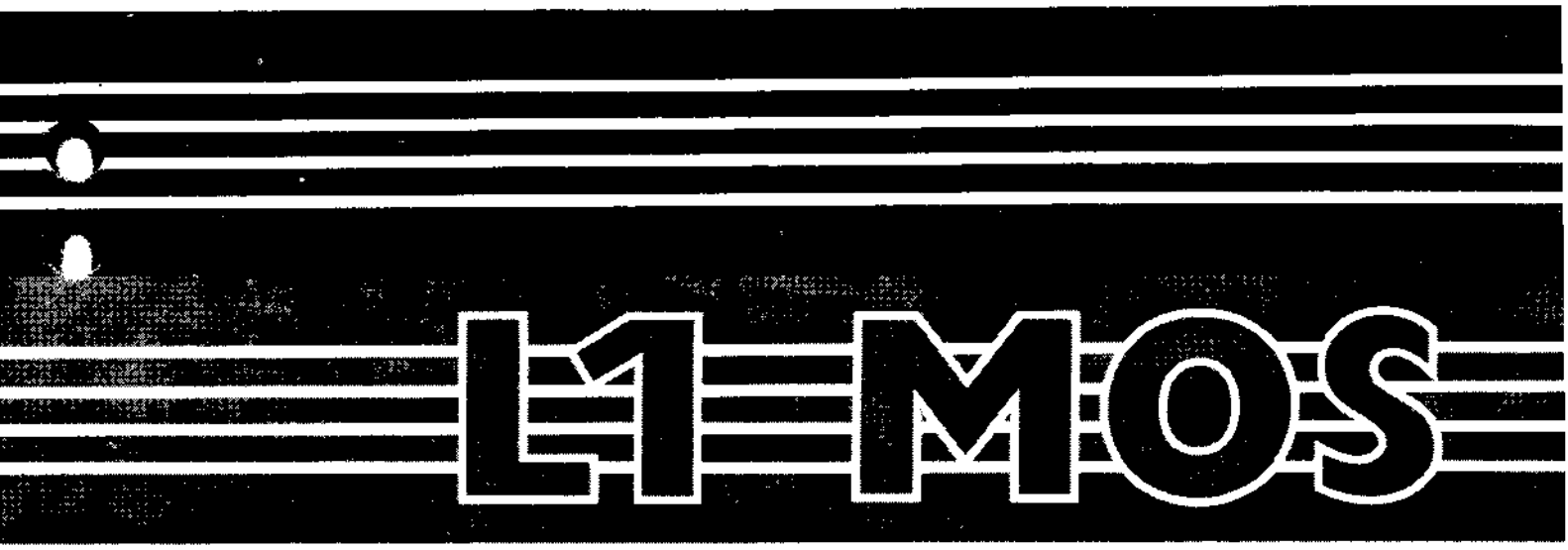
MOS
Programmer Guide

olivetti



MOS
Programmer Guide

olivetti

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C., S.p.A.
Direzione Documentazione
77, via Jervis - 10015 Ivrea (Italy)

PREFACE

This manual is intended as a guide to the programmer who wishes to prepare an application to be executed on L1 MOS systems. It provides information on how to make the best use of the software components in order to take full advantage of the services provided by MOS. (A detailed description of those components is given in other manuals of the MOS documentation line).

SUMMARY

The manual is divided into nine chapters and two appendices:

- The first describes the programming languages and tools provided by MOS for application programming.
- The second is dedicated to the "execution context" and the "program directory"; these are fundamental concepts for application programming in MOS.
- The third describes the recovery modes.
- The fourth describes some application services offered by MOS.
- The fifth gives useful information on how to make the best use of disk files.
- The sixth describes how the screen formats are handled in the various MOS user subsystems.
- The seventh provides information relevant to the use of non integrated terminals as work stations of L1 MOS systems.
- The eighth describes how a program is loaded and executed under the operating system's control.
- The last gives useful information on correct activation of the user activities (both at system initialization and at run-time).
- The appendix A describes the DMPRINT utility used to interpret the dump file.
- The appendix B describes the DISPJOUC command to display the Joucman file contents, and the RECOVERY utility to recover files under Commit.

An analytical index is present at the end of the manual.

REFERENCES

Read first ...

Introduction to MOS - Code 4002130 G (Vol. 2)

For further information, read ...

MOS Basic Architectural Concepts - Code 4002710 J (Vol. 8)

FORTRAN - Program Preparation and Execution - Code 4004510 F (Vol. 6c)

COBOL - Program Preparation and Execution - Code 4004310 T (Vol. 6d)

Compiled BASIC - Program Preparation and Execution
Code 4002180 M (Vol. 6e)

PASCAL+ - Program Preparation and Execution - Code 4002480 T (Vol. 9a)

Program Development Tools - Reference Manual
Code 4002790 S (Vol. 6f)

MOS - SHELL Commands - Reference Manual - Code 4002770 Q (Vol. 3)

MCL - MOS Command Language - User Guide - Code 4002220 H (Vol. 3)

System Software Generation and Installation - User Guide
Code 4002160 B (Vol. 7a)

OLILAN Local Area Network - User Guide - Code 4021820 R (M24 Line)

MOS Operating Guide - Code 3983040 M (Vol. 3)

Distributed System in Local Area Network (LAN) - User Guide
Code 4001060 E (Vol. 6h)

PGU - Graphics Management Package - Programmer Guide
Code 4004650 D (Vol. 6g)

FIRST EDITION: April 1984 - Release 3.0

SECOND EDITION: October 1984 - Release 4.0

THIRD EDITION: March 1985 - Releases 4.1/4.2

FOURTH EDITION: August 1985 - Release 5.0

UPDATE: September 1985 - Release 5.0

FIFTH EDITION: April 1986 - Release 5.1

UPDATE: May 1986 - Release 5.1

REPRINTED: August 1986

(Including Update 6)

SIXTH EDITION: April 1987 - Release 5.2

SUMMARY OF CHANGES

This manual is a new edition containing corrections for Rel. 5.1, the new updates for Rel. 5.2, and editing amendments.

CHANGES

The main changes to Rel. 5.2 included in this manual are:

1. Changes to the table showing organisation of acceptable files (Chapter 1).

2. Description of 1-module format (Chapter 2).

3. Handling of the Dual Log file and of a new return code for Commit Manager (Chapter 3).

4. Increase in the maximum length of syncrhonous messages, a new procedure for reading messages with timeout, and control of message sender's name for Message Switching (Chapter 4).

5. New user application services: EXEC and LOGOFF (Chapter 4).

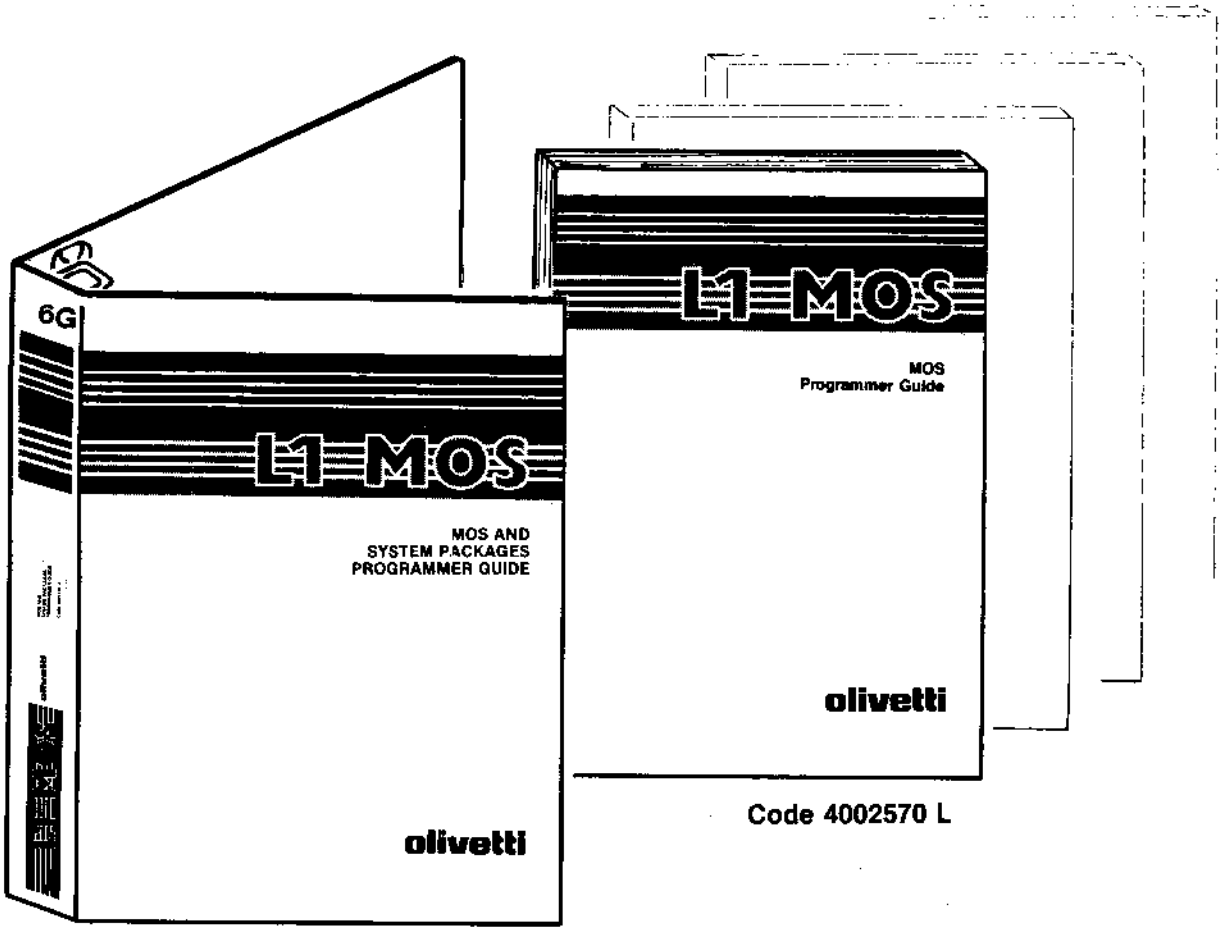
6. New emulation programs intended for the PC used as an L1 MOS work station (Chapter 7).

7. Changes in the table showing segment-content for different components (Chapter 8).

8. Shutdown with timeout by a program (Chapter 9).

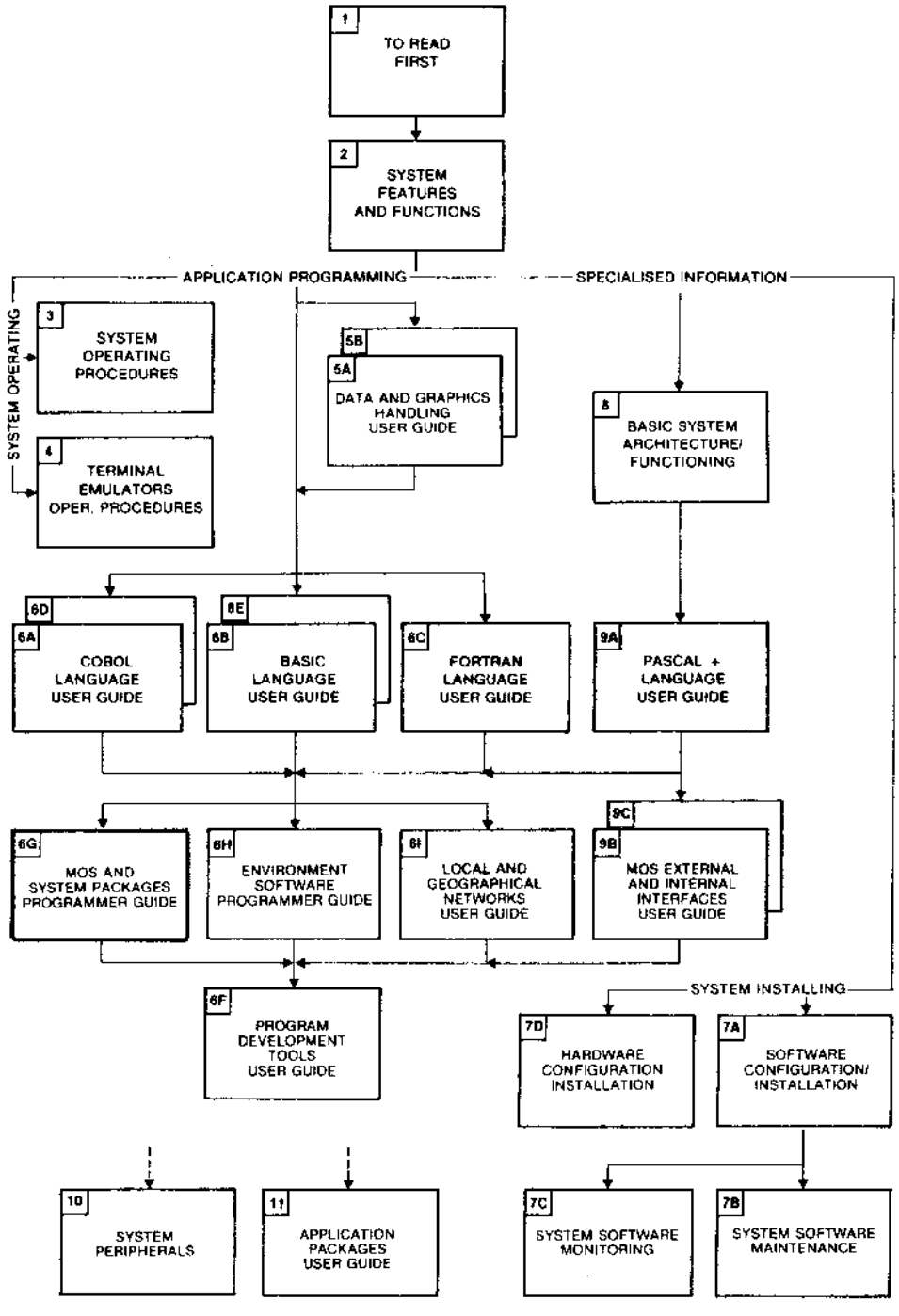
9. Removal of component description of PTP (Program to Program) for L1WSE (ex Appendix B).

10. Description of DISPJOUC command and RECOVERY utility relating to Commit Manager (Appendix B).



Code 4001190 A

Code 4002570 L


1
TO READ FIRST
2
SYSTEM FEATURES AND FUNCTIONS
APPLICATION PROGRAMMING
SPECIALISED INFORMATION
SYSTEM OPERATING
3
SYSTEM OPERATING PROCEDURES
4
TERMINAL EMULATORS OPER. PROCEDURES
5B
5A
DATA AND GRAPHICS HANDLING USER GUIDE
8
BASIC SYSTEM ARCHITECTURE/ FUNCTIONING
6D
6A
COBOL LANGUAGE USER GUIDE
6E
6B
BASIC LANGUAGE USER GUIDE
6C
FORTRAN LANGUAGE USER GUIDE
9A
PASCAL + LANGUAGE USER GUIDE
9C
6G
MOS AND SYSTEM PACKAGES PROGRAMMER GUIDE
6H
ENVIRONMENT SOFTWARE PROGRAMMER GUIDE
6I
LOCAL AND GEOGRAPHICAL NETWORKS USER GUIDE
9B
MOS EXTERNAL AND INTERNAL INTERFACES USER GUIDE
SYSTEM INSTALLING
6F
PROGRAM DEVELOPMENT TOOLS USER GUIDE
7D
HARDWARE CONFIGURATION INSTALLATION
7A
SOFTWARE CONFIGURATION/ INSTALLATION
10
SYSTEM PERIPHERALS
11
APPLICATION PACKAGES USER GUIDE
7C
SYSTEM SOFTWARE MONITORING
7B
SYSTEM SOFTWARE MAINTENANCE

CONTENTS

1. USING THE MOS USER SUBSYSTEMS AND LANGUAGES

From the point of view of application programming, MOS presents itself as a set of interfaces and tools which illustrate its versatility. These are the MOS user subsystems, inasmuch as they use the low level services offered by the operating system, and make them available to the higher level users (operators, programmers, applications), providing a logical interface which is suited to the type of application program for which they are intended.

THE MOS USER SUBSYSTEMS AND LANGUAGES

The MOS user subsystems are:

- software environments which define and control the user's operating environment:
  - Shell
  - BEAM
  - DMS
  - OWS2
  - MTS
  - COBOL ICE
  - Interpreted BASIC
  - ESE
  - Terminal emulators
- system libraries which make available specific functions which can be called by the application programs:
  - RS232/CL
  - Line Manager
  - Commit Manager
  - LMS
  - CAT
  - ONE

- system packages which make available functions not included in the operating system as they are oriented towards specialized applications:

  . PGU

  . RTGSP

  . VISA

  . VISA S6000 compatible

- specific programs such as:

  . utility programs

  . compilers

  . linkers

  . debuggers

  . editor

- a supervisor for initializing, activating and terminating the software environments and user programs.

In order to offer the most suitable interface for particular application problems, the Compiled BASIC, Interpreted BASIC, COBOL, FORTRAN and PASCAL+ programming languages are available.

MULTIFUNCTIONALITY

MOS's multifunctionality means that all the MOS subsystems and available languages can be used to the best advantage in a user-written application (with the only restrictions listed in the section "Assigning and Using the User Segments" in Chapter 9). Thus it is possible:

- to simultaneously execute application programs written in different languages, with eventual exchange of information between them using shared files

- to have different user subsystems active at the same time

- to operate from one work station in different environments, one after the other (or simultaneously, using the batch environment), during the same work session.

The use of MOS's multifunctionality is restricted only by the type of information exchange between programs written in different languages which is requested by the application. This is due to the fact that each language uses organizations for storing the data (files) which are not always used by the other languages.

The following table summarizes the types of files handled by the File System Management, by each language, by the DMS package and by the Sort/Merge utility.

It should be remembered that access to the File System is gained by PASCAL+ routine calls, so that all information relating to files handled by PASCAL+ is the same as that given in the line referring to File System Management.

System components	ACCEPTED FILE ORGANIZATIONS										Allocation Unit (Default)	Handled file number *
	Byte Stream	Positional			Keyed							
		yes/no	record length *	record deletion	yes/no	record length *	key number *	key length *	split strategy	page size		
FILE SYSTEM MANAGEMENT	yes	yes	32K b	yes	yes	32K b	6 (5 sec.)	100 b (primary internal or ext.)	user defined	user defin.	512 bytes	limit fixed in configur.
COBOL ICE	no	sequen. (LF) relat. (LF+LR)	32K b	no yes	index.	4K b	17 (16 sec)	100 bytes	50-50 (def.)	512 bytes (def.)	512 bytes	-
Interpret. BASIC	seq. of item (LF)	random (LF+LR)	4096 b (def.= 256)	no	yes (LF+LR)	4096 b (def.= 256)	6 (5 sec.)	4096 b. (primary intern.)	90-10	512 bytes	32K bytes	15
Compiled BASIC	yes (LF)	sequen. (LF) relat. (LF+LR)	4096 b (def.= 255)	no yes	yes (LF+LR)	4096 b (def.= 255)	stand.=1 exten.=6 (5 sec.)	100 bytes	50-50	512 bytes	512 bytes	100
COBOL	text file (NL)	sequen. (NL) relat. (LF+LR)	32K b	no yes	indexed (LF+LR)	32K b	6 (5 sec.)	100 bytes	50-50 (user modif.)	1024 bytes (user modif)	512 bytes	-
FORTRAN	yes	no	-	-	no	-	-	-	-	-	4096 bytes	17 (default)
DMS	yes (NL)	yes (LF+LR)	8K b	yes	yes (LF+LR)	8K b	6 (5 sec.)	4096 b (primary intern.)	90-10	512 bytes	record length X 1000	-
SORT	yes (outp)	yes	4095 b	yes	yes (inp.)	4095 b	-	-	-	-	-	16 (SORT) 4 (MERGE)

* = Maximum value (4K b for files in RFA)
LF = Admitted file lock
LR = Admitted record lock
NL = Non admitted lock

Fig. 1-1 Accepted File Organizations

MULTIFUNCTIONALITY LEVELS

Four multifunctional levels can be defined:

Level 1) The possibility of simultaneously executing programs written in different languages from different work stations, or sequentially from the same one, without exchanging information between the application environments.

