is written to a sort file. For each of these segments, a garecEdiSeg2FileMap entry is updated to reflect sort

information and sort file positions. After the last transaction segment has been created and written to the sort file, the set of segments making up the transaction is sorted by output section and creation order (within output section) and written to the EDI output file.

Shutdown Procedures

Upon receipt of a SHUTDOWN message from SB, the D2E translation module follows the standard prototype LGN shutdown procedures.

Serialization

Since this module responds to the TRANSDLSS message in particular, and the TRANSDLSS message is sent only by the LAN dequeuing module after that module has filtered a file of DLSS transactions, the DLSS-to-EDI translation module could execute serially after the LAN dequeuing module. This arrangement would not be suitable for a true transaction-processing environment.

Files

The D2E translator uses several files during the course of processing:

- Translation tables
  - DIC2TID — Maps from a regular expression describing a set of DICs to the matching TID section. This is the first table referenced for any transaction translation.
  - EVALDLSS — Defines how values for logical names are derived in a DLSS transaction.
  - DLSS2EDI — Contains conditions and instructions for building an EDI transaction, one element at a time, for all segments making up the transaction. Refers, in the conditions and transformation of EDI elements, to the logical names in the EVALDLSS table.
  - CODEMAP — Provides table look-up for certain elements, by mapping between DLSS field values and corresponding EDI element values. Unlike the other tables, CODEMAP is currently not read at module start-up, but is referenced as needed during translation.

- Text files

  - DLSS Input — A file of filtered DLSS transactions, referenced in the TRANSDLSS message from the LAN dequeuing module. The complete path name for the file is C:\LD\VAL\<time-stamp>.VAL, where <time-stamp> is the timestamp assigned to the file when it was downloaded by the local interface module.

  - EDI Output — The file of EDI transactions resulting from the transaction. The full path name is C:\DE\VAL\<time-stamp>.VAL.

  - Error File — Any transactions that the module is unable to translate are written to the error file. For each transaction in the file, all segments up to the one in error are written, along with an error message and a description of the EVALDLSS symbol that was most recently evaluated. The full path name for the error file is C:\DE\ERR\<time-stamp>.ERR.

Alternative Designs

The present design of the translator has created a reliable module with more than adequate performance for the prototype system. The fact that translation rules are compiled at run time yet reside in a full-featured data base management system (DBMS) allows ample translation speed and reasonable ease in changing translation rules. Results of field testing the translator during the course of the Phase III test prompt the following suggestions for design modifications and alternative implementations:

- *True transaction processing.* This would not require a significantly large change to the translator. Translation strategy would not need to change but would just be exercised on a single-transaction basis instead of a file of transactions.

- *Hard-code constant translation rules.* Moving some or all of the translation rules into the actual C-language program could optimize performance and eliminate the amount of compilation required during program start-up. On the other hand, the added value and ease of use of the DBMS facilities would be lost.

- *Object-oriented constructs.* Given the number of aggregate data types explicitly defined within the module and implied[7] by the data, the translator would appear to be a good candidate for object-oriented methodologies. In

[7]This is especially the case for EDI formats, where transactions are built up hierarchically from elements, segments, and loops.

the same vein, the EVALDLSS table is essentially a table of objects and associated rules on how values for the objects are formed. Using object-oriented constructs might also cut down on the profusion of system functions designed to handle unique combinations of data formats.

WAN QUEUING MODULE

Purpose and Description

The WAN queuing module responds to DISPATCHEDI messages from the D2E translator module. It compresses the EDI file named in the message and sends a SENDEDI message to the WAN interface module. The compression is done via a separate batch file process (referred to as the ARC process) that invokes a PKARC compression program. The context for the WAN queuing module is shown in Figure 10-9.

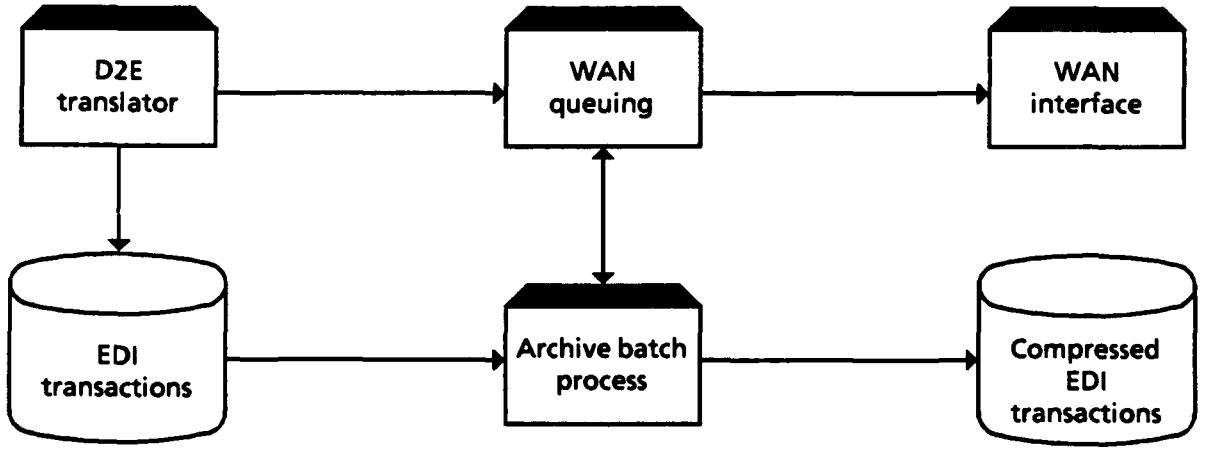


FIG. 10-9. WAN QUEUING MODULE

Start-Up Procedures

This module performs the normal module start-up sequence and some additional steps in preparation for invoking the ARC process during the Operational State. The parameters in Table 10-3, which does not include standard systemwide parameters, are read during start-up.

As part of its start-up procedure, the WAN queuing module forms the PIF name for the ARC process by concatenating (1) the PIF directory SY_PIFDIR, (2) the ARC process mnemonic WQ_ARCMNEM, and (3) the character constant "PIF.DVP."

TABLE 10-3

WAN QUEUING START-UP PARAMETERS

Parameter	Description
DE_VALROOTDIR	Directory where outbound EDI files reside
SY_PIFDIR	Directory containing PIFs
WQ_QUEUEDIR	Directory where outbound compressed EDI files are stored
WD_STAGE1DIR	Directory (on remote machine) for inbound EDI compressed files
WQ_ARCMNEM	Two-letter mnemonic for ARC process (used in building PIF name)

Next, the module reads the PIF contents into memory and overwrites a few of the PIF fields to control the ARC process screen behavior.

The module uses the expedited mailbox for normal message processing and the system mailbox for receiving SB messages. In addition, a holding mailbox, WQ_HOLD, stores messages to read while the main event-handling process awaits completion of the ARC process.

Processing

Frequency of Operation

The main process of the WAN queuing module is always running. Normally, it is in a wait state, awaiting the arrival of a DISPATCHEDI message in the IPC queue. The ARC process, invoked in response to the DISPATCHEDI message, compresses the file, notifies the main process that it has done so, and quits. It is invoked as a new, temporary process each time it is needed.

Flow of Processing

As noted above, the program is normally in a wait state, awaiting arrival of a DISPATCHEDI message in the IPC queue. When a DISPATCHEDI message does occur in the queue, the module confirms that the file referenced in the message exists. Then, by concatenating a base-36 timestamp, a one-digit qualifier, a period, and the

three-letter mnemonic for the LGN, it forms a unique name[8] for the file to be formed by the ARC process. The combination of the LGN-unique timestamp and the LGN qualifier, which is unique across LGNs, precludes receiving two EDI ARC files with the same file name.[9]

After the ARC file name is formed, the main process invokes the ARC process to compress the file. The ARC process is a batch file containing the PKARC program, followed by a C-language program that sends an ARCDONE message to the main process. The ARCDONE message contains a return code indicating the success or failure of the PKARC execution. The return code is based on the DOS error level set by PKARC.

Once the ARCDONE message is received, the module builds a SENDEDI message and sends it to the WAN interface module. The SENDEDI message contains the full path name of the ARC file to be sent and the destination path name for the file when it is received by the remote LGN.

While the main process awaits an ARCDONE message, it moves any other message received during that time to a holding mailbox. All messages sent to the holding mailbox are removed from the IPC queue by the process; if they were not, the queue would report a message in the holding mailbox and the program would keep reading the same messages over and over. When the process resumes servicing the IPC queue after receiving and processing the ARCDONE message, it checks for messages in the holding mailbox first.[10]

---

[8]Note that an ARC file consists of one or more compressed files that are bundled into one ARC file. The file names of the compressed files are retained, but the ARC file has a name of its own, not related to the names of the embedded compressed files.

[9]What about duplicate EDI translation file names? Since EDI file names are not qualified by an LGN name extension, is it possible for an LGN to receive two EDI ARC files, each containing a compressed file identically named? Currently, the answer is "yes," because of name-length restrictions. In the prototype system, a work-around was devised, using an additional qualifier, to ensure that all received EDI files have unique names. This should not be a problem in a production system running under UNIX, with its longer file names.

[10]This approach would be unsuitable for a production LGN. For instance, what happens if the ARC process hangs and an ARCDONE message is never sent to the main process? During the prototype system test, this has not occurred, but a production solution should be devised to preclude such occurrences.

Data Structures

The PIF structure (Figure 7-4) is used to invoke the ARC process. No other data structures are maintained by the WAN queuing module.

Shutdown Procedures

Upon receipt of a SHUTDOWN message from SB, the WAN queuing module follows the standard prototype LGN shutdown procedures.

Serialization

The WAN queuing module is an intermediary between the D2E translator module and the WAN interface module, called on to perform only after the D2E translation module has translated a file of DLSS transactions. The module is suitable to be invoked serially after the D2E translator. In this scenario, it would still run in parallel with the WAN interface module.

Files

The WAN queuing module takes an input file of valid EDI transactions and produces an output ARC file containing the EDI file in compressed form.

Alternative Designs

The present design of the WAN queuing module has worked satisfactorily for the prototype, file-based system. For a production, transaction-based system, the WAN queuing module must bundle transactions with common priorities and destinations. The bundled files will be compressed and a message sent to the WAN interface module, bearing in mind that high-priority transactions cannot be detained by lower priority transactions. As indicated by the processing flow, the module must be capable of receiving new transactions concurrently while compressing others.

WAN DEQUEUING MODULE

Purpose and Description

In the receiving LGN, this module is a peer process of the sender's WAN queuing module. Its primary job is to expand (decompress) EDI files received by the WAN interface module and notify the EDI-to-DLSS translation module that the expanded file is ready for translation. In many respects its processing is analogous to

that of the WAN queuing module. The context for the WAN dequeuing module is shown in Figure 10-10.

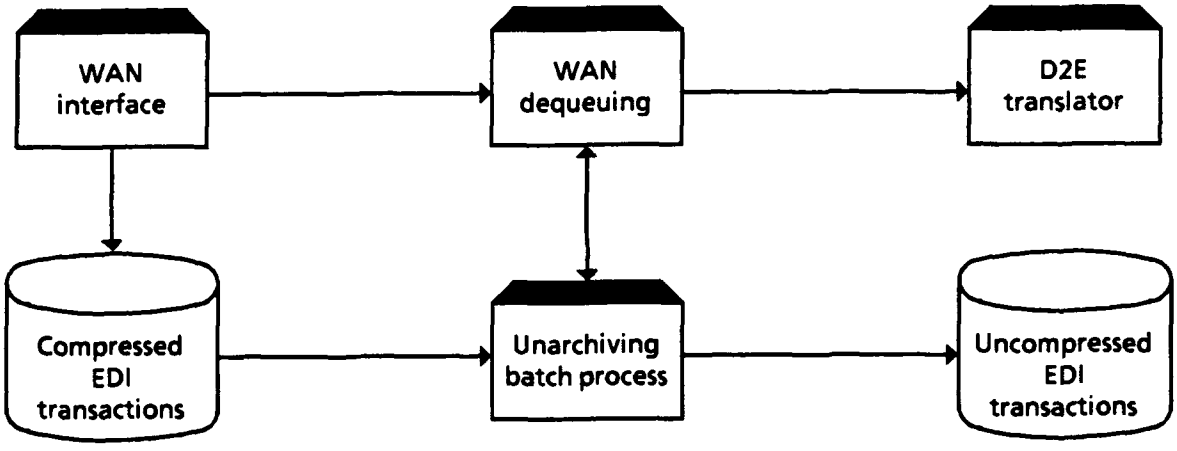


FIG. 10-10. WAN DEQUEUING MODULE

Start-Up Procedures

The WAN dequeuing module's start-up procedures are very similar to those of the WAN queuing module. In addition to the standard start-up sequence, the module forms a PIF name for the unarchiving process, reads into memory the PIF by that name, and overwrites certain fields within the PIF structure.

The expedited mailbox is used by the module for normal message processing. The system mailbox receives SB messages. A holding mailbox, WD_HOLD, holds messages received during file expansion by the unarchiving process.

During start-up, the module reads the parameters listed in Table 10-4 (which does not include standard systemwide parameters).

Processing

Frequency of Operation

This process is always running. Normally, it is in a wait state, awaiting the arrival of an RCVDEDI message from the WAN interface module.

TABLE 10-4

WAN DEQUEUING START-UP PARAMETERS

Parameter	Description
SY_PIFDIR	Directory containing PIFs
WQ_QUEUEDIR	Directory where outbound compressed EDI files are stored
WD_STAGE2DIR	Directory for inbound EDI compressed files
WQ_ARCMNEM	Two-letter mnemonic for ARC process (used in building PIF name)

Flow of Processing

The main work of the WAN dequeuing module is to await an RCVDEDI message pointing to an ARC file, expand the component EDI file, and notify the E2D translator module via a TRANSEDI message. Upon detection of an RCVDEDI message in the IPC queue, the module invokes the unarchiving process through a PIF structure. Like the ARC process in the WAN queuing module, the unarchiving process is a batch file that calls the program PKXARC. Following PKXARC in the batch file is a small C-language program that sends a message to the main process indicating that the file expansion has been completed.

Although in the prototype implementation there is just one EDI file per ARC file, the WAN dequeuing module does not make this assumption. For each expanded EDI file, the main process forms and sends a TRANSEDI message to the EDI-to-DLSS translation module. The module deletes the ARC file after processing it. Operating in a fashion similar to the WAN queuing module, the WAN dequeuing main process transfers all incoming messages to the holding mailbox during its wait for an UNARCDONE message from the unarchiving process. After processing the UNARCDONE message, when it returns to servicing the IPC queue, it checks the holding mailbox first for messages.

Data Structures

The PIF structure (Figure 7-4) is used to invoke the ARC process. No other data structures are maintained by the WAN dequeuing module.

Shutdown Procedures

Upon receiving a SHUTDOWN message from SB, the WAN dequeuing module follows the usual prototype LGN shutdown procedures.

Serialization

The WAN dequeuing module could run as a transient, serial module, invoked after receipt of an EDI ARC file.

Files

The module processes incoming ARC files containing EDI files in compressed form. Its output is the expanded EDI file, placed in the directory C:\WD\QUEUE.

Alternative Designs

For a transaction-based production system, the WAN dequeuing module would be required to recognize and read, one at a time, the transactions making up the EDI files and send the proper message to the E2D translator.

EDI-TO-DLSS TRANSLATION MODULE

Purpose and Description

In a broad sense, the EDI-to-DLSS translation module (E2D translator) is a mirror image of the D2E translator; its operating context is shown in Figure 10-11. This module translates EDI transactions, a file at a time, into their equivalent DLSS formats. The translation is triggered by the reception of a TRANSEDI message in the IPC queue. As in the D2E module, the translation is driven by tables. The purpose of the tables in the two translators is approximately the same, although their makeup and use differ. The E2D translator uses the same TransLog grammar and P-Code compiler as its D2E counterpart. Likewise, the E2D translator is a single C-language program. In many other respects, the two translators closely parallel one another. For these reasons, processes in the E2D translator that have already been covered in detail in the DLSS-to-EDI Translation Module section, will not be revisited here, except to highlight differences in design or operation.

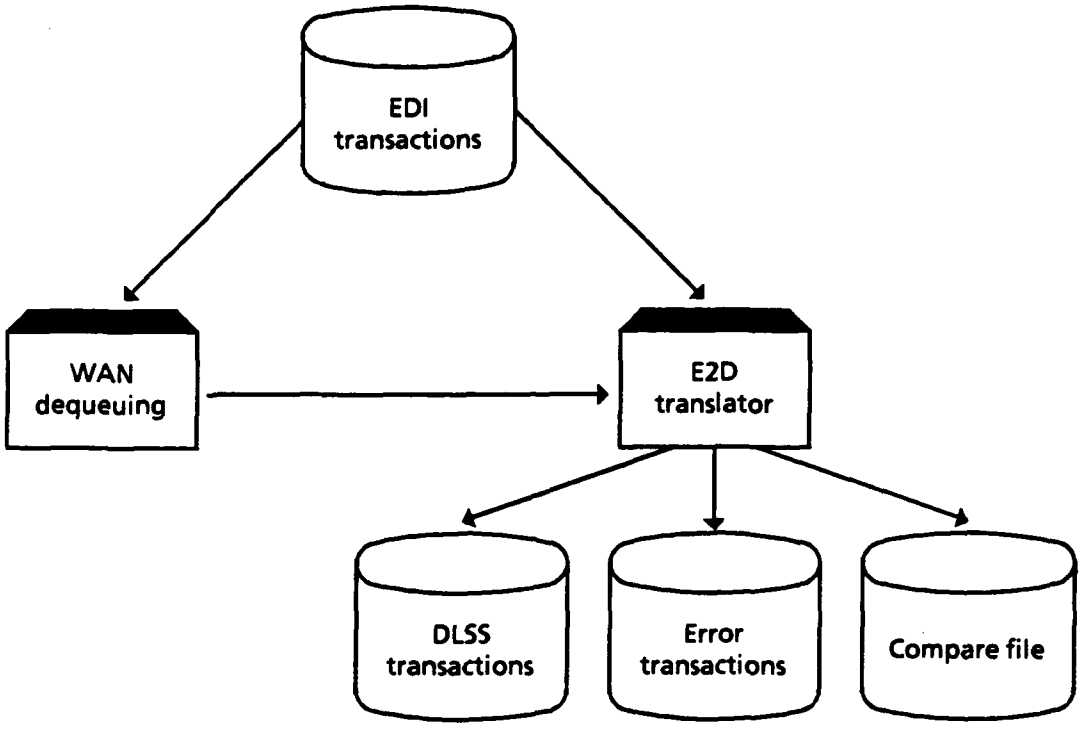


FIG. 10-11. MODULE FOR EDI-TO-DLSS TRANSLATION

Start-Up Procedures

The E2D module start-up procedures are comparable to those for the D2E translator, as is the time required for initialization. As in the D2E module, no module-specific configuration parameters are read during start-up; all information required for operation (1) is obtained through message parameters, (2) is hard-coded, or (3) is implied.

Processing

Predominantly, the E2D module is parked in the IPC queue in expectation of a TRANSEDI message or a system message. Receiving a TRANSEDI message causes the module to start a translation of an EDI file; again, translation is driven by the translation tables that were read and compiled during start-up.

Frequency of Operation

This module is always running. During the Operational State, it is dormant until an event is detected on the IPC queue.

Flow of Processing

The most significant message for the E2D module is the TRANSEDI message sent by the WAN dequeuing module. The TRANSEDI message specifies the EDI file to translate. EDI transactions are read one at a time from the input file. The program determines the TID of the transaction from its start (ST) segment, builds a template of segments and elements in memory, verifies that no required segments are missing from the transaction, and determines the starting record number of the EDI2DLSS table to use for the translation.

The module navigates through the EDI2DLSS table records associated with a transaction's TID, one record at a time. The EDI2DLSS table is analogous to the DLSS2EDI table: it contains conditions and evaluation rules for each field in the transaction. Each record in the table indicates a condition for adding a field to a DLSS transaction and the required evaluation to produce the correct field value. The evaluation is a reference either to an EDI symbol (in the EVALEDI table) or to a TransLog expression. Each field may have more than one condition associated with it, but one condition per EDI2DLSS record. The program continues reading records[11] for a field until encountering a true condition or end of records. The EDI2DLSS table does not have an explicit column designating an output DLSS transaction field; rather, DLSS fields are implicitly defined by starting column and length. Since DLSS transactions can contain multiple card images, the record also indicates the card-image number for the transaction to which the field is written. Once a condition for a field is evaluated as true, the program evaluates the EDI2DLSS table's [CardNo] field. That field is written to the appropriate card image and column positions. After writing the DLSS field, the program reads EDI2DLSS records until

[11]As is the case with the D2E translator, keep in mind that "records" is a figurative term; it actually refers to P-Code record images.

encountering a start column for the DLSS field at least one greater than the last column written, or until there are no more records for the current TID.[12]

As a validation measure in the prototype system, the original DLSS transaction is included in the EDI transaction as a series of "XXX" segments, one segment per DLSS card image. After translating from EDI to DLSS format, the E2D translator compares the XXX segments to the DLSS transaction just produced. Any discrepancies in field values or number of card images are identified and written to a compare file. If an error occurs that prevents completion of translation, an appropriate message and a list of the segments translated up to the point of the error are written to an error file.

For remote LGNs, no message results from translating a file of EDI transactions into DLSS format. Since, in the prototype, translated DLSS data are not uploaded to the host, this marks the last LGN processing step for a file. The CLGN implementation is handled differently, as will be discussed.

SUSPEND messages are handled by the E2D module in the same way that the D2E translator handles them. Upon receiving a RESTART message, the module frees all memory allocated for internal translation table representations and re-initializes.

Data Structures

The E2D translator manipulates a set of data structures comparable to those of the D2E translator: symbol tables, parse trees, the evaluation environment, and paging tables. Because the data structures are so similar, only these aspects of the E2D data that are different from the D2E environment are stressed.

The E2D module maintains two symbol tables: an EDI evaluation table and the system symbol table. The system symbol table is created and used as for the D2E

[12]There are some exceptions to this searching criterion. For example, an EDI2DLSS record may exist only to generate a side effect. As in the D2E translator, a side effect generates a system function or assigns a value to a user-defined variable but does not affect any particular output field. Side-effect records always have a value of zero in the (length) field. If a side effect results in generating a new DLSS card image, the DLSS card image position is assumed to be zero. This ensures that the next EDI2DLSS record pertaining to a DLSS field will be read (not skipped), since the start column for a field specified in the table is always greater than zero.

translator. The set of system symbols for the E2D translator is somewhat different, as shown in Appendix B.

The EDI evaluation table is the internal representation of the translation table EVALEDI; its purpose is analogous to the DLSS symbol table in the D2E translator. The C-structure of an EDI symbol table entry is shown in Figure 10-12.[13] Whereas the DLSS symbol table is basically a dictionary with instructions on how to calculate the value for each field, the EDI symbol table primarily specifies where to find the field in an EDI transaction. In contrast to the D2E symbol table, the P-Code for an EDI symbol entry indicates only how to extract the EDI field. The wSegSeq and wEltSeq subfields point to the segment and element (within the segment); the evaluation of the codUseCtr subfield yields the segment occurrence number[14] to which the other subfields refer.

```
typedef   struct tagEDI_SYMBOL
{
 char     *pszName;     /*  Symbol name(composite)        */
 WORD     wSegSwq;      /*  Segment sequencenumber        */
 PCODE    codUseCtr;    /*  P-Code for usecounter         */
 WORD     wEltSeq;      /*  Element # in segment          */
} EDI_SYMBOL;
```

FIG. 10-12. EDI SYMBOL TABLE ENTRY

Figure 10-13 shows the aggregate data structures associated with an EDI2DLSS table entry. The one-to-many relationship between the EDI2DLSS and E2DTRANS arrays is analogous to the DLSS2EDI-D2ETRANS relationship in the D2E translator. The subfield *precTranslation in each EDI2DLSS entry points to a set of E2DTRANS entries representing condition-transformation pairs for all fields in the TID section named. In addition, the subfield *precEdiSegs points to an array of EDISEGS entries associated with the EDI2DLSS record; the EDISEGS entries define the EDI segments that make up the section and their maximum allowable number of occurrences. For the current TID section, the EDI2DLSS entry also points to the

[13]The EXTERN keyword in Figures 10-6 and 10-13 is defined in the translator C-language program as the word "external" for all functions except the one in which the structure is defined; in that case, it is defined as the null string. This allows the function defining the structure to allocate space implicitly for the structure and all other functions to share the same "include" file to refer to the structure.