Level 2) The function of level 1), but the programs exchange information using files accessed separately.

Level 3) The function of level 2), but the programs executed from different work stations concurrently access shared files.

Level 4) Possibility to link objects obtained by sources written in different languages.

The restrictions of these four multifunctional levels are described below.

Level 1

The level 1) functions are guaranteed for programs written in all the available languages.

Level 2

Information on Level 2) functions, i.e those environments which can exchange information and the types of file by means of which this operation is performed (with separate access), can be obtained directly from the previous table.

For example, the languages

COBOL and Compiled BASIC

can exchange information by means of the following files:

Text COBOL - Byte Stream Compiled BASIC

Sequential COBOL - Sequential Compiled BASIC

Relative COBOL - Relative Compiled BASIC

Indexed COBOL - Keyed Compiled BASIC

The information is obtained by comparing the organization of the files relating to the two languages.

Level 3

Information on level 3) functions, i.e those environments which can exchange information and the files by means of which this operation is performed (with concurrent access), can be obtained directly from the previous table, taking into consideration only those types of file which permit file lock (LF in the table).
For example, the languages

COBOL and Compiled BASIC

can exchange information with concurrent access by means of the following files:

Relative COBOL - Relative Compiled BASIC

Indexed COBOL - Keyed Compiled BASIC

The information is obtained by comparing the organization of the files relating to the two languages.

Level 4

Objects related to sources in different languages can be linked together. This is allowed for the following languages:

- Compiled BASIC - PASCAL+

  when the Compiled BASIC program is used to recall a PASCAL+ procedure. See the manual Compiled BASIC, Program Preparation and Execution.

- COBOL - PASCAL+

  when the COBOL program is used to recall a PASCAL+ procedure or when the PASCAL+ program is used to recall a COBOL procedure (the MAIN must be written in COBOL). See the manual COBOL, Program Preparation and Execution.

NOTES ON DATA EXCHANGE

For data exchange between the various environments, it is necessary to bear in mind the relations that exist between the types of data from one environment towards the others.

PASCAL+ Types

PASCAL+	COBOL ICE	COBOL	Comp. BASIC	Interp. BASIC (1)	FORTRAN	DMS (2)	SORT (3)
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	yes
integer (2 bytes)	COMP	COMP	integer	integer	INTEGER*2	COMP	yes
real (4 bytes)	-	COMP-1	Single Precision	Single Precision	REAL*4	COMP-1	yes

(1) Data can also be exchanged between the Positional with Deletion PASCAL+ and Random Interpreted BASIC files, remembering that:

- the file must be created by the MCL utility as a Positional no Deletion file or by a PASCAL+ program
- the Interpreted BASIC program also accesses deleted or non-existent records, replying not an error indication, but data equal to binary 0.

(2) Data can also be exchanged between Positional no Deletion PASCAL+ and Sequential DMS files, remembering that:

- the file must be created by the MCL utility or by a PASCAL+ program
- the DMS cannot carry out delete operations.

(3) Data can also be exchanged between Positional no Deletion PASCAL+ and Sequential SORT files, with the latter mapped on the Positional no Deletion file, remembering that the Sequential SORT is only an input file.

COBOL ICE Types

COBOL ICE	PASCAL+	COBOL	Comp. BASIC	Interp. BASIC (1)	FORTRAN (2)	DMS (3)	SORT (4)
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings			
COMP	integer (2 bytes)	COMP	integer	integer			
COMP-3	-	COMP-3	-	-			

(1) Data can also be exchanged between the Relative COBOL ICE and Random Interpreted BASIC files, remembering that:

- the file must be created by the MCL utility as a Positional no Deletion file, or by a COBOL program
- the Interpreted BASIC program also accesses the deleted or non-existent records, replying not an error indication but data equal to binary 0.

(2) Data cannot be exchanged between COBOL ICE and FORTRAN.

(3) All types of COBOL ICE data are compatible with DMS. In addition, data can be exchanged between Sequential COBOL ICE and Sequential DMS files, remembering that:

- the file must be created by the MCL utility as a Positional no Deletion or by a COBOL program
- the DMS cannot carry out delete operations.

(4) All types of COBOL ICE data are compatible with SORT. In addition, data can be exchanged between Sequential COBOL ICE and Sequential SORT files, with the latter mapped on the Positional no Deletion file, remembering that the Sequential SORT is only an input file.

COBOL Types

COBOL	PASCAL+	COBOL ICE	Comp. BASIC	Interpr. BASIC (1)	FORTRAN	DMS (2)	SORT (3)
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings		
COMP	integer (2 bytes)	COMP	integer	integer	INTEGER*2		
COMP-1	real (4 bytes)	-	Single Precision	Single Precision	REAL*4		
COMP-3	-	COMP-3	-	-	-		
COMP-2	-	-	Double Precision	Double Precision	REAL*8		
COMP-4	-	-	-	-	INTEGER*4		

(1) Data can also be exchanged between Relative COBOL and Random Interpreted BASIC files, remembering that:

- the file must be created by the MCL utility as a Positional no Deletion file or by a COBOL program
- the Interpreted BASIC program also accesses deleted or non-existent records, replying not an error indication but data equal to binary 0.

(2) All types of COBOL data are compatible with DMS. In addition, data can be exchanged between Sequential COBOL and Sequential DMS, remembering that:

- the file must be created by the MCL utility as a Positional no Deletion or by a COBOL program
- the DMS cannot carry out delete operations.

(3) All types of COBOL data are compatible with SORT. In addition, data can be exchanged between Sequential COBOL and Sequential SORT files, with the latter mapped on the Positional no Deletion file, remembering that the Sequential SORT is only an input file.

Interpreted BASIC Types

Interp. BASIC	PASCAL+ (1)	COBOL ICE (1)	COBOL (1)	Comp. BASIC (1)	FORTRAN	DMS (2)	SORT (3)
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	yes
integer	integer	COMP	COMP	integer	INTEGER*2	COMP	yes
Single Precision	real	-	COMP-1	Single Precision	REAL*4	COMP-1	yes
Double Precision	-	-	COMP-2	Double Precision	REAL*8	COMP-2	yes

(1) Data can also be exchanged between the following files: Random Interpreted BASIC and Positional with Deletion PASCAL+, Relative COBOL ICE, Relative COBOL, Relative Compiled BASIC. It should be remembered that:

- the file must be created by the MCL utility as a Positional no Deletion or by a program in the relevant language
- the Interpreted BASIC program also accesses the deleted or non-existent records, replying not an error indication but data equal to binary 0.

(2) Data can also be exchanged between Random Interpreted BASIC and Sequential DMS files, remembering that:

- if the file is created by the MCL utility as a Positional no Deletion or by a BASIC program, the DMS cannot carry out delete operations
- if the file is created by DMS, the Interpreted BASIC program also accesses deleted or non-existent records, replying not with an error indication but with data equal to binary 0.

(3) Data can also be exchanged between Random Interpreted BASIC and Sequential SORT files, with the latter mapped on the Positional no Deletion file, remembering that the Sequential SORT is only an input file.

Compiled BASIC Types

Comp. BASIC	PASCAL+	COBOL ICE	COBOL	Interp. BASIC (1)	FORTRAN	DMS (2)	SORT (3)
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	yes
integer	integer	COMP	COMP	integer	INTEGER*2	COMP	yes
Single Precision	real	-	COMP-1	Single Precision	REAL*4	COMP-1	yes
Double Precision	-	-	COMP-2	Double Precision	REAL*8	COMP-2	yes

(1) Data can also be exchanged between Relative Compiled BASIC and Random Interpreted BASIC files, remembering that:

- the file must be created by the MCL utility as a Positional no Deletion or by a Compiled BASIC program
- the Interpreted BASIC program also accesses deleted or non-existent records, replying not an error indication but data equal to binary 0.

(2) Data can also be exchanged between Sequential Compiled BASIC and Sequential DMS files, remembering that:

- the file must be created by the MCL utility as a Positional no Deletion or by a Compiled BASIC program
- the DMS cannot carry out delete operations.

(3) Data can also be exchanged between Sequential Compiled BASIC and Sequential SORT files, with the latter mapped on the Positional no Deletion file, remembering that the Sequential SORT is only an input file.

FORTRAN Types

FORTRAN	PASCAL+	COBOL ICE (1)	COBOL	Comp. BASIC	Interp. BASIC	DMS	SORT
Alphanum. strings	Alphanum. strings		Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	yes
INTEGER*2	integer		COMP	integer	integer	COMP	yes
REAL*4	real		COMP-1	Single Precision	Single Precision	COMP-1	yes
REAL*8	-		COMP-2	Double Precision	Double Precision	COMP-2	yes
INTEGER*4	-		COMP-4	-	-	COMP-4	yes

(1) Data cannot be exchanged between FORTRAN and COBOL ICE.

DMS Types

DMS	PASCAL+ (2)	COBOL ICE (1) (2)	COBOL (1) (2)	Comp. BASIC (2)	Interp. BASIC (3)	FORTRAN	SORT (4)
Alphanum. strings	Alphanum. strings			Alphanum. strings	Alphanum. strings	Alphanum. strings	
COMP	integer			integer	integer	INTEGER*2	
COMP-1	real			Single Precision	Single Precision	REAL*4	
COMP-2	-			Double Precision	Double Precision	REAL*8	
COMP-4	-			-	-	INTEGER*4	

(1) All types of DMS data are compatible with COBOL ICE and COBOL.

(2) Data can also be exchanged between the following files: Sequential DMS and Positional no Deletion PASCAL+, Sequential COBOL ICE, Sequential COBOL, Sequential Compiled BASIC. It should be remembered that:

- the file must be created by the MCL utility as a Positional no Deletion or by a program in the relevant language
- the DMS cannot carry out delete operations.

(3) Data can also be exchanged between Sequential DMS and Random Interpreted BASIC, remembering that:

- if the file is created by DMS, the Interpreted BASIC program also accesses deleted or non-existent records, replying not an error indication but data equal to binary 0
- if the file is created by the MCL utility as a Positional no Deletion file or by a BASIC program, the DMS cannot carry out delete operations.

(4) The data which can be exchanged between DMS and SORT are all that allowed by COBOL.

Types Related to SORT

SORT	PASCAL+ (2)	COBOL ICE (1) (2)	COBOL (1) (2)	Comp. BASIC (2)	Interp. BASIC (2)	FORTRAN	DMS (3)
Alphanum. strings	Alphanum. strings			Alphanum. strings	Alphanum. strings	Alphanum. strings	
integer (2 bytes)	integer			integer	integer	INTEGER*2	
integer (4 bytes)	-			-	-	INTEGER*4	
real (4 bytes)	real			Single Precision	Single Precision	REAL*4	
real (8 bytes)	-			Double Precision	Double Precision	REAL*8	

(1) All types of COBOL ICE and COBOL data are compatible with SORT.

(2) Data can also be exchanged between the following files: SORT Sequential, mapped on the Positional no Deletion file, and Positional no Deletion PASCAL+, Sequential COBOL ICE, Sequential COBOL, Sequential Compiled BASIC, Random Interpreted BASIC. It should be remembered that the Sequential SORT is only an input file.

(3) The data which can be exchanged between SORT and DMS are all that allowed by COBOL.

USING THE SYSTEM LIBRARIES AND PACKAGES

From an application point of view, MOS provides the programmer with a set of functions oriented to specific applications.

Any user-written program can access these functions, respecting the visibility restrictions imposed by the language used.

The functions in question can be grouped in a system library, a system package or a software environment, according to the situation.

System Libraries and Packages

A system library or package is a set of routines which create specific functions. The application program which wants to call them must be linked to the library during the linking phase by appropriate interfaces or, in some cases, by including the library in the program's source code using an appropriate statement. The system libraries currently available are:

- VISA (for handling the work station), which can be called by Compiled BASIC, Interpreted BASIC, COBOL, DMS and PASCAL+.
- PGU (for producing graphic output), which can be called by Compiled BASIC, Interpreted BASIC, COBOL, DMS, FORTRAN and PASCAL+.
- RTGSP (for producing business graphic output), which can be called by Compiled BASIC, Interpreted BASIC, COBOL and PASCAL+.
- RS232/CL (for handling the RS232/CL interface, which allows connection of serial peripherals), which can be called by Compiled BASIC, Interpreted BASIC, COBOL, FORTRAN and PASCAL+.
- Line Manager (for handling the I/O on communication line), which can be called by Compiled BASIC, COBOL and PASCAL+.
- Commit Manager (for handling disk access through user defined indivisible transactions), which can be called by Compiled BASIC, Interpreted BASIC, COBOL and PASCAL+.
- LMS (for keeping a trace of hardware and software failures, events and data), which can be called by COBOL and PASCAL+.
- CAT (for handling banking devices), which can be called by COBOL and PASCAL+.
- ONE (for accessing geographical networks), which can be called by COBOL and PASCAL+.

The Transactional Environment

The transactional environment offered by MOS, MTS, provides a set of services which can be called by an application written in COBOL and PASCAL+.

The MTS application environment also allows the user to add other programs to the standard services, which are written in COBOL or PASCAL+ and known as Server Programs.

The BEAM Environment

The BEAM environment (an interactive interface for business applications) provides primitives for writing information to the log file and controlling execution of activities which cannot run concurrently. These primitives can be called from user application programs written in Compiled BASIC, COBOL and PASCAL+.

Calling the Functions

The user should read the program preparation and execution manuals relating to the language used for the application for information on how to use the features offered by the system libraries or the MTS application environment. Titles and codes of these manuals are given in the bibliography in the Preface to this manual.

The following table summarizes the functions which can be accessed by the languages available for application programming.

	Compiled BASIC (call to external procedure)	Interpreted BASIC	FORTRAN (run-time)	COBOL (call to external procedure)	PASCAL+ (procedure importing)
RS232/CL (1)	X	X	X	X	X
PGU (1)	X	X	X	X	X
RTGSP	X	X		X	X
Line Manager	X			X	X
VISA	X	X		X	X
MTS				X	X
BEAM	X			X	X
LMS				X	X
ONE				X	X
CAT				X	X
Commit Man.	X	X		X	X

Tab. 1-2 Libraries which can be Called by the Programming Languages

(1) The functions available to Interpreted BASIC are included in the language itself.

A description of the Commit Manager functions is given in the chapter "Recovery Modes" in this manual.

Other services offered by MOS are described in the Chapter "MOS Application Services" in this manual.

User Libraries

The user can prepare a customized library to provide a set of functions which are not covered by system libraries or packages.

The language to be used for extending the functions which can be called is PASCAL+.

MOS provides the programmer who wants to create a library of this type with a large set of PASCAL+ functions or procedures, known as system primitives.

These primitives allow a direct and customized handling of various components, such as:

- the PMM (Process and Memory Management)
- the File System Management
- the work station driver
- the work station printer and relevant subdevices driver
- banking peripherals drivers
- PIN-check driver.

A description of the system primitives is given in the manuals of volume 9b, MOS External Interfaces, User Guide.

The information on the various steps involved in creating a user library is given in the manual PASCAL+, Program Preparation and Execution. See also the section "Notes on Writing a User Package" in Chapter 9.

See the manuals COBOL, Program Preparation and Execution and Compiled BASIC, Program Preparation and Execution for details on how a COBOL or a Compiled BASIC program calls the functions provided by a user library.

See also the Program Development Tools, Reference Manual for details on how to link a PASCAL+ user library to a COBOL or a Compiled BASIC program.

2. APPLICATION PROGRAMMING

STATIC AND DYNAMIC OBJECTS

A distinction among the basic MOS elements can be made between:

- static objects
- dynamic objects.

Static Objects

Static objects are those that exist independently of the current processing, for example programs, procedures, data files, etc. Each is uniquely identified by a path name (and eventual aliases).

The uniqueness of this path name is guaranteed, even in a distributed configuration, by the File System Management.

This means that the user does not have the problems of handling diverse objects, resident on different and interconnected systems, which have the same name.

Dynamic Objects

Dynamic objects are those that exist (in memory) only while the process in which they are involved lasts. They are uniquely identified by MOS in the configuration in which they operate for as long as they exist.

A static object can have diverse corresponding dynamic objects at any time, such as processes (programs in execution) or program execution contexts (structures that guarantee the correct connection between the names used by the programs for the objects needed and the names by which the system refers to these objects).

PATH NAME HANDLING

All the static objects under MOS are identified by a path name. These objects are stored on disk as files, and are handled independently of their physical position and the disk on which they reside.

A set of static objects (files, directories or volumes) can be grouped in a volume, which is a contiguous space on disk whose size is defined by the user when it is created.

The character "/", in a path name, separates the nodes (volumes or directories) which must be passed through when searching for the file specified at the end of the path name.

The root of the resulting hierarchic structure is the "Memory Volume", which is a structure created in each system when they are initialized, according to the value declared at generation time as the maximum number of files allowed in the memory volume.

MEMORY VOLUME

This volume has the following characteristics:

- The volume is referred to with the character "/" and represents the "localroot" directory, which is the highest point of the local file system.

- Information relating to the local system is contained in the "localroot" directory, is as follows:

  . the system volume, which is logically and automatically mounted with the name IPL/SYS at initialization time

  . the directory with information on the peripherals (DEV)

  . the directory with information on the program execution contexts (CONTEXT$)

  . the directory which will contain the names of the transient files created in memory (TMP).

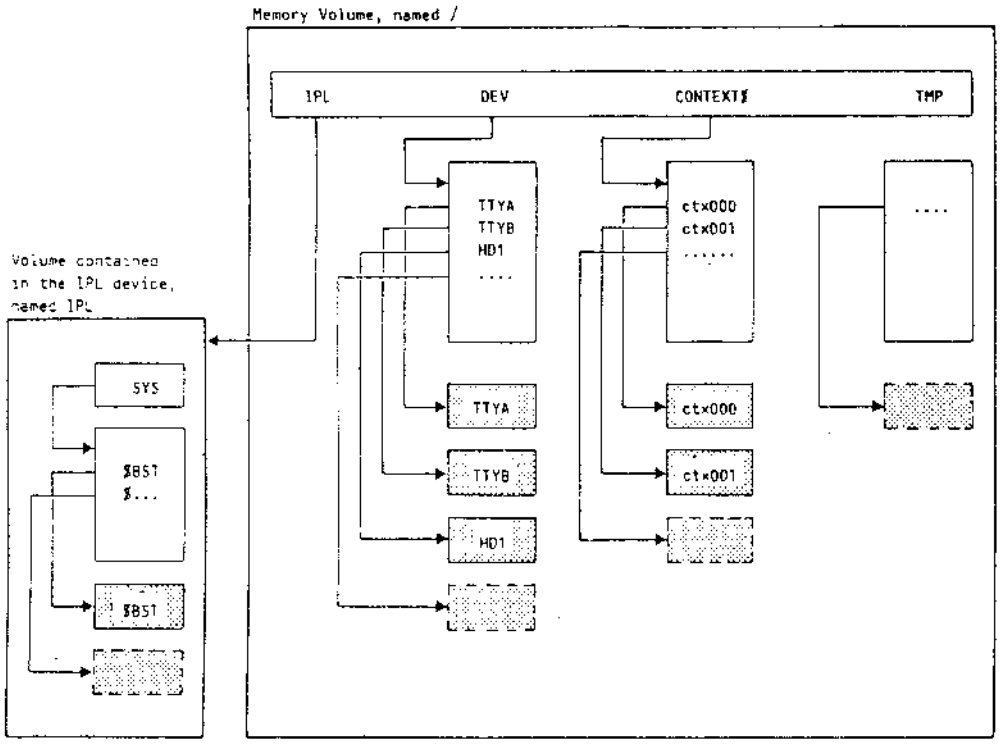


Fig. 2-1 Memory Volume

All the user volumes (not resident on the same physical support as the IPL volume), whose contents are to be used, must be logically connected (MNT command) to this memory volume before they can be referred to.

The memory volume is thus a structure created according to the initial system configuration, and then dynamically updated by MOS during the system's activities, to keep track of the availability of new resources (for example, the files contained in a volume which is logically connected to the memory volume).

PROGRAM STRUCTURING

The concepts of "program execution context" and "program directory" are introduced below. It is via these elements that the user can appropriately structure and activate the application, providing it with the visibility of the necessary resources.

An execution "environment" is associated to each program executed under the control of the MOS operating system. This environment structurally consists of two elements:

- The program execution context
- The program directory

The program execution context element takes into account the characteristics of the family to which the process executing the program belongs.

The program directory is a structure which lists the static objects available to the program.

THE PROGRAM EXECUTION CONTEXT

The part of the environment in which a program is executed, known as "execution context", consists of the list of dynamic objects associated to the program itself.

A table is associated to each activated program, which guarantees the correct connection between the logical names (used in the program) and the physical names (with which the system uniquely refers to the resources shown).

This table is compiled automatically (without requiring user intervention) by the application environment in which the program is activated.

The context table for the programs activated by Grandpa is compiled according to the specifications in the Grandpa configuration file.

There are two possibilities for the programs activated by the user in Shell environment:

- The context table keeps the values attributed to it by Grandpa when this last has activated the Shell application environment (which becomes the program's 'father').
- The user changes the table's parameters for executing the only program (e.g. redirecting the input and/or output), using the means provided for this purpose by Shell.

The program execution context tables are recorded in files belonging to the memory volume. These files reside in memory so that they can be accessed as quickly as possible and because, as they are transient objects, they need not be retained from one system initialization to another.

When the same program is subsequently reactivated a new table is used, which will take into account the new context in which the program must be executed.

THE PROGRAM DIRECTORY

The program directory is the element of the program execution environment that allows a program to be structured in a series of objects. These objects can be "passive" (data files) or "active" (executable code files).

A program directory is a normal directory in that it is identified by a name selected by the user and it contains a set of files.

It is distinguished from other directories by the way in which it must be created, as well as by the restrictions imposed on the files it contains (which are described below).

Creation

The user is responsible for creating a program directory. After having created a directory (MKDIR command), all the files or directories necessary for executing the program must be placed under it (with the normal commands for file handling: COPY, RENAME, etc).

One of these files must have the predefined name "MAIN". The others can contain portions of executable code that is only loaded in memory when necessary, program input or output data files or directories containing these files, etc.

The names of these files are those by which they are referred to in the program and, if the user wants the program to remain independent of its position in the file system's tree structure, they must not be complete path names (they must not begin with the character "/").

When this directory has been set up, it can be changed into a program directory with the command CHTYPE, which can be called in Shell environment.

The program prepared like this will be activated simply by calling the name of the program directory. The correct connection between the logical names (used by the program) and the physical ones (used by the system for identifying the same resources) is guaranteed by MOS. This means that the program can refer any object (file, directory, volume)

contained in in the program directory independently of its place in the file system tree.

The CONN command can, however, be used and, if it is called, the connections indicated by it are made instead of the automatic connections guaranteed by the program directory. Thus it is possible to define the program execution environment, increasing or modifying the set of objects available to the program. The program directory provides a static environment, while the CONN allows a dynamic handling of it.

As an alternative to the methods described so far, a program directory is created by the OLINK linker when the object code modules produced by the compiled BASIC compiler or by the COBOL compiler are linked. A detailed description of the linker and how it is used is given in Part 1 of the manual Program Development Tools, Reference Manual.

The program directory is a static structure that is configured for each program (even if activated several times). It is recorded on disk so that the connections between the program and the files used by it remain valid for as long as the program exists.

It may be useful and advantageous to use the alias files for the names of the support files contained in the program directory, to move to another file located at some point in the file system tree.

A program that is structured as a program directory can be shared by several users, which means that it can be activated simultaneously by more than one person. The objects contained in the program directory are shared between all the users who activate the program. The same program directory is used for each activation, but the execution context table is different each time.

The 'Overlays'

The program directory optimizes the use of the system. Programs with an "overlay" structure can be created.

A program using overlays optimizes the use of memory as parts of executable codes that are connected in separate files, known as overlays, can be grouped together. Another file, with the predefined name MAIN, contains the code which controls the overlays' activities. In other words, the part of the program contained in MAIN includes the calls to the overlays (for example, for a program written in COBOL, via the CALL statement).

The files in which the overlays are stored do not have predefined names. The user must guarantee their congruency with the names used in MAIN for calling them.

The overlay which is called is loaded in memory to be executed, and when a code part belonging to another overlay is executed this is also loaded and "covers" the contents of the previous overlay.

The program is, optionally, structured in overlay automatically by the system during the linking phase. The user who wishes to divide his program into overlays must split the program code into separate files, individually compile these files and link them, specifying the appropriate option when calling the OLINK linker. See Part 1 of the manual Program Development Tools, Reference Manual for greater detail on creating overlays.

An example: COBOL ICE

The COBOL ICE environment is a typical example of how the program directory structure is used.

The following figure shows the program directory in which the COBOL ICE interpreter is structured.

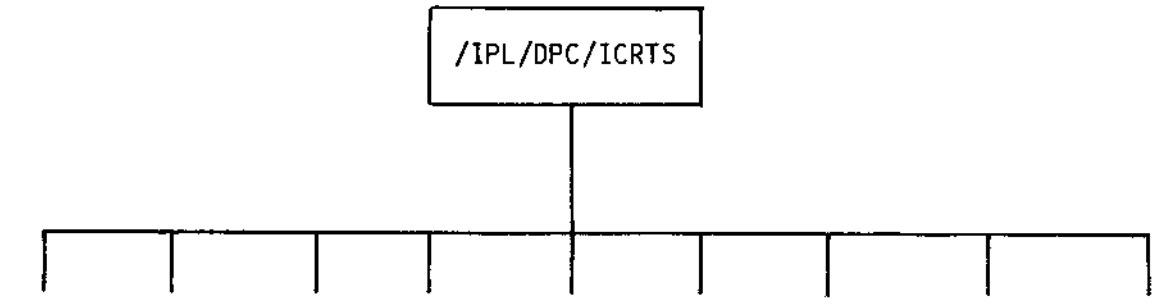


Fig. 2-2 COBOL ICE with Program Directory Structure

The program directory which activates the COBOL ICE interpreter is called ICRTS, and is found under the IPL/DPC directory.

It contains the following objects:

MAIN : is the file containing the ICRTS's executable code.

ICRTSCFM_OB : is the file containing the program which updates the CONTROL file.

PARAM : is the file, created by the PARAM utility, which contains the parameters for redirecting the output.

PARAM_OB : is the file containing the program which handles the parameters contained in the PARAM file. These two files allow the ICRTS spooler to be used.

MESSAGE : is the file containing the error messages.

CONTROL : is the file, created by the ICRTSCFM utility, which contains the values of the ICRTS parameters.

SPOOL_OB : is the file containing the program which handles the functions of the ICRTS spooler.

SPOOL_QUEUE : is the file, created the by ICRTS spooler, which contains the information on the files submitted to the ICRTS spooler.

SPOOL : is the directory containing the files submitted to the COBOL ICE application environment's spooler.

Remarks

When using the spooler (and therefore, inserting new files under the SPOOL directory, contained in the ICRTS program directory), to prevent the volume which contains the COBOL ICE interpreter exceeding its assigned limits, the SPOOL directory can be substituted with an alias file which sends it under another directory, where the files submitted to the COBOL ICE application environment's spooler will be located.

Operating in this way, the user is also guaranteed the possibility of logically disconnecting the volume containing the COBOL ICE interpreter before the spooler has finished printing the files submitted to it.

Note: Refer to the Distributed System in Local Area Network (LAN), User Guide for information relevant to programming in a distributed environment.

FORMAT OF AN L-MODULE

A description of an l-module follows. Those fields whose names start with an upper case letter are expanded into sub-fields as shown below.

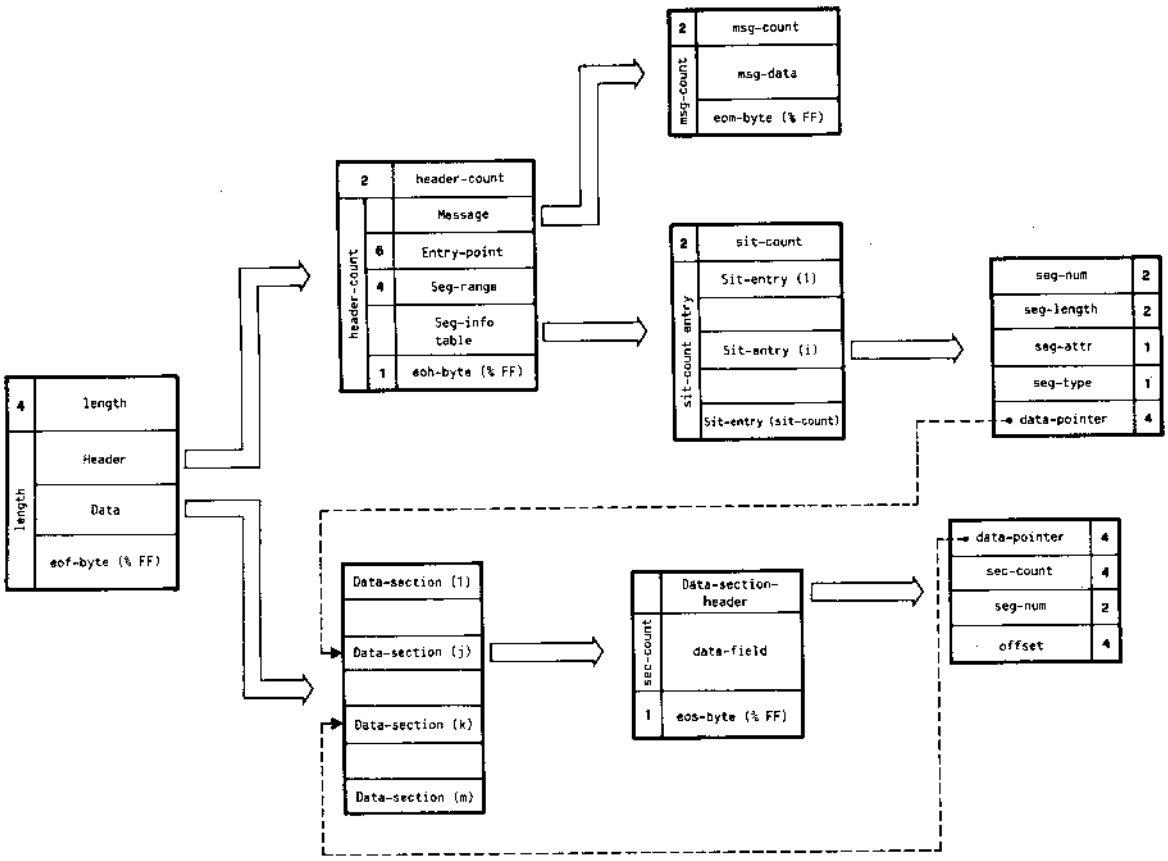


Fig. 2-3 Format of an l_module

Note that the loader interprets the contents of an l-module, sequentially, up to the data-section.

For an explanation on how to display an l-module, see the utility M5LDUMP in Program Development Tools, Reference Manual, and the HEXED utility in the manual System Software Maintenance.

L-Module

An l-module has the following fields:

length:	length of the l-module in bytes.
Header:	l-module Header. It contains general information or a description of the segments used by the program.
Data:	entries containing codes and data of the l-module.
eof-byte:	l-module closing field. Used for error checking.

Header

The l-module Header has the following subfields:

header-count:	header length in bytes.
Message:	field for user to insert information (see the MESSAGE command in the manuals Program Development Tools and PASCAL+ Program Preparation and Execution).
Entry-point:	l-module entry-points expressed as number of segments and offsets.
Seg-range:	number of the lowest and highest segments used by the program.
Seg-info-table:	table showing all the segments used, and information about each of them.
eoh-byte:	Header closing field. Used for error checking.

Data

The Data field of an l-module has the following subfields:

Data-section: entries containing the code and data of the l-module. The data-section entries serve used to initialise the segments used by the l-module. The user should bear in mind that not all the segments can be initialised and also that each segment may not be initialised completely.

Message

The Message in the Header field has the following subfields:

msg-count: length of the message in bytes.

msg-data: area containing the message written by the user. This message can be 60 characters long (max.) if the OLINK linker is used, and 80 if the ZLOC linker is used.

eom-byte: closing field of the message. Used for error checking.

Entry-point

The Entry-point of the Header field has the following fields:

seg-num: number of the segment containing the program entry-point.

offset: offset in the segment of the program entry-point.

Seg-range

The Seg-range of the Header field has the following subfields:

seg-num: number of the lowest segment used by the program.

seg-num: number of the highest segment used by the program.

Seg-info-table

The Seg-info-table of the Header field has the following subfields:

sit-count: entry number of the Seg-info-table.

Sit-entry: Set of entries, one for each segment used, containing specific information for each segment.

Data-section

A Data-section of the Data field has the following subfields:

Data-section-header: header of the Data-section containing information about the data-field field.

data-field: area containing the initialization (code and data) of the segment.

eos-byte: closing field of the Data-section. Used for error checking.

Sit-entry

A generic Sit-entry of the Seg-info-table has the following fields:

seg-num: number of the segment given in the entry.

seg-length: length of the segment in 256-byte pages.

seg-attr: set of attributes for the segment created. Hardware protection for this segment is established by this field (see the ATTRIBUTES command in the manuals Program development Tools, Reference Manual, and PASCAL+ Program Preparation and Execution).

seg-type: types of information in the segment: data, codes, stack, etc. It informs the system about the contents of the segment and the operations that can be performed on it (see the TYPE command in the manuals Program Development Tools, Reference Manual, and PASCAL+ Program Preparation and Execution).

data-pointer: offset in the load-file of the first Data-section that starts that segment.

Data-section-header

The Data-section-header of the Data-section has the following subfields:

data-pointer: pointer to the next Data-section for this segment. If it is the last in the list it, its value is NIL.

sec-count: size of the data-field in bytes.

segnum: segment number.

offset: offset in the segment where the contents of the data-field start.

3. RECOVERY MODES

WRITING MODES ON DISK

The MOS operating system writes on disk using a set of system buffers (whose number and size is defined at configuration time), in which it temporarily stores the data to be transferred onto disk.

The user process requesting a write operation can decide, at open time, which of the following modes to use for each file to be written:

- Synchronous
- Asynchronous

Synchronous Mode

When this mode is used the requesting user process will only regain control when the write operation has finished.

If errors occur while physically writing on disk, the user is immediately informed and can act accordingly.

Asynchronous Mode

When this mode is used the requesting user process does not wait for its completion and receives control immediately having made the request.

Write operations are asynchronously carried out on disk when the buffer in which they are temporarily stored is full or when the file is closed (if a file has been opened by several users, this is when it has been closed by all the users).

If errors occur during the physical writing on disk, as the system cannot identify the program which has requested the write operation, it communicates the anomaly by displaying a message on the master terminal.

Note: If screen splitting has not been requested for any of the terminals, including the master (either the terminal declared as master in the Grandpa configuration file, or the first terminal to be accessed), the asynchronous system messages are not displayed and they are lost.

When this message is displayed, in FDU configurations, the user must replace the disk on which the error has occurred, which isolates it and allows other users to continue without incurring risks to their activities.

MOS uses the synchronous mode by default for handling the system files, which are those files automatically connected by the system when a volume is logically connected (bit maps, file descriptor tables, etc) or when files are created or removed (e.g. directories), so providing maximum guarantee of the integrity of the data contained in the system tables.

The user files (byte-stream, positional, keyed) are handled by default in asynchronous mode. Another mode can be chosen, however, at the application environment level. For example, the MTS transactional environment carries out all its operations in synchronous mode. When working in asynchronous mode, the user should close his files every now and again for quicker recovery from a possible system crash.

The mode in which it is accessed is a characteristic of the file, in the sense that it determines the updating mode. Consequently, if a file is shared and accessed, at a particular time, by two environments, one of which is operating in synchronous mode and the other in asynchronous mode, the mode adopted at open time prevails.

Private Buffers

Apart from the two disk writing modes described above, a set of private buffers (defined at configuration time) can be used. By associating a private buffer to an output file (SETBUFF Shell command) the user avoids concurrency with other programs that use the system buffers. This results in an increase of the processing speed.

DATA INTEGRITY

In the MOS operating system all calls to the file system are atomic from the user point of view. This means that as far as the caller is concerned, either all the requested service are be performed or none of them are. In other words, file system data structures are only visible when they are consistent.

A consequence is that if the request is not completed (for any reason internal or external to the system), then it is possible for the caller to repeat the request with the same external results as if the request were being made for the first time.

To render these features fully available at application program level, a particular mechanism is supplied: the Commit Manager.

THE COMMIT MANAGER

This section describes the Commit Manager, the structures used by it, how it must be installed and how its features can be used.

FUNCTIONS

The Commit Manager allows the user to explicitly define a critical, atomic region. That is, it allows logically related data structures to undergo the same treatment.

Thus if an application program needs to simultaneously update, for example, two or more files, then their updating constitutes a critical region, and the Commit Manager should be invoked.

In this case all file updates are intercepted by the Commit Manager and delayed in a "stage area". When the end of the critical region is reached, the Commit Manager will make all updates permanent. This is achieved through the use of particular tables and log files which the Commit Manager uses. Sufficient information is saved such that if the system were to crash during this phase, then the Commit Manager is able, on restart, to rollback the files to their state before the execution of the critical region.

The Commit Manager must not be called for Positional no Deletion files ("Sequential" for Compiled BASIC and COBOL, "Random" for Interpreted BASIC).

Stage Area

In order to guarantee the consistency of the files, the Commit Manager uses a "stage area" for each work station.

This area is used for storing requests for output operations on application files made by an application program and included in a Commit Region, i.e. a critical region consisting of physically disjointed output operations which are treated as being only one logical operation ("transaction").

These requests are executed together as soon as the program in question requests the closing of the Commit Region (by invoking the EndCommit primitive - see below).

The user should remember that 32 updates (max.) are permitted.

In order to determine the size of the stage area, the transaction that executes most of the output operations, or the operations that use most of the data, must be taken into consideration. In any case, the size of the area must not exceed 16K.

The size of the stage area is calculated in accordance with the following diagram.

HEADING	SYSTEM ID.	PATH NAME	COMMAND HEADING	KEY	RECORD	TAIL
78 bytes	8 bytes	60 bytes	n bytes	key length	record length	5 bytes

|<————PART REPEATED FOR EVERY COMMAND————>|

where "n" may have one of the following values:

10 - if writing to a positional file

8 - if writing to a keyed file

5 - if deleting from a positional file

3 - if deleting from a keyed file

Note: the "KEY" field exists only if writing or deleting on a keyed file. Deleting from a positional file leaves no "KEY" or "RECORD" fields.

Log File

This file is created by the Commit Manager with the name specified by the user via the LOGF parameter in the Grandpa configuration file.

The Log file is used by the Commit Manager for recording all the changes made to application files used by the program executed under the Commit Manager control.

Records are written in the Log file whenever an output operation is carried out on these files.

If changes are made, a record with two fields is written; one contains the image of the record after it was changed, and the second shows the record before the change.

If insertions are made, only one field of the record is written, which contains the record to be inserted.

The Log file can be written in append mode or not depending on what is specified in the Grandpa configuration file by the LOPT parameter. If the append mode is used the record's images are queued in the Log file and as a result the history of the changes made to the application files is recorded. If append mode is not used, the image of the last record only is recorded in the Log file.

It is therefore advisable to write the Log file in append mode because, if the application files are damaged, the information recorded in the Log file guarantees their reconstruction, by means of the RECOVERY utility (see Appendix B).

It is advisable to write the Log file on a separate magnetic unit, or at least on a different volume from the one containing the application files.

Dual Log File

This file is created by the Commit Manager under the name specified by the user, by means of the DUAL parameter, in the Grandpa configuration file. If this is to be done the Log file must be written in append mode. Further, if this file is not in the specified directory, it is created automatically by the Commit Manager.

The Dual Log file is a duplicate of the Log file, and is used if the Log file is not usable because of errors on hardware devices. If a system break-down occurs during the duplicating of the data (from Log to Dual Log), it should be remembered that the Dual Log cannot contain the last update-transaction.

Transaction File

This file is created by the Commit Manager with the name specified by the user via the WFIL parameter in the Grandpa configuration file.