[14]There may be more than one instance of a segment in an EDI transaction.

parent, next sibling, and child sections.[15] These pointers are used as navigation guides during a translation. Figure 10-14 shows the relationships among the EDI2DLSS, EDISEGS, and E2DTRANS tables.

```
typedef   enum  tagE2DTRANS
{
WORD      wRecNo;          /*  Record # in EDI2DLSS                    */
WORD      wStart;          /*  Start position in card                  */
WORD      wLength;         /*  Number of columns in card               */
PCODE     pcCode;          /*  Code to determine value                 */
PCODE     pcCardNoCode;    /*  P-Code to determine card number         */
} E2DTRANS

typedef   struct  tagEDISEGS
{
char      aszQualName[kwTR_MaxQualNameSz];
                           /*  Composite name for table entry          */
WORD      wSegSeq;         /*  Segment sequence number                 */
WORD      wUseCtr;         /*  Segment use counter                     */
char      aszSegId[4];     /*  Segment ID                              */
BOOL      bRequired;       /*  Segment required?                       */
} EDISEGS;

EXTERN    struct  tagEDI2DLSS
{
WORD                       wTid;              /*  Transaction ID       */
WORD                       wTidSect;          /*  TID section          */
Struct    tagED2DLSS       *precParentSect;   /*  Parent loop pointer  */
struct    tagED2DLSS       *precRtSibSect;    /*  Right-hand sibling pointer */
struct    tagED2DLSS       *precFirstChildSect; /*  First child loop pointer */
WORD                       wNumTrans;         /*  Number of E2DTRANS   */
E2DTRANS                   *precTranslation;  /*  Set of transaction records */
WORD                       wNumSegs;          /*  Number of EDISEGS    */
EDISEGS                    *precEdiSegs;      /*  Segment template set */
WORD                       wSectOccurrences;  /*  # Occurrences encountered */
WORD                       wMaxSectOccurrences; /*  Max # occur. for section */
} garecEdi2Dlss[kwTR_MaxNumTidSects];
```

FIG. 10-13. EDI2DLSS TABLE ENTITIES

The E2D translator uses paging tables in a manner similar to their use by the D2E translator, the main difference being that EDI segments, rather than DLSS card images, are paged in.

[15]For instance, if the TID is 568 and the TID section is 110, the parent section is 100, the next sibling section is 120, and there is no child section (there are no lower level sections).

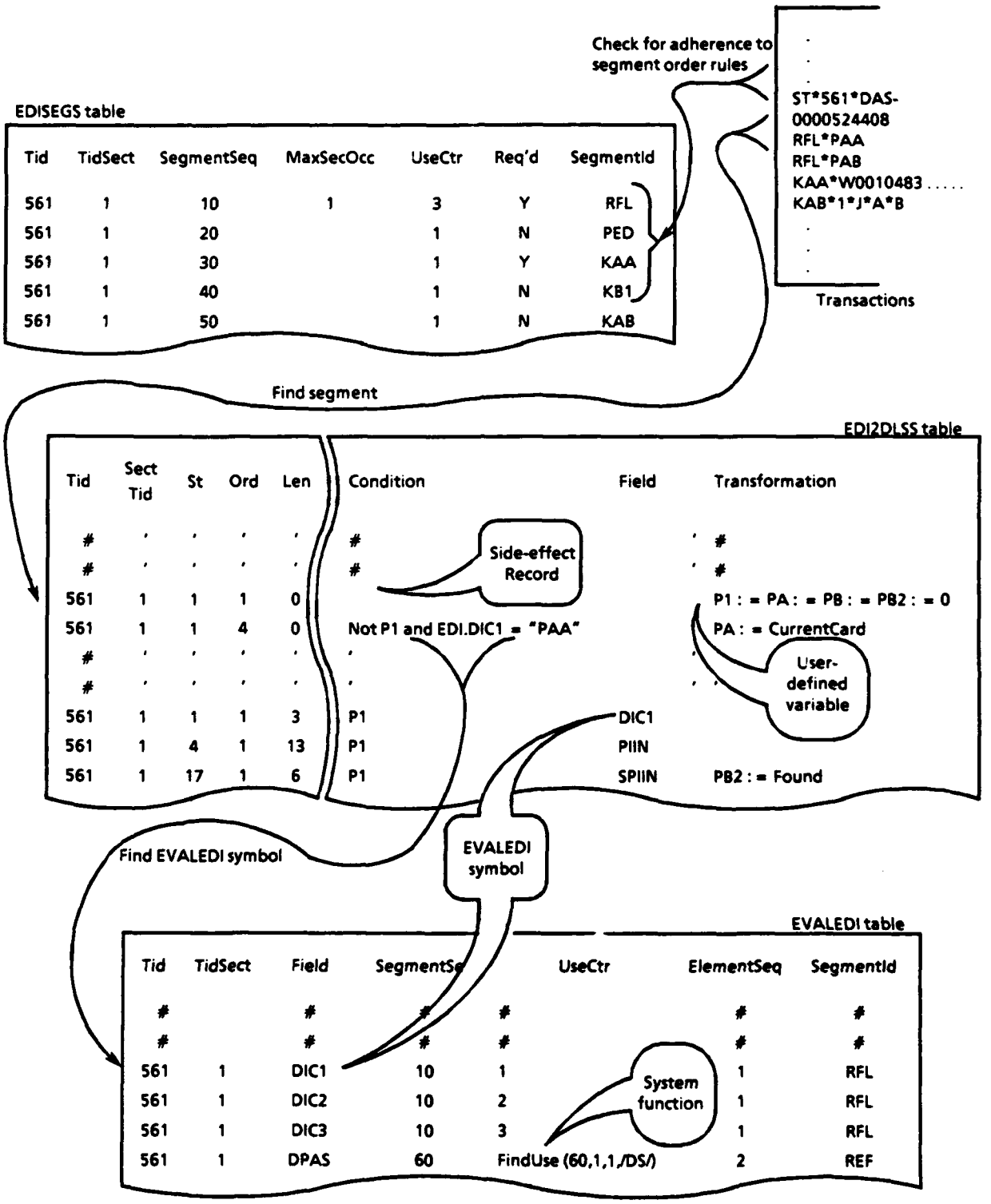


FIG. 10-14. TABLE INTERACTION FOR EDI-TO-DLSS TRANSLATION

Algorithms

The significant algorithms used in the E2D translator (P-Code compilation and evaluation, segment paging) are similar enough to those used by the D2E translator so that a separate discussion here is unnecessary.

Differences Between LGN and CLGN Implementations

The prototype CLGN models some key attributes of the production CLGN that will reside at DAASO (see Chapter 2); specifically, after receiving EDI transactions and translating them into DLSS format, it retranslates the DLSS transactions back to EDI. Thus, the EDI module in the CLGN, upon completing the translation of a file of EDI transactions, sends a RCVDDLSS message to the LAN dequeuing module, as if a DLSS file had been downloaded from the host. In the prototype case, however, the DLSS file to be filtered is the output of the E2D translator, as specified by the RCVDDLSS message.

Shutdown Procedures

Upon receipt of a SHUTDOWN message, the E2D module follows the standard prototype LGN Shutdown State procedures.

Serialization

The E2D module could be positioned to run serially after the WAN dequeuing module. However, this approach would not be suitable for a transaction-processing environment.

Files

The following files are central to the processing of the E2D translator:

- Translation tables
  - EVALEDI — Defines how values for EDI logical names are derived from an EDI transaction.
  - EDI2DLSS — Describes, for each TID section, the conditions for inclusion and transformation of each candidate field making up a DLSS transaction. References to the EVALEDI table are made liberally.

- EDISEGS — Contains the set of segments that make up an EDI transaction and their respective attributes.
- CODEMAP — Provides table look-up for certain elements, by mapping between DLSS field values and corresponding EDI element values. Unlike the other tables, CODEMAP is currently not read at module start-up but is referenced as needed during a translation.

- Text files
  - EDI Input — A file of EDI transactions, pointed to by the TRANSEDI message. Analogous to D2E's DLSS input file.
  - DLSS Output — The primary output of the E2D translator. A file of DLSS transactions having the same base name (timestamp) as the EDI input file and the file extension .VAL.
  - Compare File — A file containing transactions that were translated error-free but that differ from the XXX segment(s) of the source EDI transaction. For each transaction that has a compare discrepancy, the file lists the original DLSS card images, the EDI segments, and the retranslated DLSS card images, along with an illustration of the differences.
  - Error File — A file containing EDI transactions and partially translated DLSS transactions for each transaction that was halted because of an error. The full path name of the file is C:\ED\ERR\<time-stamp>.ERR.

LAN DEQUEUING

This module is not included in the prototype system.

CHAPTER 11

OPERATIONS SUBSYSTEM

The operations subsystem in the Phase III prototype LGN consists of three modules:

- System boot (SB)
- System monitor (SM)
- System utilities (SU).

SB is the entry point of the subsystem and also of the LGN. It keeps track of the operational status of all other LGN modules. During LGN start-up, SB invokes the other two operations subsystem modules, SM and SU. The overall structure and context of the subsystem are shown in Figure 11-1.

SYSTEM BOOT MODULE

Purpose and Description

The SB module starts all the LGN processes that run continuously. Transient processes are started by their respective parent modules when required. The SB module invokes the overall LGN processing environment and ensures that all necessary modules are started. It subsequently acts as the operator console until the system is brought down.

Boot Procedure

The SB module starts automatically upon LGN start-up, via the Autoexec.Bat file. The Autoexec.Bat file starts DESQview, which checks its script file, Desqview.Dvs, for the presence of a script that is run automatically whenever DESQview is invoked. On the prototype LGN, it is the !Auto script that starts the SB module. Alternatively, the module can be invoked from the DESQview "Open Window" menu (option SB in the prototype).

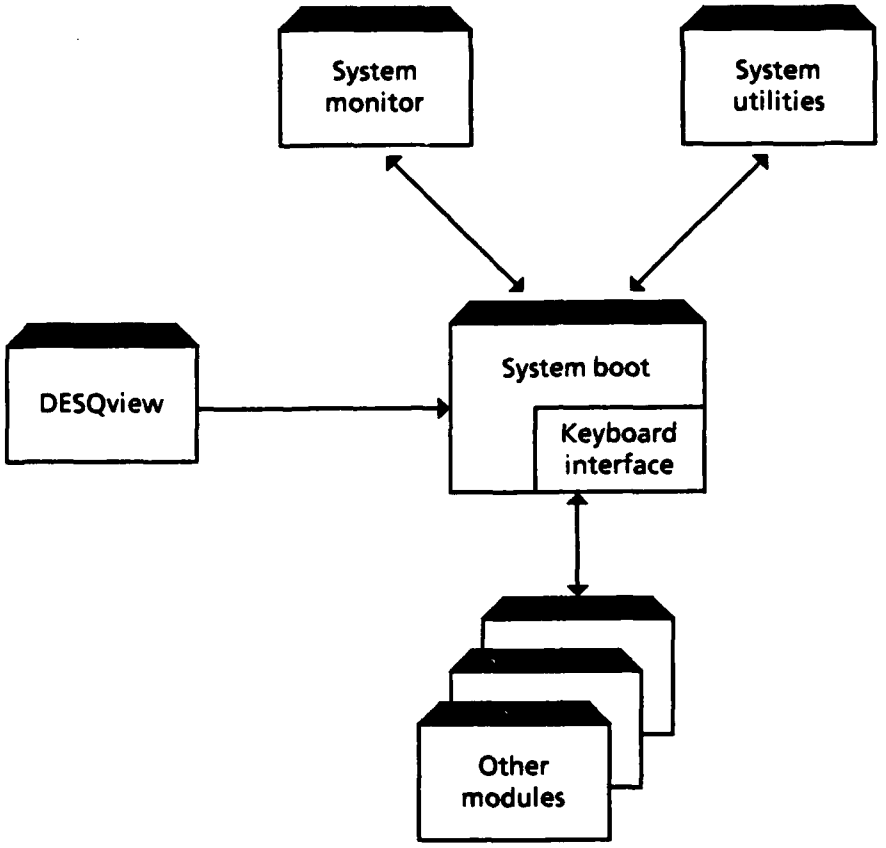


FIG. 11-1. OPERATIONS SUBSYSTEM

Processing

Frequency of Operation

The SB module is executed once when the LGN is powered up and again after a general application software failure or operating system warm boot. The module cannot be invoked while the LGN application software is running. As part of its start-up procedure, the module checks for an existing copy of its expedited mailbox; if it finds one, that indicates that a copy of the module is already active and the (redundant) module terminates with an error. Other LGN modules, however, can be shut down and restarted during operation of the LGN. The SB module is always the vehicle by which this is done.

Flow of Processing

The SB module relies on the computer's clock being set to the correct date and time. In the production LGN, CLGN clock synchronization, operator confirmation, and other programmatic methods should be used in order to verify that the clock is set correctly. This feature was removed during prototype development because of the inconvenience of entering the date and time whenever the LGN software is restarted, a frequent occurrence during program development.