The Transaction file is used by the Commit Manager for recording the code associated by the user (using the StartCommit primitive - see below) to the last successfully completed transaction for each Commit Session (identified by the CommitId, i.e. an identifier associated by the Commit Manager to the user activity - see below).

This information may be retrieved by the application program through a call to a Commit Manager primitive (StatusCommit - see below).

Joucman File

This file is created by the Commit Manager and used by it to keep a trace of the various steps carried out.

Eventual error conditions encountered by the Commit Manager are signalled by means of appropriate messages which are written in this file. The user can read this file by using the DISPJOUC Shell command (see Appendix B).

INSTALLATION AND OPERATING MODE

To exploit the Commit Manager features it is necessary to intervene at three levels:

- In the Grandpa configuration file
- During program preparation
- At program activation time

Grandpa Configuration File

To use the Commit Manager environment correctly, some directives must be inserted in the Grandpa configuration file. They are listed in the System Software Generation and Installation, User Guide manual.

Program Preparation

Any program (or procedure) which will be run under the Commit Manager control must contain the necessary directives to define its "Commit Regions", i.e. critical regions which will be handled by the Commit Manager.

There are four directives which can be inserted in PASCAL+, COBOL, Compiled BASIC and Interpreted BASIC programs.

The four directives start, end and abort a Commit Region and identify the last Commit Region successfully ended.

A Commit Region consists of all the statements included between a "StartCommit" and a "EndCommit" primitives: write operations (i.e. write, update or delete operations) requested between these two, although physically disjointed, are treated as a single logical operation, i.e. a transaction.

A Commit Session consists of all the Commit Regions (or transactions) carried out by an application program and identified by the same CommitId (see the StartCommit primitive).

Each Commit Region within a Commit Session can be uniquely identified with a user provided code (CommitCode). Then, should the system crash or the program abort, this mechanism allows the application program to retrieve the last successfully completed Commit Region (using the StatusCommit primitive) and thus to establish the right point (i.e. the transaction) from which to restart.

See the section "The Commit Manager Primitives" for a detailed description of the Commit Manager primitives and of how they can be called from within a PASCAL+, COBOL, Compiled BASIC or Interpreted BASIC program.

Program Activation

To run a program under the control of the Commit Manager (assuming that the necessary directives have been inserted in the Grandpa configuration file, and that the suitable directives have been included in the program itself) the following steps must be taken:

- The MCL environment is selected.

- The program is run by entering the following command:

```
COMMIT PROG=progname STGA=stagesize WFIL=name CMAN=COMx
```

where:

progname	is the name of the file containing the executable code. This file can be a remote file.
stagesize	is a 5 character string defining the stage area size in bytes. It must be equal to or less than the value given to the STGA parameter in the Grandpa configuration file.
name	must be the same complete path name of the Transaction File as passed to the Commit Manager via the WFIL parameter in the Grandpa configuration file.
COMx	must be the name (consisting of the three initial characters COM and of a fourth user defined character) which has been assigned, in the Grandpa configuration file, to the Commit Manager under whose control the program is to be run (this means that, in a distributed configuration, this parameter can refer to a Commit Manager resident on a system which is not the current one).

Alternatively, a program may be run under Commit Manager control by activating it directly from Grandpa. The string to be inserted in the Grandpa configuration file follows the same rule. For example, the string:

```
TTYn:/IPL/DPC/CMD/COMMIT,<PROG>progname<STGA>stagesize<WFIL>name<CMAN>COMx;
```

will cause the indicated program to be automatically activated, on TTYn, under the control of the Commit Manager identified by COMx.

RESTART

Provided that all the requirements so far mentioned have been satisfied, three types of restart are provided, if needed, for programs run under Commit:

- Hot restart
- Warm restart
- Cold restart

This means that, as far as Commit Regions are concerned, data integrity is guaranteed and interrupted programs can be restarted from a consistent situation.

Hot Restart

This type of restart is automatically carried out, at run time, if a non blocking error (i.e. neither a SYSTEMERROR nor a HARDWAREERROR) occurs when the Commit Manager is making the changes stored in the Log file permanent, that is when the actual application data base is being updated.

In this case the Commit Manager informs the user about the error occurred and rolls back the file which was being updated to the "before image" situation. This file is, therefore, in the same state as if the updating request was never been made.

The program could check the type of error which occurred and, in this case, retry the failed transaction.

Warm Restart

This type of restart is carried out at system start up.

As soon as it is started by Grandpa, the Commit Manager checks in the Log file whether the last Commit Region had been successfully completed or not.

The before image stored in the Log file is applied to the application file in the case the last Commit Region is not successfully completed.

Cold Restart

Any time that an irrecoverable error (for example, a physical error on a magnetic support) occurs, either during a normal work session or during a warm restart, the Commit Manager informs the user, through the Joucman file, of the type of error and on which data base the error occurred.

In this case, if the user had chosen to write in append in the Log file (LOPT option when starting the Commit Manager in the Grandpa configuration file), then it is possible to rebuild the inconsistent data

base using the RECOVERY utility in the Shell application environment (see Appendix B).

Damaged files must be restored from the last dump and, running this utility, after images recorded in the Log file are applied to them. The final situation consists of updated files, where the only missing transaction is the last (i.e. the one in progress when the irrecoverable error occurred).

Note that, to allow the Commit Manager to rebuild the inconsistent data base, it is necessary to stop the Commit activities. Therefore, before running the RECOVERY utility, the CLOSECOMM Shell command must be entered. When the rebuilding phase is terminated, the OPENCOMM Shell command must be entered to correctly restart the Commit activities. (This restart could also be obtained through a new IPL of the MOS system.)

How to Restore an Unsuccessful Transaction

To be able to restore an unsuccessful transaction it is necessary to know, via the primitive StatusCommit, the CommitCode that locates the last transaction completed. To be able to return the CommitCode, however, the StatusCommit primitive requires the CommitId parameter, which is obtained from the StartCommit primitive. It is therefore essential that the application program saves, at the start of the execution, the CommitId (obtained from the first StartCommit) in a user file; in fact, as the CommitId is different for every activation of a program carried out under Commit, it is necessary to arrange the saving of other CommitIds (each related to a work station) in different files or different fields of the same file.

The user file in which the CommitId is stored must be closed by the application program after the EndCommit primitive.

The following diagram illustrates this concept.

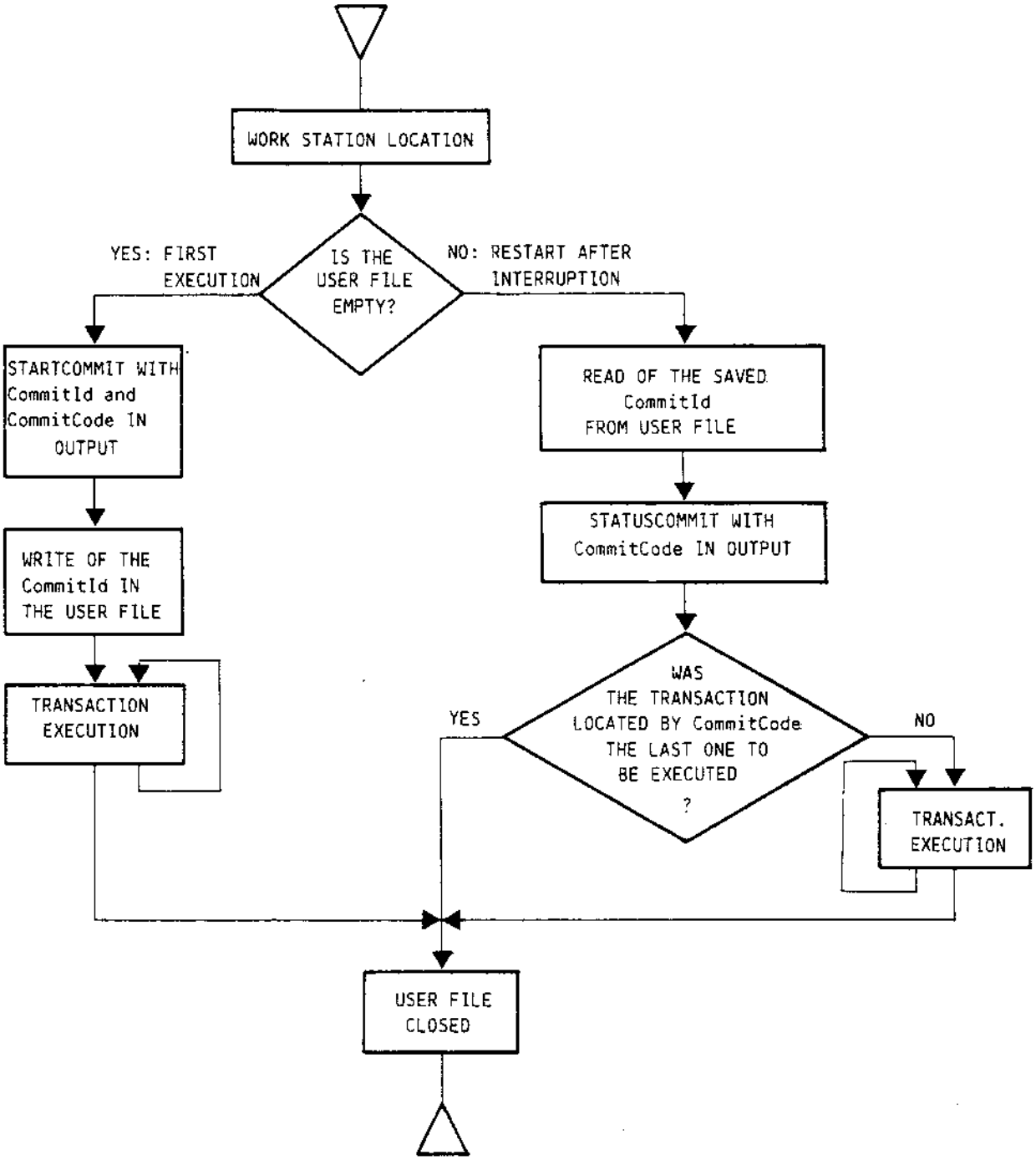


Fig. 3-1 Failed Transaction Restore

THE COMMIT MANAGER PRIMITIVES

This section describes the Commit Manager primitives. They are PASCAL+ procedures which can be called from a PASCAL+ program, a COBOL program, a Compiled BASIC program or an Interpreted BASIC program.

The rules to be followed in order to use these primitives correctly are described below.

The following table summarizes the actions carried out by the Commit Manager primitives, as well as the correct name by which they can be called from each language.

GLOBAL NAME AND ACTION PERFORMED	PASCAL+ NAME	COBOL NAME	COMPILED BASIC NAME	INTERPRETED BASIC NAME
STARTCOMMIT Opens a Commit Region	cm.StartCommit	CM_STARTCOMMIT	cm_StartCommit	STARTCOMMIT
ENDCOMMIT Closes the current Commit Region	cm.EndCommit	CM_ENDCOMMIT	cm_EndCommit	ENDCOMMIT
ABORTCOMMIT Aborts the current Commit Region	cm.AbortCommit	CM_ABORTCOMMIT	cm_AbortCommit	ABORTCOMMIT
STATUSCOMMIT Retrieves the last correctly executed Commit Region	cm.StatusCommit	CM_STATUSCOMMIT	cm_StatusCommit	STATUSCOMMIT

Tab. 3-2 Commit Manager Primitives

Reply Codes

The following table gives a complete list of the reply codes returned by the Commit Manager primitives.

COBOL	COMPILED BASIC	INTERPR. BASIC	PASCAL+	MEANING
0.0	H"0000"	0	CM_CORRECT CM_OKAY	Operation executed correctly.
1.0	H"0100"	256	CM_SYSTEM CM_SYSTEMERROR	System error.
1.1	H"0101"	257	CM_SYSTEM CM_HARDWAREERROR	Hardware malfunction.
1.2	H"0102"	258	CM_SYSTEM CM_OUTOFDISKSPACE	No more space in the volume being used.
1.3	H"0103"	259	CM_SYSTEM CM_DEVICENOTREADY	Device not available.
2.1	H"0201"	513	CM_SUBSYSTEM CM_TIMEOUT	Time expired.
2.4	H"0204"	516	CM_SUBSYSTEM CM_PERMISSION DENIED	Primitive called after the CLOSECOMM command
3.0	H"0300"	1024	CM_USER CM_USERERROR	An inconsistent I/O operation has been requested.
3.4	H"0304"	1028	CM_USER CM_INVALIDOPERATION	Invalid operation.
3.5	H"0305"	1029	CM_USER CM_RECORDNOTFOUND	No valid record at the specified position.
3.6	H"0306"	1030	CM_USER CM_RECORDALREADYEXIST	The record to be inserted already exists.
3.7	H"0307"	1031	CM_USER CM_KEYNOTFOUND	The specified key does not exist.
3.8	H"0308"	1032	CM_USER CM_DUPLICATEDKEY	New duplicate key is is a duplicate.
3.9	H"0309"	1033	CM_USER CM_RECORDOUTOFBOUNDS	The record position is outside the bounds of the file.

Tab. 3-3 Commit Manager Primitives Reply Codes

PASCAL+ INTERFACE

This section presents:

- the general structure for a PASCAL+ call of a Commit Manager primitive
- the description of all the parameters which are involved in the Commit Manager primitives.

PASCAL+ Call

The structure of a procedure call is:

```
procedure_name (parameters_list);
```

where:

procedure_name is the name of a Commit Manager PASCAL+ procedure (see Table 3.1 "Commit Manager Primitives").

parameters_list is the list of parameters for the called primitive.

In order to be able to access a Commit Manager primitive, the PASCAL+ program must import them from the module in which they are supplied. This is achieved by including in the source program the following files:

CM_p.i which contains the declaration of the import and the definition of the primitives implemented by the Commit Manager.

CM_t.i which contains the declaration of the types known by the Commit Manager.

systypes.i which contains the declarations of the system types used by by the Commit Manager.

Furthermore, the following file must be linked to the application program:

CM_ui.obj

Parameter Description

The following tables give the structure and description of all the parameters used by the Commit Manager PASCAL+ primitives. Parameters are ordered alphabetically.

PARAMETER	DEFINITION
INPUT	
CommitCode	CommitCode : longinteger; (* passed by var *)
CommitId	T_systemId = record (* passed by var *) net : T_id; local : T_id; end; CommitId : T_systemId;
ComTimeOut	ComTimeOut : longinteger; (* passed by var *)
OUTPUT	
CommitCode	CommitCode : longinteger; (* passed by var *)
CommitErrNum	CommitErrNum : integer; (* passed by var *)
CommitId	T_systemId = record (* passed by var *) net : T_id; local : T_id; end; CommitId : T_systemId;
RetCode	T_cmclass = (CM_CORRECT, CM_SYSTEM, (* passed by var *) CM_SUBSYSTEM, CM_USER); T_cmcorrect = (CM_OKAY); T_cmsyserr = (CM_SYSTEMERROR, CM_HARDWAREERROR, CM_OUTOFDISKSPACE, CM_DEVICENOTREADY); T_cmsubsyserr = (CM_SUBSYSERROR, CM_TIMEOUT, CM_LINEDOWN, CM_WSACTIVE, CM_PERMISSIONDENIED); T_cmusererr = (CM_USERERROR, CM_INVALIDPARAMETERS, CM_INVALIDCOMMAND, CM_MSGTOOLONG, CM_INVALIDOPERATION, CM_RECORDNOTFOUND, CM_RECORDALREADYEXIST, CM_KEYNOTFOUND, CM_DUPLICATEDKEY, CM_RECORDOUTOFBOUNDS, CM_RESOURCEALREADYLOCKED, CM_NOMSG, CM_SECURITYVIOLATION, CM_NAMENOTFOUND, CM_FATALMISMATCH, CM_NONFATALMISMATCH);

(Cont.)

PARAMETER	DEFINITION
	T_cmreply = record case class : T_cmclass of CM_CORRECT : (correctCode : T_cmcorrect); CM_SYSTEM : (sysCode : T_cmsyserr); CM_SUBSYSTEM : (subsysCode : T_cmsubsyserr); CM_USER : (userCode : T_cmusererr); end;

Tab. 3-4 PASCAL+ Parameters Description

PARAMETER	MEANING
CommitCode	User identifier of the current Commit Region.
CommitErrNum	Number of the failed suboperation in a Commit Region whose EndCommit primitive returned an error condition.
CommitId	Identifier used by the Commit Manager to identify the current Commit Region inside the user activity.
ComTimeOut	Number of second tenths the user is prepared to wait for the completion of a Commit Region.
RetCode	Reply code.

Tab. 3-5 PASCAL+ Parameters Meaning

COBOL INTERFACE

This section presents:

- the general structure for a COBOL call of a Commit Manager procedure
- the description of all the parameters which are involved in the Commit Manager procedures.

COBOL Call

The structure of a COBOL procedure call is:

```
CALL "literal" USING parameters_list
```

where:

literal is the COBOL name of a Commit Manager procedure (see Table 3.1 "Commit Manager Primitives").

parameters_list is the list of parameters for the called procedure.

In order to access a Commit Manager service, the application program must be compiled using the "TH" option of the COBOL compiler.

Furthermore, the COBOL program must link the EXTINTF.LIB library in order to use the Commit Manager primitives. The linking is automatically done when using the RTLINK file, which contains the default directives for the linker.

Parameter Description

The following table gives the structure and description of all the parameters used by the Commit Manager COBOL procedures. Parameters are ordered alphabetically.

PARAMETER	DEFINITION	MEANING
INPUT		
CommitCode	01 COMMITCODE PIC 9(9) COMP.	User identifier of the current Commit Region.
CommitId	01 COMMITID PIC X(8).	Identifier used by the Commit Manager to identify the current Commit Region inside the user activity.
ComTimeOut	01 COMTIMEOUT PIC 9(9) COMP.	Number of second tenths the user is prepared to wait for the completion of a Commit Region.
OUTPUT		
CommitCode	01 COMMITCODE PIC 9(9) COMP.	User identifier of the current Commit Region.
CommitErrNum	01 COMMITERRNUM PIC 9(4) COMP.	Number of the failed suboperation in a Commit Region whose EndCommit returned an error condition.
CommitId	01 COMMITID PIC X(8).	Identifier used by the Commit Manager to identify the current Commit Region inside the user activity.
RetCode	01 RETCODE. 02 CLASS PIC 99 COMP. 02 SUBCLASS PIC 99 COMP.	Reply code.

Tab. 3-6 COBOL Parameters Description

COMPILED BASIC INTERFACE

This section presents:

- the general structure for a Compiled BASIC call of a Commit Manager primitive
- the description of all the parameters which are involved in the Commit Manager primitives.

Compiled BASIC Call

The structure of a Compiled BASIC call is:

```
10 CALL literal (parameters_list)
```

where:

literal is the Compiled BASIC name of a Commit Manager primitive (see Table 3.1 "Commit Manager Primitives").

parameters_list is the list of parameters for the called primitive.

In order to access the Commit Manager services, the following statements must be inserted in the application program:

```
100 DECLARE SYSTEM PROCEDURE cm_StartCommit (CommitId$*8, &
&  CommitCode*9, RetCode$*2)

110 DECLARE SYSTEM PROCEDURE cm_EndCommit (CommitId$*8, &
&  ComTimeOut*9, CommitErrNum*1, RetCode$*2)

120 DECLARE SYSTEM PROCEDURE cm_AbortCommit (CommitId$*8, RetCode$*2)

130 DECLARE SYSTEM PROCEDURE cm_StatusCommit (CommitId$*8, &
&  CommitCode*5.0, RetCode$*2)
```

Furthermore, the Compiled BASIC program must link the extintf.lib library in order to use the Commit Manager primitives. The linking is automatically done when using the link.cmd file, which contains the default directives for the linker.

Parameter Description

The following table gives the structure and description of all the parameters used by the Commit Manager Compiled BASIC primitives. Parameters are ordered alphabetically.

PARAMETER	DEFINITION	MEANING
INPUT		
CommitCode	10 DECLARE NUMERIC CommitCode*9	User identifier of the current Commit Region.
CommitId	20 DECLARE STRING CommitId$*8	Identifier used by the Commit Manager to identify the current Commit Region inside the user activity.
ComTimeOut	30 DECLARE NUMERIC ComTimeOut*9	Number of seconds tenths the user is prepared to wait for the completion of a Commit Region.

(Cont.)

PARAMETER	DEFINITION	MEANING
OUTPUT		
CommitCode	40 DECLARE NUMERIC CommitCode*9	User identifier of the current Commit Region.
CommitErrNum	60 DECLARE NUMERIC CommitErrNum*1	Number of the failed suboperation in a Commit Region whose EndCommit returned an error condition.
CommitId	50 DECLARE STRING CommitId$*8	Identifier used by the Commit Manager to identify the current Commit Region inside the user activity.
RetCode	70 DECLARE STRING RetC$*2	Reply code. Its format is of the type hexadecimal-string--constant (a sequence of an even number of hexadecimal characters enclosed with quotation marks and preceded by H).

Tab. 3-7 Compiled BASIC Parameters Description

INTERPRETED BASIC INTERFACE

This section presents:

- the general structure for an Interpreted BASIC call of a Commit Manager primitive
- the description of all the parameters which are involved in the Commit Manager primitives.

Interpreted BASIC Call

The structure of an Interpreted BASIC procedure call is:

```
10 CALL "literal", parameters_list
```

where:

literal is the Interpreted BASIC name of a Commit Manager primitive (see Table 3.1 "Commit Manager Primitives").

parameters_list is the list of parameters for the called primitive.

Parameter Description

The following table gives the structure and description of all the parameters used by the Commit Manager Interpreted BASIC primitives. Parameters are ordered alphabetically.

PARAMETER	DEFINITION	MEANING
INPUT		
CommitCode	10 CMTCODE#(1)=1	User identifier of the Current Commit Region.
CommitId	20 CMTID#(1)=0	Identifier used by the Commit Manager to identify the current Commit Region inside the user activity.
ComTimeOut	30 CMTTIMEOUT(1)=40	Number of second tenths the user is prepared to wait for the completion of a Commit Region.
OUTPUT		
CommitCode	40 CMTCODE#(1)=1	User identifier of the current Commit Region.
CommitErrNum	60 CMTERRNUM%(1)=0	Number of the failed suboperation in a Commit Region whose EndCommit returned an error condition.
CommitId	50 CMTID#(1)=0	Identifier used by the Commit Manager to identify the current Commit Region inside the user activity.
RetCode	70 RETC0%(1)=0	Reply code. It is an integer in the range from 1 to 32767.

Tab. 3-8 Interpreted BASIC Parameters Description

Note: It is advisable to use the above parameters as elements of an integer array, in order to be sure that their contents are aligned to the word, as requested by the PASCAL+ primitives invoked.

COMMIT MANAGER CALLS

This section gives a detailed description of all the Commit Manager primitives.

They are ordered alphabetically according to their global name.

The description covers:

- the activity carried out by the primitive
- the calling syntax from COBOL, Compiled BASIC, Interpreted BASIC and PASCAL+
- for each parameter, the following aspects:
  - Its name
  - Whether it is an input or an output parameter
  - A brief description
- possible values for the reply code
- eventual characteristics.

ABORTCOMMIT

This primitive aborts the current Commit Region.

COBOL Call

```
CALL "CM_ABORTCOMMIT" USING CommitId, ReplyCode.
```

Compiled BASIC Call

```
10 CALL cm_AbortCommit (CommitId$, ReplyCode$)
```

Interpreted BASIC Call

```
10 CALL "ABORTCOMMIT", CommitId#(1), ReplyCode%(1)
```

PASCAL+ Call

```
cm.AbortCommit (CommitId, ReplyCode);
```

The meaning of all the parameters related to this primitive is described in the following table. Their structure is described in the previous "Parameters Description" sections.

PARAMETER	MEANING
INPUT	
CommitId	Identifier returned by the Commit Manager through the StartCommit primitive and associated with the current user activity.
OUTPUT	
ReplyCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the primitive.

COBOL	COMPILED BASIC	INTERPRETED BASIC	PASCAL+	MEANING
0.0	H"0000"	0	CM_CORRECT CM_OKAY	Operation executed correctly.
1.0	H"0100"	256	CM_SYSTEM CM_SYSTEMERROR	System error.
2.1	H"0201"	513	CM_SUBSYSTEM CM_TIMEOUT	Time expired.
3.4	H"0304"	1028	CM_USER CM_INVALIDOPERATION	Invalid operation.

Characteristic

This primitive allows the user to abort a Commit Region. In order to open a new Commit Region the StartCommit primitive must be called again. The aborted Commit Region has no effect.

ENDCOMMIT

This primitive closes the current Commit Region.

COBOL Call

```
CALL "CM_ENDCOMMIT" USING CommitId, ComTimeOut, CommitErrNum, ReplyCode.
```

Compiled BASIC Call

```
10 CALL cm_EndCommit (CommitId$, ComTimeOut, CommitErrNum, ReplyCode$)
```

Interpreted BASIC Call

```
10 CALL "ENDCOMMIT", CommitId#(1), ComTimeOut(1), CommitErrNum%(1),
ReplyCode%(1)
```

PASCAL+ Call

```
cm.EndCommit (CommitId, ComTimeOut, CommitErrNum, ReplyCode);
```

The meaning of all the parameters related to this primitive is described in the following table. Their structure is described in the previous "Parameters Description" sections.

PARAMETER	MEANING
INPUT	
CommitId	Identifier returned by the Commit Manager through the StartCommit primitive and associated with the current user activity.
ComTimeOut	Number of second tenths the user is prepared to wait for the completion of the Commit Region. The value -1 means an infinite time, whilst the value 0 means that the waiting phase is only started if data to be written on the disk has been successfully received by the system (otherwise an error condition is signalled).
OUTPUT	
CommitErrNum	Number of the failed suboperation in the Commit Region if an error is returned in the Reply Code.
ReplyCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the primitive.

COBOL	COMPILED BASIC	INTERPR. BASIC	PASCAL+	MEANING
0.0	H"0000"	0	CM_CORRECT CM_OKAY	Operation executed correctly.
1.0	H"0100"	256	CM_SYSTEM CM_SYSTEMERROR	System error.
1.1	H"0101"	257	CM_SYSTEM CM_HARDWAREERROR	Hardware malfunction.
1.2	H"0102"	258	CM_SYSTEM CM_OUTOFDISKSPACE	No more space in the volume being used.
1.3	H"0103"	259	CM_SYSTEM CM_DEVICENOTREADY	Device not available.
2.1	H"0201"	513	CM_SUBSYSTEM CM_TIMEOUT	Time expired.
2.4	H"0204"	516	CM_SUBSYSTEM CM_PERMISSION DENIED	Primitive called after the CLOSECOMM command
3.0	H"0300"	1024	CM_USER CM_USERERROR	A non coherent I/O operation has been requested.
3.5	H"0305"	1028	CM_USER CM_RECORDNOTFOUND	No valid record at the specified position.
3.6	H"0306"	1030	CM_USER CM_RECORDALREADYEXIST	The record to be inserted already exists.
3.7	H"0307"	1031	CM_USER CM_KEYNOTFOUND	The specified key does not exist.
3.8	H"0308"	1032	CM_USER CM_DUPLICATEDKEY	New duplicate key is a duplicate.
3.9	H"0309"	1033	CM_USER CM_RECORDOUTOFBOUNDS	The record position is outside the bounds of the file.

Characteristic

This primitive closes the current Commit Region, and allows the Commit Manager to perform the requested output operations on the application file(s).

STARTCOMMIT

This primitive opens a Commit Region.

COBOL Call

```
CALL "CM_STARTCOMMIT" USING CommitId, CommitCode, ReplyCode.
```

Compiled BASIC Call

```
10 CALL cm_StartCommit (CommitId$, CommitCode, ReplyCode$)
```

Interpreted BASIC Call

```
10 CALL "STARTCOMMIT", CommitId#(1), CommitCode(1), ReplyCode%(1)
```

PASCAL+ Call

```
cm.StartCommit (CommitId, CommitCode, ReplyCode);
```

The meaning of all the parameters related to this primitive is described in the following table. Their structure is described in the previous "Parameters Description" sections.

PARAMETER	MEANING
INPUT/OUTPUT	
CommitId	Identifier returned by the Commit Manager and associated with the current user activity. See "Characteristics" below.
INPUT	
CommitCode	User identifier of the current Commit Region.
OUTPUT	
ReplyCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the primitive.

COBOL	COMPILED BASIC	INTERPRETED BASIC	PASCAL+	MEANING
0.0	H"0000"	0	CM_CORRECT CM_OKAY	Operation executed correctly.
1.0	H"0100"	256	CM_SYSTEM CM_SYSTEMERROR	System error.
2.1	H"0201"	513	CM_SUBSYSTEM CM_TIMEOUT	Time expired.
2.4	H"0204"	516	CM_SUBSYSTEM CM_PERMISSION DENIED	Primitive called after the CLOSECOMM command.
3.0	H"0300"	1024	CM_USER CM_USERERROR	A non coherent I/O operation has been requested.
3.4	H"0304"	1028	CM_USER CM_INVALIDOPERATION	Invalid operation.

Characteristics

This primitive opens, inside a program, a Commit Region. After the execution of this primitive, all the output operations are submitted to the Commit Manager.

Opening a Commit Session for the first time, the CommitId parameter must be passed to the Commit Manager with both its fields ("net" and "local") set to 'NIL' (in PASCAL+) or set to the high value "FFFFFFFFFFFFFFFF" (in Compiled BASIC and COBOL). In this case the Commit Manager will reserve a new unique identifier for the current Commit Session. The CommitId returned by the Commit Manager must not be changed by the user, and is to be used in the following Commit operations.

STATUSCOMMIT

This primitive retrieves the user CommitCode which identifies the last Commit Region successfully completed.

COBOL Call

```
CALL "CM_STATUSCOMMIT" USING CommitId, CommitCode, ReplyCode.
```

Compiled BASIC Call

```
10 CALL cm_StatusCommit (CommitId$, CommitCode, ReplyCode$)
```

Interpreted BASIC Call

```
10 CALL "STATUSCOMMIT", CommitId#(1), CommitCode(1), ReplyCode%(1)
```

PASCAL+ Call

```
cm.StatusCommit (CommitId, CommitCode, ReplyCode);
```

The meaning of all the parameters related to this primitive is described in the following table. Their structure is described in the previous "Parameters Description" sections.

PARAMETER	MEANING
INPUT	
CommitId	Identifier returned by the Commit Manager through the StartCommit primitive and associated with the current user activity.
OUTPUT	
CommitCode	User identifier associated with the last correctly executed Commit Region.
ReplyCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the primitive.

COBOL	COMPILED BASIC	INTERPRETED BASIC	PASCAL+	MEANING
0.0	H"0000"	0	CM_CORRECT CM_OKAY	Operation executed correctly.
1.0	H"0100"	256	CM_SYSTEM CM_SYSTEMERROR	System error.
1.1	H"0101"	257	CM_SYSTEM CM_HARDWAREERROR	Hardware malfunction.
2.1	H"0201"	513	CM_SUBSYSTEM CM_TIMEOUT	Time expired.
2.4	H"0204"	516	CM_SUBSYSTEM CM_PERMISSION DENIED	Primitive called after the CLOSECOMM command
3.0	H"0300"	1024	CM_USER CM_USERERROR	A non coherent I/O operation has been requested.
3.4	H"0304"	1028	CM_USER CM_INVALIDOPERATION	Invalid operation.

Characteristics

By means of this primitive it is possible to retrieve the CommitCode of the last correctly executed Commit Region.

The use of this primitive is compulsory after a TimeOut exit condition from the EndCommit primitive.

If this primitive is used in another activity (i.e. not a Commit activity), then the CommitId passed as input parameter must be saved (in the Commit phase) and restored when running the other activity.

PROCEDURE RESTART

To restart MCL procedures, two keywords are provided: ONRST and ENDRST.

The keywords must be paired, but as many pairs as required may be used.

All commands between each pair of these keywords will only be executed during a restart phase. They will be skipped during normal execution of the procedure.

When Shell encounters an ONRST keyword in a procedure for the first time, it creates a log file.

Shell uses this file to contain the full path name of the procedure being executed, its input parameters, the working directory, all the variables (global or local) and the offset of the ONRST keyword within the procedure.

Shell skips all commands until the one immediately following the first ENDRST.

If another ONRST keyword is subsequently encountered, then the offset in the log file of the ONRST keyword and the variables are updated and, again, all commands until the next ENDRST are skipped.

When the whole procedure has been successfully executed, Shell removes the log file.

Should, however, the system crash during the execution of the procedure, then, when the user logs in again, Shell checks for the existence of the log file. If it exists, then Shell assumes that a crash had occurred and restarts the procedure at the ONRST keyword whose location had been stored in the log file. All the commands starting from the ONRST keyword are then executed, including those between the ONRST and ENDRST pair of keywords.

Following the ENDRST keyword, Shell continues to execute the procedure sequentially.

A detailed description of how to use these keywords may be found in the manual MCL, MOS Command Language, User Guide.

Furthermore, if a program activation is requested in an MCL procedure after the ONRST keyword preceded by the TRACE Shell command, as shown in the following example:

```
.
.
TRACE
ONRST
ECHO Restarting PROGRAM
ENDRST
/IPL/SYS/PROGRAM
.
.
```

then Shell also supplies the user with a second file to aid program restart: it creates an entry (named "TRACE") in the program execution context table.

The program must:

- connect and open this file (by means of the "connect" and "open" PASCAL+ functions, or the "OPEN" statement if the program is written in COBOL, etc.)

- check if this file is empty or not. In the former case, it could store in it intermediate states of its execution, or other information useful for its restart; thus, on restart (latter case), the program could interrogate this file and so restart with appropriate parameters and at an appropriate location.

Both these restart files will be automatically created and connected by Shell in the synchronous mode, and they will be part of the program execution context. This means that, if a correct reply code has been received by the caller (the program being executed), then any write operation requested by it on them is guaranteed to be successfully completed.

If the procedure successfully ends, the log file provided by Shell for the MCL procedure is deleted.

The TRACE file, instead, must be explicitly deleted by the user inserting the TRACE "D command at the end of the procedure. It is, however, automatically deleted when the user logs out.

4. MOS APPLICATION SERVICES

This chapter describes some facilities offered by MOS for application purposes.

They concern services which can be called by an application program to:

- build frames on the screen
- log, in a system file, a record of certain execution steps
- produce, if necessary, a dump of the user address space contents
- verify a signature
- send messages (Message Switching).
- perform MCL activities from Shell or Grandpa.
- suspend the modem connection, between a remote work station and the L1 MOS system.

HOW TO BUILD FRAMES

This section describes how a COBOL or Compiled BASIC program can build frames on the screen.

The Frame Characteristics

The frame which can be built may consist of:

- up to 12 horizontal lines
- up to 40 vertical lines

and may be of any dimension, provided they do not exceed the physical dimension of the screen and of the format being used.

Both horizontal and vertical lines are continuous, thus allowing the production of a frame as the following example shows:

```
*          <------ physical dimension of the screen ------>          *

+-----------+-----------+-----+-----------+
|           |           |     |           |
|           |           |     |           |
+-----------+-----------+-----+-----------+
|           |           |     |           |
|           |           |     |           |
|           |           |     |           |
+-----------+-----------+-----+-----------+
|           |           |     |           |
|           |           |     |           |
+-----------+-----------+-----+-----------+

*          <------ physical dimension of the screen ------>          *
```

Fig. 4-1 Example of a Frame

It is the responsibility of the user program to produce a frame on the screen without covering information already displayed, as well as not to display a frame on an existing one, otherwise the previously displayed information is overwritten (completely or partially) by the last created frame.

COBOL INTERFACE

In order to call the frame procedure, the COBOL application program must have already invoked the screen output statement DISPLAY.

The linking for to use the frame procedure is automatically done when using the RTLINK file, which contains the default directives for the linker.

Parameters Description

The following table gives the structure and description of all the parameters used by the COBOL frame procedure.

PARAMETER	DEFINITION	MEANING
INPUT		
HOR-LINES	01 HOR-LINES PIC 9(4) COMP.	Number of horizontal lines in the frame.
VER-LINES	01 VER-LINES PIC 9(4) COMP.	Number of vertical lines in the frame.
HOR-TABLE	01 HOR-TABLE. 02 HOR-LINES PIC 9(4) COMP OCCURS 12 TIMES.	Array of 12 numeric fields consisting of two bytes each (See the Note)
VER-TABLE	01 VER-TABLE. 02 VER-LINES PIC 9(4) COMP OCCURS 40 TIMES.	Array of 40 numeric fields consisting of two bytes each (See the Note)
OUTPUT		
RET-CODE	01 RET-CODE PIC 9(4) COMP.	Reply code.

Tab. 4-2 COBOL Parameters Description

Note: The meaning of the two parameters HOR-TABLE and VER-TABLE is as follows:

HOR-TABLE: each field specifies the row (in the range from 1 to 24), starting from the top of the screen, where a horizontal line is to be drawn.

VER-TABLE: each field specifies the column (in the range from 1 to 80), starting from the left margin of the screen, where a vertical line is to be drawn.

COMPILED BASIC INTERFACE

In order to call the frame procedure, the following statement must be inserted in the application program:

```
10 DECLARE SYSTEM PROCEDURE TABLE_GRID (HOR-LINES * I, VER-LINES * I, &
&                    HOR-TABLE() * I, VER-TABLE() * I, RET-CODE * I )
```

The linking for to use the frame procedure is automatically done when using the link.cmd file, which contains the default directives for the linker.

Parameters Description

The following table gives the structure and description of all the parameters used by the Compiled BASIC frame procedure.

PARAMETER	DEFINITION	MEANING
INPUT		
HOR-LINES	20 DECLARE NUMERIC HOR-LINES * 1	Number of horizontal lines in the frame.
VER-LINES	30 DECLARE NUMERIC VER-LINES * 1	Number of vertical lines in the frame.
HOR-TABLE	40 DECLARE NUMERIC HOR-TABLE(12) * 1	Array of 12 numeric fields consisting of two bytes each (See the Note).
VER-TABLE	50 DECLARE NUMERIC VER-TABLE(40) * 1	Array of 40 numeric fields consisting of two bytes each (See the Note).
OUTPUT		
RET-CODE	60 DECLARE NUMERIC RET-CODE * 1	Reply code.

Tab. 4-3 Compiled BASIC Parameters Description

Note: The meaning of the two parameters HOR-TABLE and VER-TABLE is as follows:

HOR-TABLE: each field specifies the row (in the range from 1 to 24), starting from the top of the screen, where a horizontal line is to be drawn.

VER-TABLE: each field specifies the column (in the range from 1 to 80), starting from the left margin of the screen, where a vertical line is to be drawn.

This procedure displays a frame.

COBOL Call

```
CALL "TABLE_GRID" USING HOR-LINES, VER-LINES, HOR-TABLE, VER-TABLE,
                        RET-CODE.
```

Compiled BASIC Call

```
CALL TABLE_GRID (HOR-LINES, VER-LINES, HOR-TABLE(), VER-TABLE(),
                 RET-CODE)
```

PARAMETER	MEANING
INPUT	
HOR-LINES	Number of horizontal lines in the frame.
VER-LINES	Number of vertical lines in the frame.
HOR-TABLE	Array of 12 numeric fields consisting of two bytes each. Each field specifies the row, in the range from 1 to 24, starting from the top of the screen, where a horizontal line is to be drawn.
VER-TABLE	Array of 40 numeric fields consisting of two bytes each. Each field specifies the column, in the range from 1 to 80, starting from the left margin of the screen, where a vertical line is to be drawn.
OUTPUT	
RET-CODE	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the frame procedure.

COBOL	COMPILED BASIC	MEANING
0	0	Operation executed correctly.
1	1	The frame procedure has not been preceded by a "DISPLAY" procedure.
2	2	The values given to the HOR-TABLE or VER-TABLE are out of the range.
3	3	System error.

LOGGING FACILITIES FOR A USER PROGRAM

Besides the mechanism (LMS) provided by MOS to keep track of system hardware and software failures, the possibility of logging errors or events which occur during execution is offered to application programs written in the COBOL or PASCAL+ language.

Errors due to the user (for example, inconsistencies of the run time environment provided, logical errors contained in the application program, errors in I/O) are expected to be handled by the application program itself, and they should not be logged.