The first two steps for the module are reading the LGN.CFG file to obtain systemwide parameters and reading the BOOT.TAB file, which contains a list of all modules to be started by SB. SB maintains a linked list of records for each valid entry found in the BOOT.TAB file. This list is a series of entries of the type BOOT_ENTRY, shown in Figure 11-2. Every line in the BOOT.TAB file corresponds to an entry in the linked list as well as to a DESQview task that is created by SB, except for lines that begin with"#"; those are considered to be comments.

```
/* One BOOT_ENTRY entry for each process started by System Boot                */
typedef struct  BOOT_ENTRYtag
{
char              sModule[SY_MODMNEMLN + 1];        /*  Module mnemonic      */
char              sPath[SY_FILENMLN + 1];     /*  Module path                 */
time_t            sStarted;                   /*  Module start time           */
DV_APIHANDLE      HProc_hdl;                  /*  Module handle               */
DV_APIHANDLE      hSy_mbox;                   /*  System mailbox handle       */
DV_APIHANDLE      hExp_mbox;                  /*  Expedited mailbox handle    */
DV_APIHANDLE      hLow_mbox;                  /*  Low-priority mailbox handle */
DV_APIHANDLE      hMailbox[MAXNOFMAILBOX];    /*  Other mailbox handles       */
TBOOL             bSuspendable;               /*  Can be suspended?           */
short             shMboxct;                   /*  Number of "other" mailboxes */
time_t            tAcknowledged;              /*  Time SB rcvd. STARTUP msg.  */
char              sErrortxt[SY_ERRMSGLN + 1]; /*  Error number reported       */
short             shError;                    /*  Error message               */
BYTE              byStatus;                   /*  Current module status       */
time_t            tLast_shdown;               /*  Time of last shutdown       */
time_t            tLast_suspnd;               /*  Time of last suspend        */
struct BOOT_ENTRYtag      *dpNext;            /*  Pointer to next entry       */
}BOOT_ENTRY;
```

FIG. 11-2. BOOT_ENTRY PROCESS

All modules are invoked using the DESQview PIF structure and app__start API function. For each module table entry, SB reads a corresponding PIF of the form:

<Drive>:<PIF-Dir>\<Module-Mnemonic><PIF-Suffix>

where:

- Drive is the systemwide parameter specifying the letter of the drive where the system software is located.
- PIF-Dir is the systemwide parameter specifying the directory containing the PIFs for LGN processes.
- Module-Mnemonic is the two-letter module mnemonic as it appears in the BOOT.TAB file.
- PIF-Suffix is a systemwide constant specifying the right-hand portion of all PIF names ("-PIF.DVP").

The PIF is read into a memory image in SB's data space. The drive and directory fields of the PIF image are overwritten with the drive and directory read from the BOOT.TAB file or, if present, the drive and directory specified in the appropriate module configuration parameter file.

SB starts DESQview processes with the DESQview API app__start function. App__start returns a handle for the process, or zero if the process cannot be started. SB displays the start-up time for each module as it is executed on the console.

Once the main processes for all LGN modules have started, SB waits for SY__LGNTIMEOUT seconds to receive a STARTED message from each process invoked. If the time expires before all processes have sent a STARTED message, or if a process sends a CANTSTART message, (1) a diagnostic is displayed on the console, (2) all processes are terminated, and (3) the LGN is shut down.

When SB receives a STARTED message from all processes, it returns an OPGO message to each, and the LGN enters the Operational State. SB shuts down the SU module, since it is needed only occasionally and can be started up and shut down as required.

The SB module's main tasks during the Operational State are (1) servicing requests from the local keyboard, (2) servicing messages from a remote LGN via the

WAN interface module, and (3) handling system messages to and from system mailboxes for other modules.

An operator communicates with an LGN via its local keyboard. SB displays a menu in the DESQview window with the following options:

- Suspend a module
- Restart a module
- Shut down a module
- Transfer a file to a remote LGN
- Update a translation table on a local or remote LGN
- Run a system utility
- Reset an LGN (shut down all LGN processes and restart the LGN)
- Request a file from a remote LGN
- Shut down an LGN.

When a menu option is entered from the keyboard, it is verified as valid by SB and transformed into a message to the SB system mailbox.

Table 11-1 shows the complete list of messages that SB services during the Operational State.

If SB determines that a request conflicts with the current status of a module (e.g., a SUSPEND request of an already suspended module), as indicated by the by Status field in that module's boot table entry, SB processes the request but issues a diagnostic warning of the conflict.

Data Structures

SB employs several data structures to store current information about LGN modules and keyboard activity. All structures listed below are defined in the include file sb_g.h.

- Boot table. One entry for each process invoked by SB. This data structure, already discussed, is shown in Figure 11-2.

TABLE 11-1

SYSTEM BOOT MESSAGES

Message name	Description
SUSPEND	Suspend a module
SUSPEND_READY	Module is about to suspend
UTIL_REQST	Perform utility (usually a batch file or executable program) via the system utilities module (which may have to be re-spawned)
SHUTDOWN_MOD	Shut down a module
MOD_SHUTDOWN	Module intention to shut down
RESTART	Restart a module
SB_RESET	Reset local LGN
LGN_SHUTDOWN	Shut down LGN
STARTED	Module has completed Initialization State procedures and is ready to make the transition to Operational State
ERROR_MSG	Display error message on local console
RELOADTRANS	Reload one or more translation tables
TABLERELOADED	Confirmation of translation table update from remote LGN
TEXTREPLY	Text reply (probably in response to query) from remote LGN
CANTSTART	Module cannot complete Initialization State tasks
PROBEQUERY or STATUSQUERY	Query from local or remote LGN; if query is directed to SB, respond; otherwise, forward query to System Monitor
PROBEQUERYRESP or STATUSQUERYRESP	Response to query; forward to System Monitor
FILERECEIVED	A remote file request has completed; display message
MSGTOOEARLY	Module received a message before receiving an OPGO message

- Structure definition to track utility requests and local table updates. This structure is passed to the SU module.

```
typedef    struct UTIL_REQSTtag
{
char       sModid[SY_MODMNEMLN + 1];      /*  Requesting module                */
char       sLgnid[SY_LGNNMLN + 1];        /*  Source LGN                       */
char       sSuspmod[SY_MODMNEMLN + 1];    /*  Modules affected (suspended)     */

/*Utility command for utility request, or table names for table reload:        */
char       sCommand[SY_MAXCOMMDLN + 1];
time_t     tDate;                         /*  Time request issued              */
short      iUtid;                         /*  Request ID                       */
short      iStatus;                       /*  Request status:                  */
                                          /*  CLEAR - available, no request    */
                                          /*  STARTED: SU started              */
}UTIL_REQUEST;
```

- Structure definition for storing operator keystrokes for current command. It is used for both local and remote LGN menu requests.

```
typedef    struct USER_COMMANDtag
{
char       sSrcIgn[SY_LGNNMLN + 1];       /* Source LGN                        */
unsigned   short   iType;                 /* Command type                      */
unsigned   short   iStep;                 /* Current step # in multi-step command */
char       sSrcmod[SY_MODMNEMLN + 1];     /* Source module                     */
char       sFor t[SY_FILENMLN];           /* Tables or files included in command */
char       sDestfile[SY_FILENMLN];        /* destination file names (for file  */
                                          /* transfer only)                    */
char       sAffect[SY_MODMNEMLN + 1];     /* Modules affected                  */
char       sDestIgn[SY_LGNNMLN + 1];      /* Command goes to this LGN          */
char       sDestmod[SY_MODMNEMLN + 1];    /*and to this this module            */
char       sSyntax[SY_MSGTXTLN];          /* Syntax of current command         */
}USER_COMMAND;
```

Differences Between LGN and CLGN Implementations

The CLGN can exchange data with any LGN; all other LGNs can exchange data only with the CLGN. This is the only difference in SB operation resulting entirely from whether or not an LGN is configured as a CLGN.

Shutdown Procedures

Shutdown procedures are similar to the processing steps previously described. Shutdown is initiated by an operator action – either locally or remotely. In the prototype configuration, only a CLGN operator can initiate the shutdown of a remote LGN. To shut down an LGN, option 3 is selected from the SB menu, causing a

SHUTDOWN message to be sent to all processes started by SB. SB then waits for a confirmation response from each affected module. When all modules have responded, or when SY_LGNTIMEOUT seconds have elapsed (the same time-out period used during initialization), the SB transaction log file SB.LOG is updated, all DESQview objects are freed, and all LGN processes are terminated.

Files

The SB module interacts with two data files: SB.LOG and APPHAN.CFG. The SB.LOG file is updated each time SB is started or shut down. Each record in the file is a tally of the total number of messages received by the module, by hour. The numbers are cumulative since the last time the file was updated.

The APPHAN.CFG file is updated during the Initialization State and each time a module is restarted. It contains the DESQview process handles of all processes started by SB. The file is overwritten each time SB is run.

Alternative Designs

In the production system, more frequent and expanded time and event recording could be added, both on the local console and in the SB.LOG file. For example, the transactions could be broken down by category, in a way similar to that used by the WAN interface module. Better verification of system time will be necessary in a production system. Ample coding, if not hardware-assisted, solutions to this particular problem are available.

SUSPEND/RESTART MODULE

Purpose and Description

Suspend/Restart is not actually a separate module but is embedded in the SB module. It makes up three commands in SB's menu: SUSPEND, RESTART, and SHUTDOWN. The functionality and operation of this module are described in the previous section on SB.

Boot Procedure

Because the suspend/restart module is not a distinct process, a discussion of the boot procedure is not applicable.

Processing

The BOOT.TAB table, read during SB start-up, contains a field in each record indicating whether or not the module associated with that record can be suspended or shut down from the SB menu. There are some exceptions. For instance, neither the WAN interface module nor the CLGN polling module can be suspended or shut down from a remote LGN. Because the necessary communication with the remote LGN would be lost, there would be no way for the remote LGN to signal the module to restart once the module was inactive.

To suspend a module, an operator at the local console or at a remote console enters the suspend option (option 1) from the SB menu. SB then prompts the operator for the two-letter module mnemonic telling the module to suspend. The mnemonic is verified in SB's boot table as valid, as is the fact that the module is not currently suspended. Also, any rules that may prevent the module from being suspended are enforced. SB then sends a SUSPEND message to the affected module and updates its status flag in the boot table.

Module restarts and shutdowns are handled in much the same way, via the Restart option (option 2) and Shutdown option (option 3) on the SB menu. The Restart option can be selected both for suspended modules and for those that have been shut down. However, the current implementation of the LGN does not fully support[1] restarting a previously shut down module. This is not an issue if the entire LGN is brought down, since, when the LGN is subsequently restarted, all modules are started fresh (in other words, they are not considered to be restarted).

Sometimes modules are suspended automatically as part of servicing a utility request or table update. The table update and system utility SB menu options (options 5 and 6) are the correct way to effect such actions.

---

[1]Recall that a module's primary mailboxes (expedited and low-priority) are created by SB during system start-up. Once the module has received the OPGO message from SB, it opens the already created mailboxes for the Operational State. When the module is shut down, those mailboxes must be closed, to ensure they can be properly recreated and reopened if the module is subsequently restarted. This aspect of module shutdown/restart has not been tested for all modules in the prototype system.

Shutdown Procedures

Since this module is not a separate executable process or a distinct process thread, a shutdown procedure is not applicable.

Serialization

Given the tight integration of module suspend/shutdown/restart functionality with the SB module as a whole, it makes sense for suspend/restart to be a subset of SB. It can remain a strictly logical subset of the SB module, or it could be packaged as a separate function called from the SB main program. In either case, it is part of the SB executable.

SYSTEM MONITOR MODULE

Purpose and Description

The SM module has two major purposes:

- Acts as the interface with other modules for status and performance queries.
- Ensures that critical messages relating to system or module outage are relayed to the CLGN.

Boot Procedure

SM is booted up as a separate process in the standard way, via SB and the BOOT.TAB file.

Processing

Frequency of Operation

The SM module is always operating, awaiting user input from the keyboard or unsolicited status messages from other modules.

Flow of Processing

As indicated, after module start-up, the SM module waits for keyboard input or a message from another module. Keyboard input can request a status or performance report on a particular module running on the local LGN. For the CLGN, keyboard input can query status or performance for any LGN in the test network.

Query requests can be directed to one of three destinations:

- The SM module on the local LGN
- Another module on the local LGN
- A module on a remote LGN (this option is available only to the CLGN).

SM maintains a table of queries received. If the number of outstanding queries exceeds available entries in the query table, SM will not be able to process the overflow. Once a query has been processed, its table entry is freed up.

If the query is for the local SM module, SM writes a report file of all outstanding and processed queries based on information in the query table. It issues a DESQview API app_gofore call and becomes the foreground process, ensuring that its window, in which the report file is viewed, will be visible to the local operator. When the report is displayed on the screen, the local user can scroll through it a line at a time.

If the query is for a local module other than SM, it is forwarded to the destination module specified in the message. In the prototype implementation, other modules do not have any functionality for responding to queries; the queries are presently ignored by modules other than SM. This deficiency requires remedy in the production LGN.

In the CLGN, queries can be destined for modules on remote LGNs. In this case, the query is bundled inside a SENDMSG message and forwarded to the WAN interface module, which sends the message to the destination LGN.