On the other hand, the application program can add a record in the LOG file to log events, anomalies or errors detected (for example, start and end time of an activity).

In order to do this, a specific primitive is provided and the rules to be followed when using it are given below.

Log Modes

There are two possible log modes:

- Normal, in which records are logged only if the LogFlag and/or AlarmFlag field of the record is set to true (see below).

- Extended, in which all records are logged.

The log mode is specified in the Grandpa configuration file.

The LOG Record Structure

Three types of log record are defined: ERROR, EVENT and DATA, each type being identified by a parameter within the record. No significance is attached to the record type as far as the logging process is concerned; this is meaningful only in subsequent analyses of the log file contents.

All log records are of a fixed length of 100 bytes and consist of two distinct parts:

- The first, common to all three types of record, is 38 bytes in length and contains a fixed number of parameters that define the record.

- The second is 62 bytes in length, but contains variable amounts of data, as received from the application program.

In order to save time in the processing of log data, a parameter in the first part specifies how many bytes are actually used in the second part.

Each record contains the following fields:

FIELD	LENGTH	PURPOSE	WRITTEN BY
alarm flag	1 byte	Specifies whether or not an alarm is to be generated by this record (TRUE = generate alarm).	application
log flag	1 byte	When log mode is set to normal, specifies whether to log this record or to ignore it (TRUE=log it).	application
log time	4 bytes	Indicates the date and time at which the record was logged.	system
resource manager type	2 bytes	Indicates the type of resource manager that requested the log.	system
log code	1 byte	Specifies the type of log (ERROR, DATA, EVENT).	application
machine id	1 byte	A character that identifies the system type.	system
resource address	4 bytes	In a distributed system this is the name of the machine (NnMm) containing the involved resource.	system
resource name	14 bytes	The logical name of the resource involved with this record (i.e. the application program).	system
resource type	2 byte	Identifies the resource type related with this log. This is the same as the resource manager type, i.e. "MOS APL ".	system
record level	2 bytes	Indicates the alarm level group with which this record is associated.	application
record code	2 byte	The error or event code.	application
unused	2 bytes	Reserved for future use.	system
string length	2 bytes	The number of data bytes actually used in the variant part of the record, i.e. in the log string.	application
log string	62 bytes max.	Variant part of the record which contains the data received from the application program.	application

Tab. 4-4 Structure of the LOG Record

Alarm Groups and Levels

Alarms generated by the system are grouped into four different categories, each corresponding to a distinct level of severity. In turn, each of the four alarm levels is assigned a unique range of numbers that may be used to further indicate the importance of a particular alarm within a given alarm group.

The alarm group classification and alarm level ranges normally used by LMS are summarized below.

ALARM GROUP	ALARM LEVEL RANGE	CLASSIFICATION
A	0-99	Alarms for errors that cause blocking of the system, for instance the non-availability of a vital resource.
B	100-199	Alarms relating to errors that have not been recovered, but which do not cause the system to be blocked.
C	200-299	Alarms relating to errors that have been recovered, but with a large penalty (therefore indicating the degradation of a resource).
D	300 -399	All other alarms, in particular those introduced as a result of program, or operator errors.

Tab. 4-5 Alarm Groups and Classifications

Alarm Routing

Alarms generated by the system may be transmitted to an external control centre of the network management services, and/or to a local L1 MOS system that is designated as a "Master Work Station" (MWS). The decision as to which alarm is directed to which destination is made at system configuration time.

With regard to the present software release, however, it should be noted that, from the point of view of satellite systems in a cluster, the "Master Work Station" can only be on the master system of the same cluster.

Furthermore, the COBOL program must link the EXTINTF.LIB library in order to use the log procedure. The linking is automatically done when using the RTLINK file, which contains the default directives for the linker.

Parameters Description

The following table gives the structure and description of all the parameters used by the user log procedure. Parameters are ordered alphabetically.

PARAMETER	DEFINITION	MEANING
INPUT		
AlarmFlag	01 AlarmFlag PIC 9 COMP.	Set to TRUE (1) or FALSE (0) according to whether or not an alarm is to be generated.
LogCode	01 LogCode PIC 99 COMP.	Set to the type of log record (0 =EVENT, 1 = ERROR, 2 = DATA).
LogFlag	01 LogFlag PIC 9 COMP.	If the log mechanism is working in normal mode, set to TRUE (1) or FALSE (0) according to whether the record is to be logged or ignored.
logString	01 logString PIC X(40).	Space for user defined error information.
RecCode	01 RecCode PIC S9(4) COMP.	Identifies the error code.
RecLevel	01 RecLevel PIC S9(4) COMP.	Set to the relative importance code of the record. (See above "Alarm Groups and Levels".)
stringlen	01 stringlen PIC S9(4) COMP.	Specifies the length in bytes of logString.
OUTPUT		
reply	01 reply. 02 reply-class PIC 99 COMP. 02 reply-code PIC 99 COMP.	Reply code.

Tab. 4-6 COBOL Parameters Description

Note: The stringlen parameter must contain the value 40 before issuing the CALL statement to invoke the user log procedure. (A VALUE clause can be used in the WORKING-STORAGE SECTION).

PASCAL+ INTERFACE

In order to be able to access the user log primitive, the PASCAL+ program must import it from the module in which it is supplied. This is achieved by including in the source program the following files:

usrlog.d	which contains the definition of the primitive and its types available for the user log.
systypes.i	which contains the declaration of all the common system types and is needed for the reply code definition.

Furthermore, the following file must be linked to the application program:

lms_ui.obj

Parameters Description

The following table gives the structure and description of all the parameters used by the user log primitive. Parameters are ordered alphabetically.

PARAMETER	DEFINITION	MEANING
INPUT		
AlarmFlag	AlarmFlag : boolean;	Set to TRUE (1) or FALSE (0) according to whether or not an alarm is to be generated. Passed by value.
LogCode	T_LogCode = (EventLog, ErrorLog, DataLog); LogCode : T_LogCode;	Set to the type of log record (EventLog, ErrorLog, DataLog). Passed by value.
LogFlag	LogFlag : boolean;	If the log mechanism is working in normal mode set to TRUE (1) or FALSE (0) according to whether the record is to be logged or ignored.
logString	const maxdata = 62; T_LogString = packed array [1..maxdata] of char; logString : T_LogString;	Space for user defined error information. Passed by var.
RecCode	RecCode : integer;	Identifies the error code. Passed by value.
RecLevel	RecLevel : integer;	Set to the relative importance code of the record. (See above the "Alarm Groups and Levels" section.) Passed by value.
stringlen	stringlen : integer;	Specifies the length in bytes of logString. Passed by value.
OUTPUT		
reply	reply = T_reply; (* defined in systypes.i *)	Reply code.

Tab. 4-7 PASCAL+ Parameters Description

WRITEUSRLOG

This primitive adds a record in the LOG file.

Symbolic names have been assigned to the COBOL parameters and are in lower case. The user may substitute these with more suitable names.

COBOL Call

```
CALL "LOG.WUSRLOG" USING AlarmFlag, LogFlag, LogCode, RecLevel,
                         RecCode, stringlen, logString, reply.
```

PASCAL+ Call

```
reply := log.WriteUsrLog (AlarmFlag, LogFlag, LogCode, RecLevel,
                          RecCode, stringlen, logString);
```

PARAMETER	MEANING
INPUT	
AlarmFlag	Set to TRUE (1) or FALSE (0) according to whether or not an alarm is to be generated.
LogFlag	If the log mechanism is working in normal mode, set to TRUE (1) or FALSE (0) according to whether the record is to be logged or ignored.
LogCode	Set to the type of log record. (See above the "Parameters Description" section.)
RecLevel	Set to the relative importance code of the record. (See above the "Alarm Groups and Levels" section.)
RecCode	Identifies the error code.
stringlen	Specifies the length in bytes of logString.
logString	Space for user defined error information.
OUTPUT	
reply	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the user log primitive.

COBOL	PASCAL+ REPLY CLASS	PASCAL+ REPLY	MEANING
0.0	CORRECT	OKAY	Operation executed correctly.
1.3	SYSTEM	HARDWAREERROR	Hardware malfunction.
1.4		SYSTEMERROR	System error.
1.6		OUTOFDISKSPACE	No more space on disk.
2.1	DEBUG	INVALIDOPERATION	Invalid operation.
2.2		INVALIDID	Invalid identifier.
2.3		INVALIDPARAMETERS	Invalid parameters.
2.5		SECURITYVIOLATION	Operation incompatible with the protection status of the file.
6.0	FSERROR	TIMEOUT	Time expired.
6.2		DEVICENOTREADY	Device not available.

USER ADDRESS SPACE DUMP

After a fatal event has occurred during a user program execution, it could be useful to save an image of the memory contents in a disk file, in order to examine it subsequently.

The part of memory whose contents are to be dumped is the "user address space", that is the set of user segments (see later the "Segments, Families and Processes" chapter for greater detail about memory segments) currently in use.

A specific set of procedures is provided, which can be called by a user program in order to include the possibility of executing, if necessary, a user address space dump.

A utility is provided to interpret, display and print the dump. See Appendix A.

DUMP DEFINITION

The user address space dump is the process of saving the image of all the user active segments into a file named "dump", which is associated with the program execution context relevant to the program being dumped. This file is created if it does not exist already, or its contents are replaced otherwise.

Greater detail about the dump file and its contents is given in Appendix A.

DUMP ACTIVATION

As far as a user program is concerned, a fatal event is an error after which the user program must be stopped. Such an error can be detected both by the system and by the run time support of the language used in the user program. In both cases the user program is notified by means of an exception.

The dump activation after a fatal event is, therefore, based on the exception mechanism.

A set of primitives is provided which allow the user to:

- notify the system that, in case of fatal event, the user address space dump must be activated (EnableDump)
- reset the previous notification (DisableDump)
- explicitly activate the user address space dump (MemoryDump).

The user address space dump primitives are PASCAL+ procedures which can be called from a PASCAL+ program.

The rules to be followed in order to use these procedures correctly are described below.

PASCAL+ INTERFACE

The PASCAL+ primitives are invoked as follows:

```
primitive_name;
```

where:

primitive_name is the name of a user address space dump procedure.

In order to be able to access a user address space dump procedure, the PASCAL+ program must import them from the module in which they are supplied. This is achieved by including the following file in the source program:

dump.d which contains the definition of the procedures available for the user address space dump.

Furthermore, the following file must be linked to the application program:

PM_Dump.obj

USER ADDRESS SPACE DUMP CALLS

This section gives a detailed description of all the user address space dump procedures.

They are ordered alphabetically, and the description covers:

- the activity carried out by the procedure
- the calling syntax from PASCAL+
- eventual characteristics.

DISABLEDUMP

This procedure disables the dump activity.

PASCAL+ Call

```
PM_Dump.DisableDump;
```

Characteristic

This procedure resets the exception handler for the calling program to the value before the invocation of the EnableDump procedure.

ENABLEDUMP

This procedure enables the dump activity.

PASCAL+ Call

```
PM_Dump.EnableDump;
```

Characteristic

This procedure substitutes a new exception handler to the current one. When activated by exception, the new handler performs a user address space dump before invoking the previous exception handler.

MEMORYDUMP

This procedure performs the user address space dump of the calling program.

PASCAL+ Call

```
PM_Dump.MemoryDump;
```

SIGNATURE VERIFICATION

The MOS provides two PASCAL+ procedures for verifying signatures. These procedures respectively allow to acquire the signature to be verified, written on sheet, by an optical reader (SCANNER) in MHC (Modified Huffman Code), and to decompress it in bitmap into normal or zoomed format. The signature can be displayed by means of the PGU (Graphics Management Package) "PUT" procedure.

The "Signature verification" functional diagram is as follows:

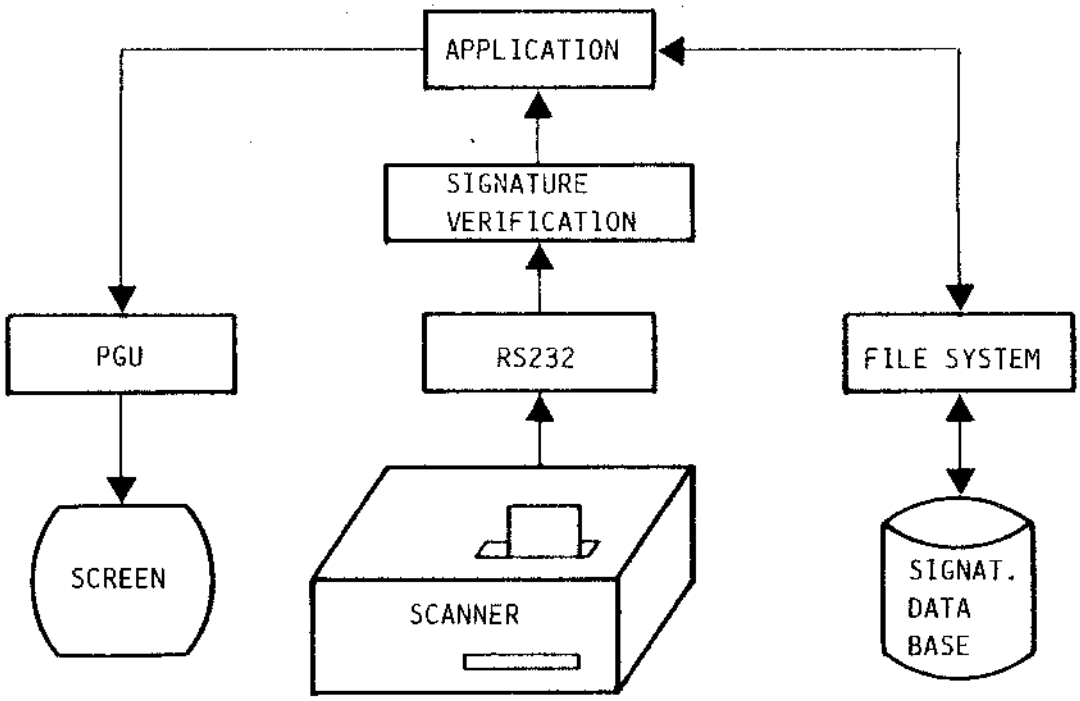


Fig. 4-8 Functional Diagram of the Signature Verification Feature

The application program:

- executes the SCANNER procedure to acquire the signature in MHC code and receives it in a user defined area.

- executes the BITMAP procedure, which obtains the two bitmaps of the signature from the MHC code (with and without zoom) and returns them to the application program

- calls the PGU, opens a graphics session (necessary for signature display) and executes other PGU control functions

- executes the PGU "PUT" procedure to display the signature

- stores, if considered appropriate, the signature MHC code, contained in an array, on an external keyed file (key = client code) by means of the COBOL or PASCAL+ I/O instructions. It is also possible to store the signature in the form of a bitmap, bearing in mind that this solution occupies four times as much space as storing it in MHC code. When storing the signature in MHC code, before displaying it through the PGU procedure, it is necessary to execute the BITMAP procedure.

CONFIGURATION PARAMETERS FOR THE SCANNER

At the configuration phase of the system comprising the optical reader, it is necessary to specify, for the parameters relative to the RS232/CL (UNIT 5) driver, the following values:

```
RSDRCHKEY    = TxyO   (RS232 interface type: TWIN; see the note)
INITRXBUF    = 256    (dimensions of the driver reception buffer)
INITTXBUF    = 512    (dimensions of the driver transmission buffer)
PGMRXTXBL    = %0B    (transmission speed: 4800 baud)
OUTSTDTRRTS = %0102   (line control signal: DTR)
RXTXCHLK     = %0303  (number of bits per character: 8)
STOPBPAR     = %0000  (number of stop bits: 1; parity check not required)
INITXOFFLEV = 192     (transmission check parameter)
INITXONLEV   = 128    (transmission check parameter)
RSDROFFCTL1 = %0101   (the DSR and CTS signals are ignored)
RSDROFFCTL2 = %01     (the DCD signal is ignored)
```

Note: the letter x has the value A or B depending on the board channel.

The letter y has one of the values A, B, C.... depending on the positions of the board relative to the other TWIN boards that may be present. For further details on the above parameters refer to the manual System Software Generation and Installation, User Guide

Signature Verification Procedures

The two procedures available for signature verification are the following:

- SCANNER, to acquire the signature by an optical reader, in MHC code.
- BITMAP, which decompresses the MHC code of the signature into bitmap.

PGU Procedures to be Called

The following PGU procedures need to be called the application program in order to obtain signature display:

- OPENPGU to open the PGU session and be able to use the graphics procedures.
- BRESET to cause immediate display procedure execution.
- PUT for signature display.
- CLOSEPGU to close the PGU session.

In addition, the SETALPHA or SETGRAPH procedures must be called if the user who is not using the PGU alphanumeric I/O procedures wishes to use the alphanumeric or graphics bitmap for the alphanumeric output.

When using an OLIVETTI PC as work station it is possible to have all the PGU "output" functions on the PC, thus obtaining an improvement in performance due to the increased speed of the signature display.

See below for the description of the PGU Procedures listed above.

For further information on the PGU, see the manual PGU, Graphics Management Package, Programmer Guide.

COBOL INTERFACE FOR PGU CALL

Furthermore, the COBOL program must link the EXTINTF.LIB library in order to use the PGU procedures. The linking is automatically done when using the RTLINK file, which contains the default directives for the linker.

Parameters Description

The following table gives the COBOL structure and description of the parameters used by the PGU procedures. The parameters are listed in alphabetical order.

PARAMETER	DEFINITION	MEANING
INPUT		
openPgu	77 openPgu PIC 9(2) COMP.	PGU initialization mode.
putType	77 putType PIC 9(2) COMP.	Specifies where the signature is to be displayed.
startIndex	77 startIndex PIC S9(4) COMP.	Ordinal value of the first byte of the array containing the signature to be displayed.
store	77 store PIC X(nn).	Array containing the bitmap of the signature to be displayed.
storeSize	77 storeSize PIC S9(9) COMP.	Number of bytes of array containing the signature to be displayed.
x1,x2,y1,y2	77 x1 COMP-1. 77 x2 COMP-1. 77 y1 COMP-1. 77 y2 COMP-1.	Coordinates of the upper left hand and lower right hand corners of area where the signature is to be displayed.
OUTPUT		
retcode	77 retcode PIC S9(4) COMP.	Reply code.

Tab. 4-9 COBOL Parameters Description

PASCAL+ INTERFACE FOR PGU CALL

In order to be able to access the PGU procedures, the PASCAL+ program must include the module:

PGU.d

Furthermore, the following interface must be linked to the application program:

P_PGU_ui.obj

Parameters Description

The following table gives the PASCAL+ structure and description of the parameters used by the PGU procedures. The parameters are listed in alphabetical order.

PARAMETER	DEFINITION	MEANING
INPUT		
openPgu	T_PguOpenMode = (i_t, e_t, i_s, e_s); openPgu : T_PguOpenMode;	PGU initialization mode. Passed by value.
putType	T_put_mode = (none, first, firstandsecond); putType : T_put_mode;	Specifies where the signature is to be displayed. Passed by value.
startIndex	startIndex : integer;	Ordinal value of the first byte of the array containing the signature to be displayed. Passed by value.

(Cont.)

PARAMETER	DEFINITION	MEANING
store	byte = - 128..127; store : packed array [min..max: integer] of byte;	Array containing the bitmap of the signature to be displayed. Passed by VAR.
x1,x2,y1,y2	x1 : real; x2 : real; y1 : real; y2 : real;	Coordinates of the upper left hand and lower right hand corners of the area where the signature is to be displayed. Passed by value.
OUTPUT		
retcode	retcode : integer;	Reply code.

Tab. 4-10 PASCAL+ Parameters Description

PGU PROCEDURES

This section describes the PGU procedures needed to open the graphics session, display the signature and close the graphics session. They are listed below in alphabetical order:

- BRESET
- CLOSEPGU
- OPENPGU
- PUT
- SETALPHA/GRAPH

For further details on the PGU functions, see the manual PGU, Graphics Management Package, Programmer Guide.

BRESET

This procedure disables buffer use and subsequently the PUT procedure is immediately executed.

COBOL Call

```
CALL "BRESET".
```

PASCAL+ Call

```
Breset;
```

CLOSEPGU

This procedure:

- closes the current PGU session
- clears the screen
- sets the work station as it was before the execution of the OPENPGU procedure
- unloads the PGU.

COBOL Call

```
CALL "CLOSEPGU".
```

PASCAL+ Call

```
ClosePGU;
```

This procedure initialises the PGU variables and the bitmap. The existence of font files is tested. If there is no font file this procedure returns a warning.

COBOL Call

```
CALL "OPENPGU" USING retcode, openPgu.
```

PASCAL+ Call

```
retcode := OpenPGU ( openPgu );
```

PARAMETER	MEANING
INPUT	
openPgu	The values which can be assigned to this parameter are one of the following: 0 the PGU procedures are used for alphanumeric I/O and both text and bitmap may be displayed together. 1 procedures other than those of the PGU are used for alphanumeric I/O and both text and bitmap are displayed together. 2 the PGU procedures are used for alphanumeric I/O. Text and bitmap are displayed separately. 3 procedures other than those of the PGU are used for alphanumeric I/O. Text and bitmap are displayed separately. See characteristic 1.
OUTPUT	
retcode	Reply code. See the following table.

Reply Codes

The following table gives a list of the reply codes returned by the procedure:

COBOL AND PASCAL+	MEANING
-2	There are no font files.
-1	The system is not graphic.
0	Operation executed correctly.
1	System error.
2	PGU not found.
3	PGU already open.
4	PGU can not be loaded.
5	PGU can not be started.
13	Not enough memory.

Characteristics

1. If the values 0 or 2 are chosen then "alphanumeric text" is automatically assumed.

   If the values 1 or 3 are chosen then the user must choose whether to use "alphanumeric text" or "graphics text". See the SETALPHA/GRAPH procedure.

2. If an integrated work station with colour, M30 with colour or monochromatic PC are used, the graphic bitmap and alphanumeric text are always displayed together.

3. If a monochromatic work station, monochromatic M30, M31 or PC with colour are used and if the "openPgu" parameter has been set to 2 or 3 then in order to display the graphic bitmap (or the alphanumeric text) it is necessary to set (or reset) the hide-alpha attribute (attribute 96) in all the character cells.

This procedure displays the signature previously stored in an array.

COBOL Call

```
CALL "PUT" USING retcode, putType, x1, y1, x2, y2, store, startIndex,
                 storeSize.
```

PASCAL+ Call

```
retcode := Put ( putType, x1, y1, x2, y2, store, startIndex );
```

PARAMETER	DESCRIPTION
INPUT	
putType	Specifies where the signature is to be displayed. If it is set to 0 the coordinates (x1, y1) and (x2, y2) are ignored and the signature is displayed from the upper left hand corner of the screen. If it is set to 1 only the coordinate (x1, y1) is considered and is the position of the upper left hand corner from where the signature is to be displayed. If it is set to 2 then both coordinates (x1, y1) and (x2, y2) are considered and are the positions of the upper left hand and lower right hand corners from where the signature is to be displayed.
x1	Coordinate along the x-axis of the upper left hand corner from where the signature is to be displayed.
y1	Coordinate along the y-axis of the upper left hand corner from where the signature is to be displayed.
x2	Coordinate along the x-axis of the lower right hand corner to where the signature is to be displayed.
y2	Coordinate along the y-axis of the lower right hand corner to where the signature is to be displayed.
store	Array of bytes containing the bitmap area to be displayed. See the characteristic 5.
startIndex	Ordinal value of the first byte of "store" to be displayed The ordinal value of the first byte of "store" is 0.

PARAMETER	DESCRIPTION
storeSize	Number of bytes to be displayed from the "store" array. See characteristic 4.
OUTPUT	
retcode	Reply code. See the following table.

Reply Codes

The following table gives a list of the reply codes returned by the procedure:

COBOL AND PASCAL+	MEANING
0	Operation executed correctly.
6	Illegal procedure call.
10	Parameter out of range.
11	Invalid screen position.

Characteristics

1. If the coordinates (x1, y1) and (x2, y2) are both considered then they must fall within the current window.

2. The coordinates x1, x2, y1, y2 can have the following values:

   x1, x2 — From 0 to 639 for graphics or alphanumeric integrated work station, M30, M31 and PC used as work stations.

   y1, y2 — From 0 to 399 for graphics integrated work station, M30, M31 and PC used as work stations.

   From 0 to 299 for alphanumeric integrated work station.

3. The point (0,0) is placed in the lower left corner of the screen.

4. The values of the expression (x2-x1) and (y2-y1) must be equal to the values of the output parameters x_pixel_Zoom (or x_pixel_Norm) and y_pixel_Zoom (or y_pixel_Norm) of the BITMAP procedure described below.

5. This parameter does not appear in the PASCAL+ interface insofar as it is obtained by subtracting "min" from "max", specified in the

definition of the "store" array.

6. The "store" array is loaded with the values of the "Norm_Bitm" array or with those of the "Zoom_Bitm" array (see BITMAP procedure), depending on whether the signature is to be displayed with or without zoom respectively.

7. Display of the signature may take place on graphic integrated work stations, M30, M31, PC used as work stations and alphanumeric integrated work stations with an MEG 3354 board.

SETALPHA/GRAPH

This procedure sets the PGU to the alphanumeric (SETALPHA) / graphic (SETGRAPH) state. If it already is in such a state the procedure call has no effect.

COBOL Call

```
CALL "SETALPHA".
CALL "SETGRAPH".
```

PASCAL+ Call

```
SetAlpha;
SetGraph;
```

Characteristics

1. After the execution of a graphics procedure the PGU is left in the graphics state. In such a state it is possible to write alphanumeric strings (the characters of the text are closer to each other than in the alphanumeric state). If text is desired in the alphanumeric state the PGU must be set to such a state via the SETALPHA procedure.

2. If the PGU alphanumeric I/O procedures are used and if the "openPgu" parameter of the OPENPGU procedure is set to 0 or 2 then it is not necessary to use the SETALPHA and SETGRAPH procedures.

COBOL INTERFACE FOR SIGNATURE VERIFICATION PROCEDURES CALL

Furthermore, the COBOL program must link the EXTINTF.LIB library in order to use the signature verification procedures. The linking is automatically done when using the RTLINK file, which contains the default directives for the linker.

Parameters Description

The following table gives the COBOL structure and description of the parameters used by the BITMAP and SCANNER procedures. The user must replace the "_" character with the "-" character because it is not accepted by the compiler. The parameters are listed in alphabetical order.

PARAMETER	DEFINITION	MEANING
Length_MHC	01 Length_MHC PIC S9(4) COMP.	Length of the generated MHC code.
line_id	01 line_id PIC S9(4) COMP.	Identifier of the channel to which the optical reader is connected.
MHC_Area	01 MHC_Area OCCURS 4098 TIMES PIC S9(2) COMP SYNC.	Area containing the signature MHC code.
Norm_Bitm	01 Norm_Bitm OCCURS 5126 TIMES PIC X.	Area containing the signature non zoomed bitmap.
reply	01 reply PIC S9(4) COMP.	Reply code.
screen_type	01 screen_type PIC A.	Letter which indicates the screen type.
x_mm	01 x_mm PIC S9(4) COMP.	Width in millimeters of the window to be read.
x_pixel_Norm	01 x_pixel_Norm PIC S9(4) COMP.	Pixel number per row of the non zoomed bitmap.
x_pixel_Zoom	01 x_pixel_Zoom PIC S9(4) COMP.	Pixel number per row of the zoomed bitmap.

(Cont.)

PARAMETER	DEFINITION	MEANING
x_pos	01 x_pos OCCURS 3 TIMES PIC 9.	Distance, in millimeters from the left side of the sheet, of the window to be read.
y_mm	01 y_mm PIC S9(4) COMP.	Height in millimeters of the window to be read.
y_pixel_Norm	01 y_pixel_Norm PIC S9(4) COMP.	Pixel row number of the non zoomed bitmap.
y_pixel_Zoom	01 y_pixel_Zoom PIC S9(4) COMP.	Pixel row number of the zoomed bitmap.
y_pos	01 y_pos OCCURS 3 TIMES PIC 9.	Distance, in millimeters from the upper side of the sheet, of the window to be read.
Zoom_Bitm	01 Zoom_Bitm OCCURS 10246 TIMES PIC X.	Area containing the signature zoomed bitmap.

Tab. 4-11 COBOL Parameters Description

PASCAL+ INTERFACE FOR SIGNATURE VERIFICATION PROCEDURES CALL

To access the two signature verification procedures, the PASCAL+ program must must link the interfaces:

visu.obj
tubo.obj
decomp.obj
clear.obj

Parameters Description

The following table gives the PASCAL+ structure and description of the parameters used by the signature verification procedures. The parameters are listed in alphabetical order.

PARAMETER	DEFINITION	MEANING
Length_MHC	Length_MHC : integer;	Length of the generated MHC code. Passed by VAR.
line_id	line_id : integer;	Identifier of the channel to which the optical reader is connected.Passed by VAR.

MHC_Area	byte=-128..127;
	dataitem=byte;
	MHC_Area:packed array [1..4098]
	of dataitem;
		Area containing the signature MHC code. Passed by VAR.

Norm_Bitm	byte=-128..127;
	dataitem=byte;
	Norm_Bitm:packed array [1..5126]
	of dataitem;
		Area containing the signature non zoomed bitmap. Passed by VAR.

reply	reply : integer;	Reply code.
screen_type	screen_type : char;	Letter which indicates the screen type. Passed by VAR.

(Cont.)

PARAMETER	DEFINITION	MEANING
x_mm	x_mm : integer;	Width in millimeters of the window to be read. Passed by VAR.
x_pixel_Norm	x_pixel_Norm : integer;	Pixel number per row of the non zoomed bitmap. Passed by VAR.
x_pixel_Zoom	x_pixel_Zoom : integer;	Pixel number per row of the zoomed bitmap. Passed by VAR.

x_pos	byte=-128..127;
	dataitem=byte;
	x_pos:packed array [1..3] of
	dataitem;
		Distance, in millimeters from the left side of the sheet, of the window to be read. Passed by VAR.

y_mm	y_mm : integer;	Height in millimeters of the window to be read. Passed by VAR.
y_pixel_Norm	y_pixel_Norm : integer;	Pixel row number of the non zoomed bitmap. Passed by VAR.
y_pixel_Zoom	y_pixel_Zoom : integer;	Pixel row number of the zoomed bitmap. Passed by VAR.

y_pos	byte=-128..127;
	dataitem=byte;
	y_pos:packed array [1..3]
	of dataitem;
		Distance, in millimeters from the upper side of the sheet, of the window to be read. Passed by VAR.

Zoom_Bitm	Byte=-128..127; dataitem=byte; Zoom_Bitm:packed array [1..10246] of dataitem;	Area containing the signature zoomed bitmap. Passed by VAR.

Tab. 4-12 PASCAL+ Parameters Description

SIGNATURE VERIFICATION PROCEDURES

This section describes the two procedures used for signature verification:

- BITMAP
- SCANNER

BITMAP

This procedure decompress the signature MHC code into two bitmaps (normal and zoomed format), thus making it possible to display the signature by means of the PGU "PUT" procedure.

Symbolic names have been assigned to the COBOL parameters and are in lower case. The user may replace these with more suitable names.

COBOL Call

```
CALL "decomp_mod_bit_map" USING MHC_Area, screen_type, Norm_Bitm,
                          Zoom_Bitm, x_pixel_Zoom, y_pixel_Zoom,
                          x_pixel_Norm, y_pixel_Norm, reply.
```

PASCAL+ Call

```
decomp_mod.bit_map (MHC_Area, screen_type, Norm_Bitm, Zoom_Bitm,
                    x_pixel_Zoom, y_pixel_Zoom, x_pixel_Norm,
                    y_pixel_Norm, reply);
```

PARAMETER	MEANING
INPUT	
MHC_Area	Area containing the signature MHC code.
screen_type	Parameter which indicates the screen type. The value "N" indicates a non graphics screen, otherwise it indicates a graphics screen. See characteristic 2.
OUTPUT	
Norm_Bitm	Area containing the signature non zoomed bitmap. See characteristic 1.
Zoom_Bitm	Area containing the signature zoomed bitmap. See characteristic 1.
x_pixel_Zoom	Pixel number per row of the zoomed bitmap.
y_pixel_Zoom	Pixel row number of the zoomed bitmap.
x_pixel_Norm	Pixel number per row of the non zoomed bitmap.
y_pixel_Norm	Pixel row number of the non zoomed bitmap.
reply	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL AND PASCAL+	MEANING
0	Operation executed correctly.
255	Decompressing error.

Characteristics

1. The first six bytes do not contain the signature code; four bytes contain the window size in which the signature will be displayed, and two bytes are reserved.

2. Display of the signature may take place on graphic integrated work stations, M30, M31, PC used as work stations and alphanumeric integrated work stations with an MEG 3354 board.

This procedure permits to acquire the signature in MHC code by an optical reader.

Symbolic names have been assigned to the COBOL parameters and they are in lower case. The user may replace them with more suitable names.

COBOL Call

```
CALL  "Scanner_mod_scanner" USING x_pos, y_pos, x_mm, y_mm,
                                  line_id, MHC_Area, Length_MHC, reply.
```

PASCAL+ Call

```
Scanner_mod.scanner (x_pos, y_pos, x_mm, y_mm, line_id, MHC_Area,
                     Length_MHC, reply);
```

PARAMETER	MEANING
INPUT	
x_pos	Distance, in millimeters from the left side of the sheet, of the window to be read.
y_pos	Distance, in millimeters from the upper side of the sheet, of the window to be read.
x_mm	Width, in millimeters, of the window to be read. The value must be in the range from 10 to 160 millimeters. See the characteristic.
y_mm	Height in millimeters of the window to be read. The value must be the range from 10 to 100 millimeters. See the characteristic.

PARAMETER	MEANING
line_id	Identifier of the channel to which the optical reader is connected.
OUTPUT	
MHC_Area	Area containing the signature MHC code.
Length_MHC	Length, in byte number, of the generated MHC code.
reply	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL AND PASCAL+	MEANING
0	Operation executed correctly.
1	Invalid channel identifier.
2	One or more invalid parameters.
4	Line busy.
8	Non allowed operation.
16	Line error (linedown, overrun, etc.)
32	Reading or writing break.
64	Time out.
100	Optical reader is off.
101	Non ready sheet.
102	Optical reader off-line
103	Transmission error.
104	Reading window too wide.
105	Reading window too high.
106	Reading window too big.
255	Non recognized MHC code.
1000+n	Line closing error with an error whose code is "n". The "n" value is in the range from 0 to 64.

Characteristic

The area of the window to be read must in every case be less than or equal to 50 sq.cm.

MESSAGE SWITCHING

Message switching in a distributed configuration is of particular importance. It is often necessary to switch messages (e.g. to synchronise operations) between applications that are active on different nodes in a distributed configuration.

MOS provides some PASCAL+ procedures to this end, which can also be called from COBOL, and with which synchronous and asynchronous messages can be exchanged between users in a distributed configuration.

LOCAL MESSAGE SWITCHING AGENCIES

The whole "Message Switching" service is based on a structure called the "Local Message Switching Agency (LMSA)"; there is one of these for each node in the distributed configuration, and at start-up each machine has complete visibility of its LMSA.

The LMSA consists of two files; the first contains information on each user at a node using "Message Switching", the second, whose default name is /IPL/MAILBOX, contains information on the messages in the queue at the node.

A decision can be taken either to create this file whenever the machine is started-up, or write in append mode on the previous one, using a parameter in the Grandpa configuration file. For more details as to the above operations and for the directives to insert in the Grandpa configuration file for Message Switching, see the manual System Software Generation and Installation, User Guide.

HANDLING NAMES AND MESSAGES

To enable a user to perform Message Switching, his logical name must be inserted in the corresponding LMSA, using the procedure "PUTNAME". This name must be 4 characters long and must be unique in the network. All user names must be reinserted at every system start up.

A user can be removed from the service by using the procedure "DELNAME".

The only check that "Message Switching" carries out on the user names is on the name of the sender and even if, for example, the name of the receiver does not exist in the service, the message sent will be ignored by each LMSA, but the procedures do not return any reply codes.

Immediate Messages

Immediate messages are those sent in asynchronous mode from one user to another, and are immediately displayed in the asynchronous window of the receiver's terminal.

In order to receive these messages, each user must supply the identifier of the asynchronous window of his work station, to the LMSA, by means of the "WSUP" procedure (returned by the "CONTX" procedure).

Enqueued Messages

Enqueued messages are those that are not displayed immediately on the receiver's terminal, but use the "mailbox".

Each user is not statically associated with a mailbox, (only one on each node) as this is only created when a message is received, and is present until there are no more messages to be read.

The messages sent by a user are put in the receiver's LMSA queue. An active selection process adds each message for a particular user to a chain of messages linked by pointers. The chain is started when the first message is received.

When a message has been read, the pointers of the message chain are updated, so the message is logically, but not physically, deleted from the mailbox.

MESSAGE SWITCHING PROCEDURES

The procedures available for "Message Switching" are the following:

- BROADCAST: sends an enqueued message to all users.
- BROADDISPLAY: sends an immediate message to all users.
- CLEARMESSAGE: deletes all the messages of a user.
- CONTX: returns the identifier of the asynchronous window of the work station.
- DELNAME: deletes a user from the service.
- PUTNAME: adds a user to the service.
- READMESSAGE: reads a message.
- READTIMEOUT: waits for a message for a specified time.
- READWAITING: waits for a message.
- SENDDISPLAY: sends an immediate message to a specific user.
- SENDMESSAGE: sends an enqueued message to a specific user.
- TESTMESSAGE: shows the state of a user's mailbox.
- WSDOWN: deletes the identifier of the asynchronous window of the work station, from the LMSA.
- WSUP: supplies the LMSA with the identifier of the asynchronous window of the work station.

Reply Codes

The following table gives a list of the reply codes returned by the Message Switching procedures.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	Invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.3	TH_USER TH_MSGTOOLONG	Message too long.
3.11	TH_USER TH_NOMSG	No messages.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

Tab. 4-13 Message Switching Procedures Reply Codes

COBOL INTERFACE FOR MESSAGE SWITCHING PROCEDURES CALL

A COBOL program must be linked with the EXTINTF.LIB library in order to use the Message Switching procedures. The linking is automatically done when using the RTLINK file which contains the default directives for the linker.

Parameters Description

The following table gives the COBOL structure and description of the parameters used by the Message Switching procedures. The parameters are listed in alphabetical order.

PARAMETER	DEFINITION	MEANING
MSGAREA	01 MSGAREA PIC X(4000).	Message text.
MSGLEN	01 MSGLEN PIC 9(4) COMP.	Message length.
MSGNUM	01 MSGNUM PIC 9(4) COMP.	Number of messages still to be read in the mailbox.
OPT	01 OPT PIC 9(4) COMP.	Not used.
OTHERONE	01 OTHERONE PIC X(4).	Name of a user.
RETCODE	01 RETCODE. 02 CLASS PIC 99 COMP. 02 SUBCLASS PIC 99 COMP.	Reply code.
RTIMEOUT	01 RTIMEOUT PIC 9(4) COMP.	Waiting time.
TIMEOUT	01 TIMEOUT PIC 9(9) COMP.	Not used.
WHOAMI	01 WHOAMI PIC X(4).	Name of a user.
WSID	01 WSID PIC X(8).	Identifier of the asynchronous window.
WSNAME	01 WSNAME PIC X(4).	Name of a user.

Tab. 4-14 COBOL Parameters Description

PASCAL+ INTERFACE FOR MESSAGE SWITCHING PROCEDURES CALL

To access the Message Switching procedures, the PASCAL+ program must link the interface

MSW_ui.obj

and must include the following modules:

- systypes.i
- msw.i

for the definition of the types and procedures.

Parameters Description

The following table gives the PASCAL+ structure and description of the parameters used by the Message Switching procedures. The parameters are listed in alphabetical order.