Query responses are serviced in a manner similar to the way in which query requests are serviced. Responses can refer to a previous query or can be unsolicited. Both are handled in much the same way. When SM receives a query response message, it checks the query table for response matches with unanswered queries. If it finds a match, the appropriate query table entry is updated. If the response is directed to the local SM module, or if the response is unsolicited, it is displayed on the local console. Unsolicited responses are displayed in a small window inside the regular SM window on the screen. The format of unsolicited responses is assumed by SM to be as follows (not including the line number designations):

- (Line 1) .hdr
- (Line 2) <Headline>

- (Line 3) .hdr
- (Lines 4 to n) <Response Text>.

As the local user scrolls the report, the headline stays fixed on the screen. The headline could be, for example, column headings for the text that follows. A small program logic change could make it possible for unsolicited query responses to be sent to the CLGN instead of being displayed on the local console.

Responses directed to the local SM module are displayed in the default SM window on the console. As with SM-bound queries, SM captures the foreground processing position and scrolls the response under user control.

Responses for modules on remote LGNs are packaged within a SENDMSG message and sent to the WAN interface module, where they eventually are sent to the destination LGN. The CLGN can send a response to any LGN in the test network; all other LGNs can send responses to the LGN only (presumably as a reply to a previous query from the CLGN).

Data Structures

The SM module keeps a table of queries originating from the local LGN. Each entry is uniquely identified by its query ID. The record structure for the query table entries is shown below. Every response received by SM is checked against this table to determine whether or not the response is unsolicited.

```
#define    SM_MAXQUERYONHOLD        50

typedef  struct  SM_QUERYONHOLDtag
{
MODELSHEAD      mQueryhead;                          /*  MODELS message header          */
char            sDestmod[SY_MODMNEMLN + 1];          /*  Destination module             */
char            sLgn[SY_LGNNMLN + 1];                /*  Destination LGN                */
time_t   tQrytime;                                   /*  Time SM received the query     */
time_t   tResptime;                                  /*  Time SM received query response */
BYTE            byQueryid;                           /*  Unique ID of query             */
} SM_QUERYONHOLD;
```

SM also uses data in this table to form the performance/activity report in response to a query made of SM.

Differences Between LGN and CLGN Implementations

As noted previously, the CLGN can query any other LGN. All other LGNs are restricted to making local queries and responses, and sending responses to the CLGN.

Shutdown Procedures

SM enters the Shutdown State in the usual way, in response to an unrecoverable error or upon receiving a SHUTDOWN message from SB.

Serialization

This module needs to be continuously available; it cannot be serialized with any other modules.

Files

Query responses refer to an ASCII file containing the actual response to any query. These files are scrolled on an appropriately sized window on the screen.

Alternative Designs

For a discussion of SM module alternative designs, see *Remote Control Facility*.

SYSTEM UTILITIES MODULE

Purpose and Description

The SU module enables the execution of operating system commands or executable files on a remote LGN. An operator enters the utility commands as part of a dialog generated when the Remote Utility option from the SB menu is selected. The utility commands can be simple DOS commands, off-the-shelf programs, or batch files. Additionally, SU is automatically used during table update operations; the receiving end of a table update message runs an SU-controlled batch file to copy downloaded files from a temporary directory to the target directory.

SU spawns a separate process, with the mnemonic UT, to run the batch file. This process executes the batch file, and thus the requested commands, and then runs the program Subatdon.Exe, which sends a BATCHDONE message to SU. The UT process then shuts down.

UT's size must be kept minimal. It uses the C-language system call, which invokes a secondary copy of the DOS command interpreter but does not release the calling program from memory. This arrangement decreases available memory for commands invoked by the secondary command interpreter, and UT simultaneously exacts as much allocated memory as possible from DESQview to accommodate the most space-consuming utility commands. From the point of view of DESQview, UT and the commands it invokes make up one process.

Boot Procedure

SU is created by SB at system start-up. Once the LGN's Initialization State procedures are completed, SB shuts down SU to save system cycles and re-spawns SU as needed to run requested utility commands.

Processing

Frequency of Operation

SB creates SU anew to run system utilities whenever SU's waiting period since the previous utility request has run out. When SU completes its task, it remains dormant for a pre-determined amount of time, awaiting a UTIL_REQST or a RELOADTRANS message. If it receives neither within the allotted time, it shuts down. For the next utility run, SB checks to see whether SU has shut down since it was last used. If not, messages are sent to SU, and operation continues apace; if so, SU is started again.

Flow of Processing

SU is activated upon receiving a RELOADTRANS or UTIL_REQST message. For a RELOADTRANS message, SU constructs a batch file named SURELOAD.BAT to copy the designated tables from a temporary download directory to the appropriate target directory (C:\DE\TRANSTBL or C:\ED\TRANSTBL). SU does not construct a batch file for other utility requests. SU then asks SB, via IPC message, to suspend modules affected by the request, and waits 5 seconds for the module suspensions to be completed. SU can then spawn the SU-batch (UT) process, which executes the requested command or the SURELOAD.BAT file. At this point, SU waits for a BATCHDONE message indicating completion of the UT process.

UT executes the command or batch file via a system call. Upon completion, control returns to UT, and UT executes the Subatdon.Exe program via another system call. Subatdon sends a BATCHDONE message to SU, informing SU that the utility command has been run. Then SU signals SB to restart all suspended modules, starts a timer, and waits in the object queue for the next request.

If no requests are received by the time the timer expires, SU notifies SB that it is shutting down and terminates its process.

Shutdown Procedure

SU shuts down in the usual way in response to a SHUTDOWN message from SB or in response to an unrecoverable error. Unlike other modules, and as described, SU also shuts down after a period of inactivity.

Serialization

SU is functionally independent; it cannot be serialized with any other modules.

Files

The batch file SURELOAD.BAT in the directory \SU\BIN is executed by the UT process exclusively for table updates. It is rewritten every time SU receives a table update request. Additional files may be produced as by-products of commands invoked by SU.

REMOTE CONTROL FACILITY

There are two aspects of the prototype LGN's remote control facility, or remote access capability. First, it is possible to direct any file or executable command to a remote LGN from the CLGN; likewise, data can be sent from any LGN to the CLGN in response to a CLGN request. Second, by using off-the-shelf communications software products, it is possible to make a keyboard at the CLGN appear to the remote LGN to be local to that LGN; in this way an operator can enter keystrokes at the CLGN keyboard that are sent immediately to the remote LGN. However, implementing this second aspect of remote control is complex, as was observed first-hand during the prototype LGN development.

Remote Commands

The SB menu, in tandem with the WAN interface module, allows the CLGN operator to perform the following functions on any remote LGN:

- Suspend, shut down, or restart a module (except for the WAN interface and CLGN polling modules)
- Shut down or reset (shut down and restart) an LGN
- Send a file
- Request any file (including a file that matches a prototype, such as "*.CFG")
- Update a translation table.

Moreover, through the SU module, one can execute any operating system command, executable program, or batch file on a remote LGN.

The LGN operates in local mode by default. This means that all user options entered at the SB menu refer to local modules. By pressing Ctrl-R (the Ctrl key simultaneously with the R key) followed by the Enter key, the operator switches the LGN to remote mode.[2] From the CLGN, all menu operations in the SB menu are available in remote mode. For other LGNs, the remote operations from the keyboard are limited to file requests from the CLGN. (Of course, LGNs exchange other data without operator intervention in response to requests from the CLGN.)

When a remote command is entered at the CLGN keyboard, the operator is prompted for pertinent information to ensure that the command is executed correctly. For all commands, the user furnishes the remote LGN name. In the case of file transfers and file requests, both the remote path name and local path name must be supplied. For table updates and module operations (Suspend, Shutdown, Restart), the affected modules must be entered. For utility requests, the user enters the name of the utility command to be executed remotely.[3]

---

[2]In this mode for the prototype, all SB menu commands are directed to the CLGN (from remote LGNs) or to a remote LGN (from the CLGN).

[3]One effective device is to execute a command for which the output is redirected to a file (e.g., DIR>\test\dir.doc). This command can be followed by a file request for the redirected output file. During prototype development, executing this maneuver proved to be a useful and inexpensive way to analyze a remote LGN from a central location.

Remote Keyboard Control

During prototype development, the software package Remote2[4] was investigated to enable keyboard entry at the CLGN to be redirected to a remote LGN in real time. The virtual local keyboard is connected to its host PC through the WAN and the remote communications software. Conceivably, this could be an effective way to analyze and diagnose LGN problems remotely. The remote communications software could be initiated in host mode during LGN start-up, facilitating remote keyboard control by the CLGN at any time. However, Remote2 communications software has not yet provided a feasible approach to remotely controlling an LGN because (1) a host personal computer (PC) must be callable and (2) incompatibilities exist between Remote2 and other software components of the LGN. Sometimes it has worked flawlessly – even when running concurrently with another communications package – but at other times, it would lock up the remote LGN with no means for recovery, except to physically reboot the machine at the remote site. Nonetheless, with the proliferation of interconnected networks and the requisite remote management software, this approach may merit a second look by the time the production system is developed.

Alternative Designs

Consider a scenario: 50,000 high-priority EDI transactions are in process between the CLGN and a remote LGN, which is directly connected to the WAN. While this process is occurring, an external factor dictates that the CLGN operator must reset the receiving LGN. In the prototype design, all 50,000 transactions will be sent to the LGN and translated before the reset action takes place. To prevent such a problem, high-priority commands must be able to enter the WAN queue ahead of in-process routine traffic. This can be accomplished by including additional priority logic or by using two (or more) concurrent virtual X.25 sessions in the production system.

The first solution could involve program logic capable of interrupting the transmission of the 50,000 transactions and sending the reset message. Implementing this solution may require establishing more priority levels than the present normal and high-priority ones.

---

[4]The Remote2 software provides a virtual local keyboard capability allowing a remote keyboard to act as though it were a local keyboard.

The second approach would require that the WAN interface module use separate virtual sessions for exchanging transactions and for exchanging other messages and data. This arrangement would give inter-LGN message and data transfers some independence from inter-LGN transaction transfers. The approach is not applicable to dial-up sites (although it could be partially implemented if the dial-up site had two modems and two phone lines, an arrangement that seems impractical at this time).

The module most affected by this change is the WAN interface module. The operations subsystem modules would also be affected, to a smaller degree.

CHAPTER 12

LOGGING AND LOGISTICS DATA BASE SUBSYSTEM

Because of time and development platform constraints, no separate logging and logistics data base subsystem has been included in the prototype LGN. These two components of the LGN can serve as the basis, in the production environment, for extracting analytical information about the nature, volume, and efficiency of the data flow throughout the LGN network. A modest logging capability, developed for the prototype LGN, has proved useful in analyzing LGN throughput and in detecting and solving operational problems.

LOGGING

The prototype LGN does not have a separate logging module. Instead, each module uses common LGN logging functions to log events and errors to a log file and/or the screen, depending on the parameters passed to the functions. Each module has its own log file.

A utility program, CLEARLOG.BAT, clears out – but does not erase – a designated log file. The CLEARLOG utility is not run automatically but rather is invoked manually from the keyboard (or through a remote command using the SU module).

The main reason for not designing a stand-alone logging process for the prototype was a concern that having a large volume of log messages passed to the logging module would overburden DESQview's IPC facility. This problem may not occur in the UNIX environment, or there may be a design work-around. One alternative is to pass error events to a common error module, which can log them and take corrective action at its option, while logging non-error events directly as is done currently.

DATA BASE MODULE

No data base module was developed for the prototype systems. In the prototype, all postprocessing of log files is done off-line. At regular intervals, all log files on all

LGNs are requested and subsequently cleared via SU remote commands entered at the CLGN.

CHAPTER 13

LGN MAINTENANCE

This chapter is an overview on maintaining the software modules, data files, and hardware components of the LGN during the course of its day-to-day operations.

Most software maintenance and tuning of the LGN can be accomplished through the various SB menu options previously noted. The most common maintenance tasks are:

- Translation table updates
- Module software updates
- Module configuration file updates
- Download window file updates
- Remote commands (e.g., DIR, DEL, CLRLOG)
- File requests
- LGN reset
- LGN shutdown
- Manual LGN boot.

The rest of this chapter addresses these tasks in more detail. Unless otherwise noted, this chapter views LGN maintenance from the perspective of an operator located at the CLGN. While this implies that modification or analysis refers to a remote LGN, maintenance procedures performed locally usually go through the same menu options.

SYSTEM BOOT MENU

The SB menu is the primary interface between an operator and the LGN software. Discussed earlier, it is listed again here for convenience in Table 13-1. In addition, Ctrl-R toggles the menu operation between local and remote mode. In local mode, all operations refer to the local LGN; in remote mode, all operations refer to a

TABLE 13-1

SB MODULE OPTIONS

Option number	Description
1	Suspend a module
2	Restart a module
3	Shut down a module
4	Transfer a file to a remote LGN
5	Update a translation table on a local or remote LGN
6	Run a system utility
7	Reset an LGN (shut down all LGN processes and restart the LGN)
8	Request a file from a remote LGN
0	Shut down an LGN

remote LGN, whose name the user furnishes in response to a prompt. The SB menu is available as soon as the LGN enters the Operational State.

To select a menu option, type the menu option number and press the Enter key. At any time during the dialog following a menu option selection, pressing the Escape key cancels the option and brings up the menu again. Two-letter module mnemonics must always be keyed in upper case.

The LGN is in local mode by default; pressing Ctrl-R followed by the Enter key switches the LGN to remote mode. When a menu option is selected in remote mode, the next prompt is always for the mnemonic identifier of the target LGN. In the discussions of the various menu options below, this prompt is implied.

GENERAL PROCEDURE FOR REMOTE MODE

SB menu options entered in remote mode cause SB to create a corresponding message, which it embeds within a SENDMSG, SENDFILEMSG, or SENDFILE message. This outer message goes to the WAN interface module, which sends it over the WAN to the target LGN. (The WAN interface module divides SENDFILEMSG messages into separate SENDFILE messages and SENDMSG messages.) On the receiving end, the WAN interface module determines the message type via the MNP,

strips off the outer part of the message, and sends the embedded contents to SB. At that point, SB services the message as if it were a local command.

TRANSLATION TABLE UPDATES

Translation table updates are translation table replacements; the entire file is overwritten. There is a danger that a replacement translation table may be truncated or corrupted by a communications error while it is replacing the old table. For protection, replacement tables are copied first to a temporary directory on the target LGN and then to the destination directory.

To update (replace) a translation table, select option 5 from the SB menu, enter the full path name of the file to be replaced, and enter the translation process (DE or ED) that will be affected. The new table is sent to the directory C:\SU\DOWNLDED on the target LGN. Then, the affected translator module is suspended, the table is copied from the \SU\DOWNLDED directory to the correct target directory (\DE\TRANSTBL or \ED\TRANSTBL), and the translator module is restarted. A copy of the replacement table will remain in the \SU\DOWNLDED directory until deleted via the remote command facility.

MODULE SOFTWARE UPDATES

To update an executable module program, select file transfer (option 4) from the SB menu. Answer prompts for source file and destination file names (including full path). If left blank, the destination name defaults to the source name. The file will be sent to the target LGN and directory by the WAN interface module. Remember that remote LGNs can send files to the CLGN only.

Theoretically, at this point the affected module would be shut down and immediately restarted. But, since not all testing in the area of opening and closing mailboxes has been completed, it is not certain that this procedure will work for all modules. If shutting down and restarting the module does not work, it will be necessary to reboot the LGN manually. This is an acknowledged limitation of the prototype LGN.

To shut down a module, select option 3 from the SB menu. This option is straightforward; enter the two-letter mnemonic of the module to be shut down. The

SHUTDOWN message will be delivered over the WAN to the target LGN, where SB will issue an appropriate SHUTDOWN message.

To restart a module, select option 2 from the SB menu. This option is the converse of the module shutdown option; enter the two-letter mnemonic of the module to restart. The RESTART message will be relayed to the target SB module, which will restart the module in question.

MODULE CONFIGURATION FILE UPDATES

Often a change in the environment of an LGN will necessitate updating one or more module configuration files. The file transfer option (option 4) on the SB menu triggers a remote configuration file update in the normal way. As with other file updates, the configuration file is actually overwritten, not updated in the normal way. On prompt, enter the full path name of the source file and, optionally, the full path name of the destination file. If not entered, the destination name defaults to the source file name. A SENDFILE message is relayed to the WAN interface module, which uploads the file directly to the destination path on the receiving LGN.

Usually, it is necessary to suspend and restart the module associated with an updated configuration file if the new parameter values are to take effect. This is done by selecting suspend (option 1) and restart (option 2) in that order. Both of these options are straightforward, requiring only the two-letter mnemonic of the module affected.

Table 13-2 shows some of the more frequently changed configuration parameters.

DOWNLOAD WINDOW FILE UPDATES

This task traces nearly the same steps as updating module configuration files. The file transfer option (option 4) is selected from the SB menu. The source file name is always C:\LI\DNLD\TABLES\DLWINDOW.DAT. The destination file name may be left blank, since it will default to the source file name. This file is likely to be updated frequently. It is subject to change when mainframe host file names or availability times change. Other factors also may result in an update to the download schedule.

TABLE 13-2

FREQUENTLY CHANGED CONFIGURATION PARAMETERS

File	Parameter	Description
CL.CFG	Poll Interval	Number of seconds between successive polls to the CLGN for message or files. Normally, this is set to a high value resulting in a poll once or twice a day. When several files or messages need to be sent to an LGN, this value may be set to as low as 10 minutes (600 seconds).
LI.CFG	DNLDTimeout	Number of seconds after which to assume that the download process encountered an unrecoverable error. This value may change on the basis of the download software used, the mainframe response time, and the size of download files.
LI.CFG	HostSessionInterval	Number of seconds between download attempts. This value may be adjusted depending on the drain on host resources caused by a download attempt and the likelihood of the download attempt to fail.
WI.CFG	DeferTimeout	Number of seconds to wait before retrying to send a file or message. This value may be temporarily decreased if the CLGN requires several files from the LGN or if the likelihood of a communications failure is high.

REMOTE COMMANDS

One of the more valuable LGN features is an ability to enter a DOS command, executable program, or batch file (collectively called a command or a utility command) to be run on a remote LGN. To invoke this feature, select the run a system utility option (option 6) from the SB menu and (in response to the prompt that follows) the exact command to be run. At the next prompt, enter the two-letter mnemonic of any process to be suspended during the execution of the command; usually, this can be left blank. It is often useful to redirect the output of a command to a file that can be retrieved later. For example, the command DIR \DE\TRANSTBL*.* > \TEST\DIR.DAT saves the list of all DLSS-to-EDI translation tables and sends it to the file \TEST\DIR.DAT, which can be retrieved later using the file request option (option 8).

Two commonly used commands are DEL and CLEARLOG. The DEL command is often used after a table update to delete the temporary copy of the replacement

table (in the \SU\DOWNLDED directory). Sometimes LGN errors result in a proliferation of intermediate files that normally would have been deleted. A DOS TREE command, followed by one or more DEL commands, erases the superfluous files. The CLEARLOG command clears out, but does not delete, a log file. If this command is not periodically applied to log files, they grow indefinitely.

FILE REQUESTS

To retrieve a file from a remote LGN, select file request (option 8) from the SB menu. Answer the prompts for source file name (on the remote LGN) and destination file name. Again, the destination file name defaults to the source file name if left blank. The next prompt is for a file request ID, which is simply a reference number for the file request. The WAN interface log will display the file request ID in the log entry corresponding to the status of the ensuing file transfer. Once all the required information is entered, the WAN interface module downloads the file from the remote LGN. If the requested file does not exist, an appropriate error message is displayed in the WAN interface window and entered in the WAN interface log.

A special feature of the file request option allows an operator to enter a file name prototype (e.g., \WD\STAGING1*.*) for the source file name. In this case, the first file on the remote LGN that matches the prototype will be downloaded. If no files on the remote LGN match, an error message will result.

LGN RESET

The SB menu reset LGN item (option 7) shuts down and immediately restarts all modules on the designated LGN. No further prompts are involved with this option. However, as previously discussed, because of imperfections in this area of the prototype LGN, there is no assurance that all modules will restart once shut down. Moreover, this option will not necessarily bring back a hung LGN if the hang-up results from a hardware problem or from a severe operating system or memory error. Thus, in the prototype, this option is less useful than it could be. More often, a combination of using the LGN shutdown option (option 0) and manually restarting an LGN will be employed.

LGN SHUTDOWN

This is a frequently used option, especially during prototype development. Use of this option, combined with a manual LGN boot, often follows major software

updates on an LGN. Select shut down LGN (option 0) from the SB menu. This option will have the effect of shutting down each LGN module, one by one. The SB module will be the last to go, and the LGN PC will be at the DESQview main menu. Should any module windows remain on the screen, their presence is most likely due to (1) the DESQview "Close on exit" option not being set to "Y" in the module's PIF or (2) an unrecoverable error in the module that prevents it from processing the SHUTDOWN message.

MANUAL LGN BOOT

An LGN can be started in one of three ways:

- Warm-booting the PC (Ctrl-Alt-Del)
- Cold-booting the PC (turning the power on)
- Invoking DESQview and selecting SB from the Open Window menu.

If the LGN has a X.25 PAD board installed, a cold-boot may be necessary to recover from certain X.25 protocol failures. Also, cold-boot may be the only way to recover from other unidentified errors that hang the LGN.

GLOSSARY

AccSys	=	API function library for Paradox
ACK	=	acknowledgment of electronic message
AdCom2-I	=	FTC telecommunications board for PC housing X.25 protocol
Alpha	=	alphabetic data type
ANSI	=	American National Standards Institute
API	=	Application Programming Interface
ASCII	=	American Standard Code for Information Interchange
AWK	=	programming language developed by Aho, Kernighan, Weinberger
baud	=	telecommunications transfer rate unit of measure
BOOL	=	Boolean data type
CCITT	=	International Telegraph and Telephone Consultative Committee
char	=	character data type
CLGN	=	central logistics gateway node
CODEMAP	=	LMI-developed translation table
CRC	=	cyclical redundancy checking
D2E	=	LMI-developed DLSS-to-EDI translation software
DAAS	=	Defense Automatic Addressing System
DAASO	=	Defense Automatic Addressing System Office
DBMS	=	data base management system
DDN	=	Defense Data Network
DIC	=	Document Identifer Code
DIC2TID	=	LMI-developed translation table

DLSS	=	Defense Logistics Standard Systems
DLSS2EDI	=	LMI-developed translation table
DODAAC	=	DoD Activity Address Code
E2D	=	LMI-developed EDI-to-DLSS translation software
EDI	=	electronic data interchange
EDI2DLSS	=	LMI-developed translation table
EDISEGS	=	LMI-developed translation table
EVALDLSS	=	LMI-developed translation table
EVALEDI	=	LMI-developed translation table
FTC	=	Frontier Technologies Corporation
FTTSO	=	file-transfer software
GMT	=	Greenwich Mean Time
int	=	integer data type
IPC	=	interprocess communication
LAN	=	local area network
LEX	=	lexical analyzer
LGN	=	logistics gateway node
LMI	=	Logistics Management Institute
MB	=	megabyte
MODELS	=	Moderization of Defense Logistics Standard Systems
MNP	=	MODELS Network Protocol
NAK	=	negative acknowledgment of electronic message
P-Code	=	"Portable-code"
PAD	=	packet assembler/disassembler
Paradox	=	relational data base by Boland Corporation
PC	=	personal computer
PIF	=	Program Information File

PKARC	=	PKware data compression software
PKXARC	=	PKware data uncompression software
RAM	=	random access memory
SB	=	System Boot (Module)
SM	=	System Monitor (Module)
SU	=	System Utilities (Module)
Super-X	=	FTC run-time software
SuperSort V1.6	=	commercial sorting software
TID	=	transaction ID
TransLog	=	LMI-developed language that captures translation logic
TSR	=	Terminate-and-Stay-Resident
UNIX	=	AT&T Bell Laboratories Operating System
WAN	=	wide area network
WORM	=	Write-Once-Read-Many
XMODEM	=	file transfer protocol
X.25	=	packet-switching network access protocol
XXX	=	transaction test segment
YACC	=	parsing software

APPENDIX A

PROTOTYPE LOGISTICS GATEWAY NODE DISK DIRECTORIES

This appendix contains brief descriptions of all significant files in the prototype logistics gateway node. The files are listed by directory. The file list is for a development personal computer; thus, source files and development tools are included, as well as the executable files and data files that are found on a field LGN. Data file name extensions would be slightly different on a central LGN.

Directory C: - DOS / System Files, Autoexec.Bat

File Name	Description
autoexec.bat	Environment Variables, Adcom2-I TSR Calls, Call to DESQview
dv.bat	DESQview Batch File
loadhi.com	High-Memory TSR Loader
qemm.com	Quarterdeck 386 Memory Manager / Multitasker
config.sys	Qemm and Other Device Driver Specifications
qemm.sys	Quarterdeck System File
qext.sys	Quarterdeck System File

Directory C:\AWK - Awk Software

File Name	Description
awk.exe	Awk pattern language interpreter

Directory C:\CL\BIN - CLGN Polling Executable Code

File Name	Description
cl010.exe	CL CLGN Polling Executable

Directory C:\CL\SRC - CLGN Polling Source Code

File Name	Description
cl010.c	CL CLGN Polling Main Program

Directory C:\CONFIG - Module and System-Wide Configuration Files

File Name	Description
am.cfg	Automsg (Test Message Generator) Parameters
cl.cfg	CLGN Polling Parameters
de.cfg	D2E Parameters
ed.cfg	E2D Parameters
ld.cfg	Lan Dequeuing Parameters
lgn.cfg	LGN Parameters
li.cfg	Local Interface Parameters
sb.cfg	System Boot Parameters
serialnm.cfg	Next Serial Number to be Used by Lan Dequeuing Filter
sm.cfg	System Monitor Parameters
su.cfg	System Utility Parameters
validmod.cfg	Permissable Module Mnemonics
wd. cfg	WAN Dequeuing Parameters
wi.cfg	WAN Interface Parameters
wq.cfg	WAN Queuing Parameters
boot.tab	DESQview Process Table for System Boot

Directory C:\DE\BIN - DLSS-to-EDI Binary Code

File Name	Description
d2e.exe	D2E Executable
*.obj	D2E Object Files

Directory C:\DE\ERR - DLSS-to-EDI Error Files

File Name	Description
*.err	D2E Error Files

Directory C:\DE\INCLUDE - DLSS-to-EDI Include Files

File Name	Description
*.h	D2E Include Files

Directory C:\DE\SRC - DLSS-to-EDI Source Code

File Name	Description
d2e.c	D2E Main Source File
*.c	D2E Function Source Code
d2escan.l	D2E Grammar Definitions
grammar.y	D2E Grammar

Directory C:\DE\TRANSTBL - DLSS-to-EDI Translation Tables

File Name	Description
codemap.db	DLSS Value - EDI Value Mapping
dic2tid.db	DIC Pattern to TID / TID Section Mapping
dlss2edi.db	Rules and Transformations for Forming EDI Transactions
evaldlss.db	DLSS Logical Name Definitions
codemap.px	Index for CODEMAP Table

Directory C:\DE\VAL - DLSS-to-EDI Valid Output Files

File Name	Description
*.val	D2E Output Files

Directory C:\DV - DESQview System and PIF Files

File Name	Description
setup.bat	DESQview Setup Batch File
px-load.com	DESQview Special Loader Program for Paradox 3
dvsetup.dv	DESQview Setup Program Output
desqview.dvo	DESQview Open Window Menu Configuration
am-pif.dvp	AutoMessage PIF File
cl-pif.dvp	CLGN Polling PIF File
de-pif.dvp	D2E PIF File
dl-pif.dvp	Download Script PIF File
ed-pif.dvp	E2D PIF File
l2-pif.dvp	LAN Dequeuing Filter Batch File PIF File
ld-pif.dvp	LAN Dequeuing PIF File
li-pif.dvp	Local Interface PIF F.file
p3-pif.dvp	Paradox 3 PIF File
sb-pif.dvp	System Boot PIF File
sm-pif.dvp	System Monitor PIF File
su-pif.dvp	System Utility PIF File
w2-pif.dvp	WAN Queuing Archive Batch File PIF File
w3-pif.dvp	WAN Dequeuing Un-Archive Batch File PIF File
wd-pif.dvp	WAN Dequeuing PIF File
wi-pif.dvp	WAN Interface PIF File
wq-pif.dvp	WAN Queuing PIF File
desqview.dvs	DESQview Autostart Script
dv.exe	DESQview Program

Directory C:\DVAPI\INCLUDE - DESQview API Include Files

File Name	Description
dvapi.h	DESQview API Type Definitions and Low-Level Definitions
dvapi2.h	DESQview API Function Prototypes

Directory C:\DVAPI\LIB - DESQview API Function Library

File Name	Description
api.lib	DESQview Library of All API Functions
api.lst	Listing of api.lib Contents

Directory C:\DVAPI\OBJ - DESQview API Object Code

File Name	Description
*.obj	DESQview API Object Files (Components of api.lib)

Directory C:\ED\BIN - EDI-to-DLSS Binary Code

File Name	Description
e2d.exe	E2D Executable
*.obj	E2D Object Files

Directory C:\ED\COMPARE - EDI-to-DLSS Compare Files

File Name	Description
.	E2D Compare Files

Directory C:\ED\ERR - EDI-to-DLSS Error Files

File Name	Description
*.err	E2D Error Files

Directory C:\ED\INCLUDE - EDI-to-DLSS Include Files

File Name	Description
*.h	E2D Include Files

Directory C:\ED\SRC - EDI-to-DLSS Source Code

File Name	Description
e2d.c	E2D Main Source File
*.c	E2D Function Source Code
scan.l	E2D Grammar Definitions
grammar.y	E2D Grammar

Directory C:\ED\TRANSTBL - EDI-to-DLSS Translation Tables

File Name	Description
codemap.db	DLSS Value - EDI Value Mapping
edi2dlss.db	Conditions, EDI Field References, and Transformations for Building DLSS Transactions
edisegs.db	EDI Segment Descriptions / Template
evaledi.db	E2D Field Definition
codemap.px	Index for CODEMAP Table

Directory C:\ED\VAL - EDI-to-DLSS Valid Output Files

File Name	Description
*.val	E2D Output Files (Valid Transactions)

Directory C:\FTC\INCLUDE - FTC API (Super-X.25) Include Files

File Name	Description
ftctypes.h	Super-X.25 Type Definitions
ftcx25.h	Super-X.25 Constant and Structure Definitions
intfcx25.h	Super-X.25 Structure and Status Code Definitions
l2_init.h	Super-X.25 Level 2 Structure Definition
l3_init.h	Super-X.25 Level 3 Structure Definitions

Directory C:\FTC\LIB - FTC API Library and Source Files

File Name	Description
call_pkt.c	Super-X.25 Source Code for Converting To/From X.25 Packet Structure
intfcx25.c	Super-X.25 Source Code for API Functions
lctomsc.c	Super-X.25 Source Code for Utility Functions
x25.lib	Super-X.25 Function Library - Recompiled With Turbo C

Directory C:\GLEAF\INCLUDE - GreenLeaf Communications API Include Files

File Name	Description
asiports.h	Constants, Structure Definitions, and Function Prototypes for Communication Ports, Interrupts, Status Registers, Etc.
gf.h	General Include File
ibmkeys.h	Include File for getkey() Special Key Codes
xmodem.h	Constants, Structures, and Function Prototypes for Xmodem Routines

Directory C:\GLEAF\LIB - GreenLeaf Communications API Libraries

File Name	Description
gfc?.lib	GreenLeaf Comm Libraries for Different Memory Models (gfcl - Large Model - Used for LGN Prototype)

Directory C:\GLEAF\SOURCEB - GreenLeaf Communications API Source Files

File Name	Description
*.c	GreenLeaf Comm Source Files

Directory C:\LD\BIN - LAN Dequeuing Binary Code

File Name	Description
automsg.exe	AutoMessage Executable (For Development Only)
ld010.exe	LD Main Executable Program
ld020.exe	LD Semaphore Lock Executable
ld021.exe	LD Semaphore Unlock / FILERDONE Message Sender Executable
sbstub.exe	SB Stub for Testing LD Executable

Directory C:\LD\FILTER - Awk Filter Programs; Sort Directives; Batch File

File Name	Description
addkey.awk	Assigns a Key to All 568 Cards
billkey.awk	Assigns a Key to All MILSBILLS Cards
cons.awk	For MILSTAMP Cards, Determine Container Consolidation Number and Append to Card
gbl.awk	Filters All Single-Card DLSS Transactions; Extracts All Cards Associated with GBL Cards
m561.awk	Groups 561 Cards into Transactions
m562.awk	Groups 562 Cards into Transactions
m564.awk	Groups 564 Cards into Transactions
m565.awk	Groups 565 Cards into Transactions
m566.awk	Groups 566 Cards into Transactions
m568.awk	Groups 568 Cards into Transactions and Strips Off Leading Key
mbills.awk	Groups MILSBILLS into Transactions
mt23.awk	Writes Transactions for T2 and T3 Independent TCMDs (and Subordinate T4s)
rm01.awk	Extracts and Writes T0 and T1 Transactions
filter.bat	LD Filter Batch File - Controls All Filtering and Sorting
sort.com	SuperSort
*.crs	SuperSort Directive Files
error.txt	Transactions That Did Not Pass Filter (Overwritten Each Time)

Directory C:\LD\SRC - LAN Dequeuing Source Code

File Name	Description
automsg.c	LD Test Message Generator
ld010.c	LD Main Process
ld020.c	LD Filter Semaphore Lock
ld021.c	LD Filter Semaphore Unlock and FILTERDONE Message Sender
sbstub.c	System Boot Test Stub (For Development Only)

Directory C:\LD\VAL - LAN Dequeuing Filtered Transaction Files

File Name	Description
*.val	DLSS Transactions That Passed the Filter

Directory C:\LI\BIN - Local Interface Binary Code

File Name	Description
li010.exe	LI Main Executable
li020.exe	LI DNLDDONE Message Sender

Directory C:\LI\DNLD\QUEUE - Uniquely-Named Download Files

File Name	Description
*.dld	Downloaded Raw DLSS Transaction Files

Directory C:\LI\DNLD\STAGING - Staging Area For Downloaded Files

File Name	Description
.	Downloaded DLSS Files - Temporary Holding Area

Directory C:\LI\DNLD\TABLES - Local Interface Download Parameter Files

File Name	Description
dlwindow.dat	LI Download Script Window Paramters

Directory C:\LI\SCRIPTS - Local Interface LGN-Specific Download Scripts

File Name	Description
testscrp.exe	LI Test Download Script Executable
testscrp.c	LI Test Download Script Source
*.bat	LI Download Script Batch File (Site-Specific)
*.exe	LI Download Script Executable (Site-Specific)

Directory C:\LI\SRC - Local Interface Source Code

File Name	Description
li_dnld.bat	LI Script Batch Controlling File
li010.c	LI Main Process Source
li020.c	LI Batch Completion Message Sender Source

Directory C:\SB\BIN - System Boot Binary Code

File Name	Description
sb.exe	SB System Boot Executable
sbreset.exe	SB Restart Executable
subatdon.exe	BATCH_DONE Message Sender Executable

Directory C:\SB\LOG - System Boot Transaction Log

File Name	Description
sb.log	Number of SB Messages by Hour; Record Added for Each Restart

Directory C:\SB\SRC - System Boot Source Code

File Name	Description
oplib.c	SB Operations Library
sb.c	SB Main Process
sblib.c	SB Main Library
sbreset.c	SB Restart
sb_kbin.c	SB Keyboard Interface
sb_g.h	SB Global Variable Definitions
sb_l.h	SB Local Variable Declarations

Directory C:\SM\BIN - System Monitor Binary Code

File Name	Description
sm.exe	SM Executable

Directory C:\SM\SRC - System Monitor Source Code

File Name	Description
oplib.c	SM Utility and Support Functions
sm.c	SM Main Process (Services Queries)
smlib.c	SM Miscellaneous Query Support Functions
sm_kbin.c	SM Keyboard Interface
sm_g.h	SM Global Variable Definitions
sm_l.h	SM Local Variable Declarations

Directory C:\SU\BIN - System Utility Binary Code

File Name	Description
sureload.bat	Temporary File to Copy Translation Table From Temporary Directory to Destination Directory (\DE\TRANSTBL or \ED\TRANSTBL)
su.exe	SU System Utility Executable
su-ut.exe	SU Executable to Spawn Requested Process and SUBATDON Process
subatdon.exe	SU Executable to Send BATCHDONE Message to SU Main Process

Directory C:\SU\DOWNLDED - System Utility Table Holding Area

File Name	Description
*.db	Translation Table

Directory C:\SU\SRC - System Utility Source Code

File Name	Description
oplib.c	SU Operations Functions
su.c	SU Main Process
su-ut.c	SU Spawn Batch Utilities
subatdon.c	SU Send Batch Done Message to SU Main Process

Directory C:\SY\BIN - System-Wide Binary Code

File Name	Description
*.obj	System Function Object Files

Directory C:\SY\INCLUDE - System-Wide and WAN Interface Include Files

File Name	Description
cfg.h	SY Configuration Parameter Mnemonics
errmgr.h	SY Error Definitions/Constants
mnp.h	WI MODELS Network Protocol
models.h	SY MODELS Constants Definitions
paths.h	SY File and Path Definitions
pif.h	SY DESQview PIF Structure Definition
pdox.h	AccSys Paradox API Variable and Constant Definitions and Function Prototypes
su.h	SY Function Return Code Definitions and Prototypes
sulog.h	SY Logging Constant Definitions
sylen.h	SY Variable, Array, and Data Length Constants
symsg.h	SY IPC Message Definitions
systddef.h	SY Standard Types, Constants and Macros
syutil.h	SY Definitions of Constants and Macros That Are Used Only By System-Wide Library Functions
timers.h	SY DESQview Timer Constants
wi.h	WI Constants, Macros, and Type Definitions
wimodem.h	WI Hayes Modem Definitions
wistats.h	WI Definitions for Performance Statistics Maintenance and Reporting
xm.h	WI Xmodem Constants

Directory C:\SY\LIB - Various Libraries

File Name	Description
pdoxl.lib	AccSys Paradox API Library
su.lib	SY System-Wide Utility Function Library (Contains Functions in \SY\BIN)
tllex.lib	LEX Lexical Analyzer Library
tlyacc.lib	YACC Parser Library

Directory C:\SY\Log - LGN Module Logs

File Name	Description
??. log	Module Log File
filter.log	Filter Log File (Overwritten Each Time)

Directory C:\SY\SRC - Source Code for System-Wide Library Functions

File Name	Description
error.c	SY Error Handler Functions
mail.c	SY Mailbox / IPC Functions
sufuncs.c	SY Common Utility and DESQview API Functions
sulog.c	SY Logging Functions
timers.c	SY DESQview Timer Functions

Directory C:\TP\BIN - Transaction Processing Subsystem Binary Code

File Name	Description
tp010.obj	Functions Common to TP Modules

Directory C:\TP\INCLUDE - Transaction Subsystem Include Files

File Name	Description
tp.h	Function Prototypes, Constants, and Global Variables Common to TP Modules

Directory C:\TP\SRC - Transaction Subsystem Source Code

File Name	Description
tp.c	Source for Common TP Functions

Directory C:\UTIL - Utility Programs

File Name	Description
clearlog.bat	Utility to Clear Module Log
pkarc.com	File Compressor
pkxarc.com	File Uncompressor

Directory C:\WD\BIN - WAN Dequeuing Binary Code

File Name	Description
wd010.exe	WD Main Executable
wd020.exe	WD UNARCDONE Message Sender Executable

Directory C:\WD\QUEUE - Destination Directory for Uncompressed EDI Files

File Name	Description
.	Uncompressed EDI File - Input for E2D Translator

Directory C:\WD\SRC - WAN Dequeuing Source Code

File Name	Description
wd_unarc.bat	WD Un-Archive Batch File
wd010.c	WD Main Process
wd020.c	WD UNARCDONE Message Sender

Directory C:\WD\STAGING1 - EDI Archive (Compressed) Files

File Name	Description
.	EDI Archive file

Directory C:\WD\STAGING2 - EDI Uncompressed Files

File Name	Description
.	Uncompressed EDI File - Input for E2D Translator

Directory C:\WI\BIN - WAN Interface Binary Code

File Name	Description
wi. exe	WI Executable
*.obj	WI Object Files

Directory C:\WI\QUEUE\FILE - WAN Interface Outbound Files (Dial-up Sites)

File Name	Description
.	Outbound (For Remote LGN) File

Directory C:\WI\QUEUE\MSG - WAN Interface Outbound Messages (Dial-up Sites)

File Name	Description
.	Outbound Message

Directory C:\WI\QUEUE\REQUEST - WAN Interface Outbound File Requests (Dial-up Sites)

File Name	Description
.	Outbound File Request

Directory C:\WI\QUEUE\TBL - WAN Interface Outbound Translation Tables (Dial-up Sites)

File Name	Description
*.db	Outbound Translation Table

Directory C:\WI\SRC - WAN Interface Source Code

File Name	Description
clkresp.c	WI Clock Response Message Routine
clksync.c	WI Clock Synchronize Routines
comm.c	WI Generic Communication Routines
deferred.c	WI Deferred Message Handler
dialup.c	WI Dial-Up Routines
dodaac.c	WI DoDaac Address Handler
getedi.c	WI Check for and Receive EDI File From Remote LGN
getmsg.c	WI Check for Message on Remote LGN
getparam.c	WI Configuration Parameters Retrieval
halt.c	WI Exit Routine
initx25.c	WI Initialize Super-X.25 Interface
ipcinit.c	WI Initialize DESQView Interface
ipcserve.c	WI IPC Service Routines
linkx25.c	WI X.25 Levl 2 and Level 3 Initialization
makecall.c	WI Connect to LGN
mnpchk.c	WI Check Polling Requests From Remote LGNs
mnprcv.c	WI Handle Receive Portion of MODELS Newtork Protocol (MNP)
mnpsend.c	WI Service an MNP Send File Request
mnpserve.c	WI MNP Functions
modem.c	WI Hayes Modem Routines
netutil.c	WI MNP-to-X.25 Interface
rcvedi.c	WI EDI Message/Data Reception
received.c	WI EDI Reception Support Functions
request.c	WI Initiate File Request (From Remote LGN)
reset2i.c	WI Oversee resetting of FTC AdCom2-I Board

Directory C:\WI\SRC (cont.)

File Name	Description
sendedi.c	WI Send EDI File (Archived)
sendfile.c	WI File Transfer
sendfmsg.c	WI Send File-Message Pair
sendmsg.c	WI Send Message
shutdown.c	WI Normal Termination
startx25.c	WI Manage X.25 Network, Hardware, and Software Initialization
stats.c	WI Performance Statistics - Update and Log
superx25.c	WI X.25 Support Routines - Calls Super X.25 API
suspend.c	WI Suspend Process Routine
switch.c	WI Manage File Sempahore (At TROSCOM Only)
wanexec.c	WI WAN Executive - IPC and WAN Service Manager
wanserv.c	WI Service IPC Queue
wilog.c	WI Logging Routines - Calls SULOG System-Wide Function
wimain.c	WI Main Program
x25serv.c	WI X.25 Packet Send/Receive
xmack.c	WI XMODEM Send Acknowledge/NAK
xmbuffer.c	WI XMODEM Transmit/Receive
xm*.c	Greenleaf XMODEM Functions - Modified for WI Larger Block Size and Use of X.25 Board / Super-X.25 API Instead of Serial Port
ym*.c	Like xm*.c Routines, Except for Dial-up Sites (Not Direct-Connect)

Directory C:\WI\SWITCH - WAN Interface Semaphore Directory (TROSCOM Only)

File Name	Description
port.sem	WI TROSCOM Semaphore File (For Serial Port Contention)

Directory C:\WI\TABLES - WAN Interface Run-Time Tables

File Name	Description
dodaac.dat	WI DoDaac Address File

Directory C:\WQ\BIN - WAN Queuing Binary Code

File Name	Description
wq010.exe	WQ Main Process Executable
wq020.exe	WQ ARCDONE MEssage Sender Executable

Directory C:\WQ\QUEUE - WAN Queuing Outbound EDI Archive (Compresed) Files

File Name	Description
.	EDI Archive File

Directory C:\WQ\SRC - WAN Queuing Source Code

File Name	Description
wq_arc.bat	WQ Archiving Batch File
wq010.c	WQ Main Process
wq020.c	Wq_arc.bat Completion Program; Sends ARCDONE Message to WQ010

APPENDIX B

TRANSLATOR SYSTEM FUNCTIONS

The table on the following pages lists all system functions for both the Defense Logistics Standard Systems (DLSS) to Electronic Data Interchange (EDI) and EDI-to-DLSS translators. The system functions are callable by TransLog expressions in the Paradox tables (and thus in the internal P-Code representations of those tables). The functions receive parameters and pass values via the execution stack in the translator's memory space.

The argument lists after each system function in the table represent values that are passed on the stack and pulled off by the function during its span of control. System symbols in the table that are followed by an asterisk are system variables; all other symbols are functions.

Function Name	Description
AddDays("yymmdd", iDays)	Returns a date incremented by a specified number of days.
Cat("string1", "string2")	Concatenates two string values.
CC()	Synonym (in function form) of CurrentCard.
CurrentCard*	The number of the current card (counting the first card as 1) within the current transaction.
DiffDays("yymmdd-hi", "yymmdd-low")	Returns the difference in days between two YYMMDD dates.
DiffMonths("yymmdd-hi", "yymmdd-low")	Returns the difference in months between two YYMMDD dates.
DlssValue("codename", "edival")	Converts from EDI values to DLSS values via the CODEMAP table.
Dd2Dat("dd")	When passed a two-digit Julian day, supplies the third (most significant) Julian digit and one-digit year, based on current date, and then converts the resulting date to EDI format. The two-digit Julian day is assumed to refer to the next day in the future where the Julian day ends in those two digits.
DocNo2Date("Document-number")	Converts a document number to an EDI date.
DQuant("DLSS-quantity")	Converts a DLSS quantity field into a string containing a numeric quantity. A DLSS quantity field may have an "M" in the last position, which gets converted to "000" (thousands).
EdiElement(wSegSeq, wUse, wEltSeq)	Retrieve the value of an element given the segment sequence number, segment use number, and element sequence number.

Function Name	Description
EdiValue("codename", DLSSValue)	Converts from DLSS values to EDI values via the CODEMAP table.
Exit()	Causes processing of the current TID section to cease.
FindCard(/Pattern/, iDirection, iCardOrigin, wStartCol, wLen [[, "string1", wStartCol1, wLen1]...]	Returns card number that matches a specified pattern in specified columns on the card. More than one pattern-columns pair can be passed. The range of cards to search is also passed.
FindUse(wSegSeq, wSegUse, wEltSeq, /Pattern/ [[, wEltSeq, /Pattern/]...]	Finds a use of an EDI segment based on a value from a particular element within the segment. If wSegUse > 0, than a particular instance of the segment is searched.
InRange(wLower, wUpper)	Boolean function that determines whether an integer is within a certain range.
IsAlpha("string")	Boolean function that determines whether a string is composed entirely of upper case alphabetic characters.
IsNumeric("string")	Boolean function that determines whether a string is composed entirely of digits (0-9).
IsPunch(wCardNo, wCardPos)	Determines if the specified position in a card contains one of the overpunch characters ()JKLMNOPQ).
IsValidMMM("mmm")	Determines whether a month of the form MMM (e.g., DEC) is a valid 3-letter abbreviation.
LastCard*	The number of cards in the transaction.
Len("string")	Returns the length of a string.

Function Name	Description
LZero("string", wLen)	Returns a string padded with leading zeros when passed a string and a length for the returned string, including the zeros.
Matches("string", /pattern/)	Boolean function that determines whether a string matches a regular expression.Mm2mmm("mm")Converts an MM month string to MMM.
Mmmdd2dat("mmmdd", "yy")	Takes an MMMDD string and a YY string and returns an EDI date.
NumVal("string")	Converts a string to an integer.
Punch("string", wPunchPos [, wPunchPos, ...])	Examines up to 19 positions within a string and "punches" those that contain a numeric digit.
RDDD2Dat("ddd" [, "Document-date"])	Calculates an EDI date from a julian day offset in the form DDD and the year of the current date or, if provided, a document date.
RDDS(iMonths, "yymmdd")	Calculates a date equal to the last day of the month of (start date + iMonths.)
RP(iCard, wPos, wLen)	Returns the string found in DLSS card number iCard, starting in position wPos, for wLen columns.
Spaces (wNumber)	Generates a string of wNumber spaces.
StartRDP(iDays, "yymmdd", "day-code")	Computes a date equal to a document date plus number of days minus a day code ("A"=1, "B"=2, etc.)
StrVal(wInt)	Converts a numeric to a string.
SubStr("string", wStart, wLen)	Returns a substring of a given string, starting in position wStart, for a length of wLen characters.
SubtractDays("yymmdd", iDays)	Returns a date decremented by a specified number of days.

Function Name	Description
ToDQuant(lQuant)	Converts a quantity to a DLSS quantity (opposite of DQuant).
UnPunch("string", wUnPunchPos [, wUnPunchPos,...])	Examines up to 19 positions on a card and "un-punches" those that contain overpunch characters.
Value*	Shorthand for the value extracted using the CardNo, Start, and Length fields in the EVALDLSS table.
Yddd2Dat("yddd")	Converts a date of the form YDDD (e.g., 0352) to an EDI date (e.g., 901201).
Ymd2Dat("ymd")	Converts a YMD date to an EDI date. In a YMD date, M is a one-character month code and D is a one-digit day code.
Ymm2Dat("ymm")	Returns the date for the first day of the month in EDI format (e.g., 901201) when passed a year and month of the form YMM (e.g., 012).
Y2yy("y")	Returns the appropriate two-digit year (e.g., 90) when passed a one-digit year (e.g., 0).
Yymmdd2J("yymmdd")	Converts a YYMMDD date to YJJJ format.
Yymmm2Dat("yymmm")	Converts a YYMMM date to an EDI date (YYMMDD).
Yymmmdd2Dat("yymmmdd")	Converts a YYMMMDD string to an EDI date.
Yymmmdd2J("yymmmdd")	Returns the appropriate Julian date (e.g., 90352) when passed an EDI date (e.g., 901218).
Yymmdd2Ymd("yymmdd", bOor1Flag)	Converts a YYMMDD date to a YMD date, where M is a one-character month code, and D is a one-digit day code; the flag parameter affects the usage of the D code.