PARAMETER	DEFINITION	MEANING
MsgArea	data = packed array [1..4000] of char; MsgArea : data;	Message text. Passed by VAR.
MsgLen	MsgLen : integer;	Message length. Passed by VAR.
MsgNum	MsgNum : integer;	Number of messages still to be read in the mailbox. Passed by VAR.
Opt	Opt : integer;	Not used.
Otherone	T_key = packed array [1..4] of char; Otherone : T_key;	Name of a user. Passed by VAR.
RetCode	RetCode : T_threply; (* defined in msw.i *)	Reply code. Passed by VAR.
RTimeout	RTimeout : integer;	Waiting time
Timeout	Timeout : longinteger;	Not used.
Whoami	T_key = packed array [1..4] of char; Whoami : T_key;	Name of a user. Passed by VAR.
WsId	WsId : T_systemid; (* defined in systypes.i *)	Identifier of the asynchronous window. Passed by VAR.
WsName	T_key = packed array [1..4] of char; WsName : T_key;	Name of a user. Passed by VAR.

Tab. 4-15 PASCAL+ Parameters Description

BROADCAST

This procedure sends an enqueued message to all users.

COBOL Call

```
CALL "MSG_FUNCT_BROADCAST" USING WHOAMI, MSGAREA, MSGLEN, TIMEOUT,
                                 RETCODE.
```

PASCAL+ Call

```
MSG_FUNCT.BroadCast (Whoami, MsgArea, MsgLen, Timeout, RetCode);
```

PARAMETER	MEANING
INPUT	
Whoami	Message sender.
MsgArea	Message text, 4000 characters maximum length.
MsgLen	Message length. It must be an integer <= 4000.
Timeout	Not used at present.
OUTPUT	
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.3	TH_USER TH_MSGTOOLONG	Message too long.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

BROADDISPLAY

This procedure sends an immediate message to all users.

COBOL Call

```
CALL "MSG_FUNCT_BROADDISPLAY" USING WHOAMI, MSGAREA, MSGLEN, TIMEOUT,
                                    RETCODE.
```

PASCAL+ Call

```
MSG_FUNCT.BroadDisplay (Whoami, MsgArea, MsgLen, Timeout, RetCode);
```

PARAMETER	MEANING
INPUT	
Whoami	Message sender.
MsgArea	Message text, 64 characters maximum length.
MsgLen	Message length. It must be an integer <= 64.
Timeout	Not used at present.
OUTPUT	
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.3	TH_USER TH_MSGTOOLONG	Message too long.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

CLEARMESSAGE

This procedure deletes all the messages of a user contained in the mailbox.

COBOL Call

```
CALL "MSG_FUNCT_CLEARMESSAGE" USING WHOAMI, RETCODE.
```

PASCAL+ Call

```
MSG_FUNCT.ClearMessage (Whoami, RetCode);
```

PARAMETER	MEANING
INPUT	
Whoami	User who requires the function.
OUTPUT	
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

This procedure returns the identifier of the asynchronous window of the work station.

COBOL Call

```
CALL "MSG_UT_CONTX" USING WSID, RETCODE.
```

PASCAL+ Call

```
MSG_UT.ConTx (WsId, RetCode);
```

PARAMETER	MEANING
OUTPUT	
WsId	Identifier of the asynchronous window.
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.

Characteristic

This procedure must be called to receive immediate messages because the identifier returned must be passed to LMSA by means of the "WSUP" procedure.

This procedure deletes a user from the service.

COBOL Call

```
CALL "MSG_NAME_DELNAME" USING WSNAME, RETCODE.
```

PASCAL+ Call

```
MSG_NAME.DelName (WsName, RetCode);
```

PARAMETER	MEANING
INPUT	
WsName	User to be deleted.
OUTPUT	
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.

PUTNAME

This procedure adds a user to the service.

COBOL Call

```
CALL "MSG_NAME_PUTNAME" USING WSNAME, OPT, RETCODE.
```

PASCAL+ Call

```
MSG_NAME.PutName (WsName, Opt, RetCode);
```

PARAMETER	MEANING
INPUT	
WsName	User to be added.
Opt	Not used at present.
OUTPUT	
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.

This procedure reads an enqueued message sent from a specified user, or from any user.

COBOL Call

```
CALL "MSG_FUNCT_READMESSAGE" USING WHOAMI, OTHERONE, MSGNUM, MSGAREA,
                                   MSGLEN, RETCODE.
```

PASCAL+ Call

```
MSG_FUNCT.ReadMessage (Whoami, Otherone, MsgNum, MsgArea, MsgLen,
                       RetCode);
```

PARAMETER	MEANING
INPUT	
Whoami	User who requires the function.
Otherone	Message sender. See the characteristic.
OUTPUT	
Otherone	Message sender.
MsgNum	Messages number still to be read in the mailbox.
MsgArea	Message text.
MsgLen	Message length.
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.11	TH_USER TH_NOMSG	No messages.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

Characteristic

The "Otherone" parameter is to be filled with four blanks if the user wants to read a message sent from any user.

This procedure waits, for a specified time, for a message sent from a specified user, or from any user.

COBOL Call

```
CALL "MSG_FUNCT_READTIMEOUT" USING WHOAMI, OTHERONE, MSGNUM, RTIMEOUT
                                   MSGAREA,MSGLEN, RETCODE.
```

PASCAL+ Call

```
MSG_FUNCT.ReadTimeout (Whoami, Otherone, MsgNum, RTimeout, MsgArea,
                       MsgLen, RetCode);
```

PARAMETER	MEANING
INPUT	
Whoami	User who requires the function.
Otherone	Message sender. See the characteristic.
RTimeout	Time for waiting a message.
OUTPUT	
Otherone	Message sender.
MsgNum	Messages number still to be read in the mailbox.
MsgArea	Message text.
MsgLen	Message length.
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.11	TH_USER TH_NOMSG	No messages.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

Characteristic

The "Otherone" parameter is to be filled with four blanks if the user wants to read a message sent from any user.

This procedure waits for a message sent from a specified user, or from any user.

COBOL Call

```
CALL "MSG_FUNCT_READWAITING" USING WHOAMI, OTHERONE, MSGNUM, MSGAREA,
                                   MSGLEN, RETCODE.
```

PASCAL+ Call

```
MSG_FUNCT.ReadWaiting (Whoami, Otherone, MsgNum, MsgArea, MsgLen,
                       RetCode);
```

PARAMETER	MEANING
INPUT	
Whoami	User who requires the function.
Otherone	Message sender. See the characteristic.
OUTPUT	
Otherone	Message sender.
MsgNum	Messages number still to be read in the mailbox.
MsgArea	Message text.
MsgLen	Message length.
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.11	TH_USER TH_NOMSG	No messages.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

Characteristic

The "Otherone" parameter is to be filled with four blanks if the user wants to read a message sent from any user.

This procedure sends an immediate message to a specified user.

COBOL Call

```
CALL "MSG_FUNCT_SENDDISPLAY" USING WHOAMI, OTHERONE, MSGAREA, MSGLEN,
                                   TIMEOUT, RETCODE.
```

PASCAL+ Call

```
MSG_FUNCT.SendDisplay (Whoami, Otherone, MsgArea, MsgLen, Timeout,
                       RetCode);
```

PARAMETER	MEANING
INPUT	
Whoami	Message sender.
Otherone	Message receiver.
MsgArea	Message text, 64 characters maximum length.
MsgLen	Message length. It must be an integer <= 64.
Timeout	Not used at present.
OUTPUT	
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.3	TH_USER TH_MSGTOOLONG	Message too long.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

This procedure sends an enqueued message to a specified user.

COBOL Call

```
CALL "MSG_FUNCT_SENDMESSAGE" USING WHOAMI, OTHERONE, MSGAREA, MSGLEN,
                                   TIMEOUT, RETCODE.
```

PASCAL+ Call

```
MSG_FUNCT.SendMessage (Whoami, Otherone, MsgArea, MsgLen, Timeout,
                       RetCode);
```

PARAMETER	MEANING
INPUT	
Whoami	Message sender.
Otherone	Message receiver.
MsgArea	Message text, 4000 characters maximum length.
MsgLen	Message length. It must be an integer <= 4000.
Timeout	Not used at present.
OUTPUT	
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.3	TH_USER TH_MSGTOOLONG	Message too long.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

This procedure shows the state of a user's mailbox.

COBOL Call

```
CALL "MSG_FUNCT_TESTMESSAGE" USING WHOAMI, MSGNUM, RETCODE.
```

PASCAL+ Call

```
MSG_FUNCT.TestMessage (Whoami, MsgNum, RetCode);
```

PARAMETER	MEANING
INPUT	
Whoami	User who requires the function.
OUTPUT	
MsgNum	Messages number still to be read in the mailbox.
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

WSDOWN

This procedure deletes from LMSA the identifier of the asynchronous window of the work station.

COBOL Call

```
CALL "MSG_WS_WSDOWN" USING WSNAME, RETCODE.
```

PASCAL+ Call

```
MSG_WS.WsDown (WsName, RetCode);
```

PARAMETER	MEANING
INPUT	
WsName	User who requires the function.
OUTPUT	
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

This procedure supplies the LMSA with the identifier of the asynchronous window of the work station.

COBOL Call

```
CALL "MSG_WS_WSUP" USING WSNAME, WSID, RETCODE.
```

PASCAL+ Call

```
MSG_WS.WsUp (WsName, WsId, RetCode);
```

PARAMETER	MEANING
INPUT	
WsName	User who requires the function.
WsId	Identifier of the asynchronous window of the work station.
OUTPUT	
RetCode	Reply code.

Reply Codes

The following table gives a list of the reply codes returned by the procedure.

COBOL	PASCAL+	MEANING
0.0	TH_CORRECT TH_OKAY	Operation executed correctly.
1.0	TH_SYSTEM TH_SYSTEMERROR	System error.
2.1	TH_SUBSYSTEM TH_TIMEOUT	Time out.
3.1	TH_USER TH_INVALIDPARAMETERS	One or more invalid parameters.
3.2	TH_USER TH_INVALIDCOMMAND	Invalid command.
3.13	TH_USER TH_NAMENOTFOUND	Unknown user.

Characteristics

1. This procedure must be called to receive immediate messages.
2. The identifier of the asynchronous window of the work station is returned by means of the "CONTX" procedure.
3. This procedure cannot be called from an application program in the background.

EXECUTION OF MCL ACTIVITIES FROM SHELL OR GRANDPA

MOS provides a function which can call an MCL activity (executable programs, MCL procedures,..) to be called and executed by a program activated via Shell or Grandpa, returning control to the calling program when the activity ends.

PASCAL+ INTERFACE

In order to access the EXEC function the program must include the modules:

```
exec.d
systypes.f
```

and link the interface:

```
exec.obj
```

Definition of Types

```
T_reply       = record
                  case class : T_class of
                  CORRECT    : (correctCode : T_correctCode);
                  SYSTEM     : (sysErrCode  : T_sysErrCode );
                  DEBUG      : (dbgErrCode  : T_dbgErrCode );
                  PMMWARNING : (pmmWarnCode : T_pmmWarnCode);
                  PMMERROR   : (pmmErrCode  : T_pmmErrCode );
                  FSWARNING  : (fsWarnCode  : T_fsWarnCode );
                  FSERROR    : (fsErrCode   : T_fsErrCode  );
                  WSWARNING  : (wsWarnCode  : T_wsWarnCode );
                  WSERROR    : (wsErrCode   : T_wsErrCode  );
                  RUNTIME    : (rtCode      : T_rtCode     );
                  USRCLASS   : (usrCode     : T_usrCode    );
                  LMWARNING  : (lmWarnCode  : T_lmWarnCode );
                  LMERROR    : (lmErrCode   : T_lmErrCode  );
                  THWARNING  : (thWarnCode  : T_thWarnCode );
                  THERROR    : (thErrCode   : T_thErrCode  );
                  FRWARNING  : (frWarnCode  : T_frWarnCode );
                  FRERROR    : (frErrCode   : T_frErrCode  );
                  end;

T_class       = (CORRECT, SYSTEM, DEBUG, PMMWARNING, PMMERROR,
                 FSWARNING, FSERROR, WSWARNING, WSERROR,
                 RUNTIME, USRCLASS, LMWARNING, LMERROR,
                 THWARNING, THERROR, FRWARNING, FRERROR);

T_correctCode = (OKAY);

T_sysErrCode  = (TERMINALDOWN, DIAGNOSTICERROR, RECEIVEERROR,
                 HARDWAREERROR, SYSTEMERROR, TABLEOVERFLOW,
                 OUTOFDISKSPACE, FATALERROR, ROFAULT, KERNSEGFAULT,
                 SEGLENFAULT, INHIBSEGFAULT, XQTFAULT, INVALIDINST,
                 PRIVILEGEDINST, FATALSEGFAULT, NOPROCEDURE,
                 NOROUTING, RESULTSDOUBTFUL, OPERATIONABORTED);
      .
      .
      .
```

The types are defined in the systypes.i module.

This function executes MCL activities from a program activated via Shell or Grandpa.

Function Definition

```
function exec (var cmd      :  packed array [lb1..ub1 : integer] of char;
               var usrtype  :  integer;
               var result   :  packed array [lb2..ub2 : integer] of char;
               start        :  integer;
               var count    :  integer) : T_reply;
```

PARAMETER	DESCRIPTION
INPUT	
cmd	This array specifies the character-string defining the required activity. The string may be 160 characters long (max.). Each MCL procedure or executable program must be followed by /LF/.
OUTPUT	
usrtype	Reply code for the MCL activity. See the characteristic.
result	This array specifies the character-string contained in the Shell variable %STATUS.
start	Ordinal number of the character at which the the calculation as to the length of the string contained in the Shell variable %STATUS is started.
count	Length of the string of characters contained in the Shell variable %STATUS.

Reply Codes

The function returns the code 0 is the MCL activity can be executed, or a different value if errors have occurred in the request for execution of the activity (e.g. if the name of an executable program does not exist).

Characteristics

The reply code returned by the MCL activity can have one of the following values:

0 : the MCL activity has been executed correctly

1 : warning

2 : one or more errors have occurred either inside or outside the MCL activity during execution.

127 : the activity has been executed correctly and there is further information in the result.

SUSPENSION OF THE CONNECTION ON A SWITCHED LINE

MOS provides a function which suspends the connection between a remote work station and the L1 MOS system to save telephone expenses for use of the switched line.

COBOL INTERFACE

A COBOL program must be linked with the EXTINTF.LIB library in order to use the LOGOFF function. The linking is automatically done when using the RTLINK file which contains the default directives for the linker. Therefore the variable "WAITFORON" must be defined in the application program as follows:

```
01  WAITFORON  PIC S9(9) COMP.
```

PASCAL+ INTERFACE

The program must include the following modules in order to access the LOGOFF function:

```
systypes.i
term_p.i
term_t.i
```

and must link to the interface

```
WSM_ui.obj
```

Definition of Types

```
T_reply       = record
                  case class : T_class of
                  CORRECT    : (correctCode : T_correctCode);
                  SYSTEM     : (sysErrCode  : T_sysErrCode );
                  DEBUG      : (dbgErrCode  : T_dbgErrCode );
                  PMMWARNING : (pmmWarnCode : T_pmmWarnCode);
                  PMMERROR   : (pmmErrCode  : T_pmmErrCode );
                  FSWARNING  : (fsWarnCode  : T_fsWarnCode );
                  FSERROR    : (fsErrCode   : T_fsErrCode  );
                  WSWARNING  : (wsWarnCode  : T_wsWarnCode );
                  WSERROR    : (wsErrCode   : T_wsErrCode  );
                  RUNTIME    : (rtCode      : T_rtCode     );
                  USRCLASS   : (usrCode     : T_usrCode    );
                  LMWARNING  : (lmWarnCode  : T_lmWarnCode );
                  LMERROR    : (lmErrCode   : T_lmErrCode  );
                  THWARNING  : (thWarnCode  : T_thWarnCode );
                  THERROR    : (thErrCode   : T_thErrCode  );
                  FRWARNING  : (frWarnCode  : T_frWarnCode );
                  FRERROR    : (frErrCode   : T_frErrCode  );
                  end;

T_class       = (CORRECT, SYSTEM, DEBUG, PMMWARNING, PMMERROR,
                 FSWARNING, FSERROR, WSWARNING, WSERROR,
                 RUNTIME, USRCLASS, LMWARNING, LMERROR,
                 THWARNING, THERROR, FRWARNING, FRERROR);

T_correctCode = (OKAY);

T_sysErrCode  = (TERMINALDOWN, DIAGNOSTICERROR, RECEIVEERROR,
                 HARDWAREERROR, SYSTEMERROR, TABLEOVERFLOW,
                 OUTOFDISKSPACE, FATALERROR, ROFAULT, KERNSEGFAULT,
                 SEGLENFAULT, INHIBSEGFAULT, XQTFAULT, INVALIDINST,
                 PRIVILEGEDINST, FATALSEGFAULT, NOPROCEDURE,
                 NOROUTING, RESULTSDOUBTFUL, OPERATIONABORTED);
      .
      .
      .
```

The types are defined in the systypes.i module.

This function suspends the connection on a switched line between a remote work station and the L1 MOS system, so as to save telephone expenses on the line.

Definition of the Function

```
function Logoff (wsId        : T_systemId;
                 waitForOn   : longinteger) : T_reply;
```

COBOL Call

```
CALL "WSR_CLOGOFF"  USING WAITFORON.
```

PARAMETER	DESCRIPTION
INPUT	
wsId	Identifier of the work station which requests suspension of the line connection. See the characteristics.
waitForOn	Time during which the connection is to be suspended, expressed as units of 100 miliseconds.

Reply Codes

The following Table gives a list of the reply codes returned by the function.

CODE	MEANING
INVALIDID	Invalid work station identifier.
UNITNOTAVAILABLE	Work station not configured.

Characteristics

1. See the manual PMM and Driver Primitives, Reference Manual at the section "Work station Identification" in Chapter 3 for the work station identifier.
2. This function cannot be used if an Error Controller is present.

5. FILE SIZE

This chapter gives information on calculating the size of the files present on disk, in order that they can be handled in the best possible way. It also gives indications on handling disk space.

CALCULATING FILE OCCUPATION

This section lists the size of the files handled by MOS, as well as some other structures used by the File System Management, so that calculating the number of bytes occupied is made easier.

This information can be useful for establishing the most convenient size of the "Allocation Unit" parameter when a file is created, for calculating the number of bytes occupied by the auxiliary structures of some types of files or for calculating how much space on disk is available at a particular moment while the system is running (knowing the type and number of files, etc.).

Byte-stream File

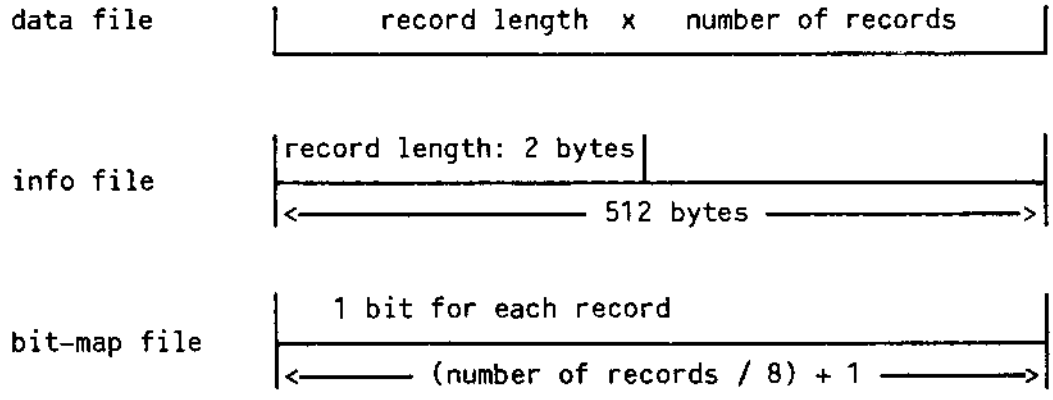
The number of bytes occupied by a byte-stream file is equal to the value obtained from the equation:

(position of the file's last recorded byte) - (position of the first byte) + 1

Positional File

A positional file physically consists of two files (one, known as "data", contains the data and the other, known as "info", contains the necessary information for accessing the desired record) if created with the 'no-deletion' option, or three files (as well as the two mentioned above, there is a third, the "bit-map" file, which is used by the File System Management to check the validity of each record in the file) if created with the 'deletion' option.

The number of bytes occupied by a positional file is therefore equal to the number of bytes occupied by all the files making it up, and its occupation is the following:



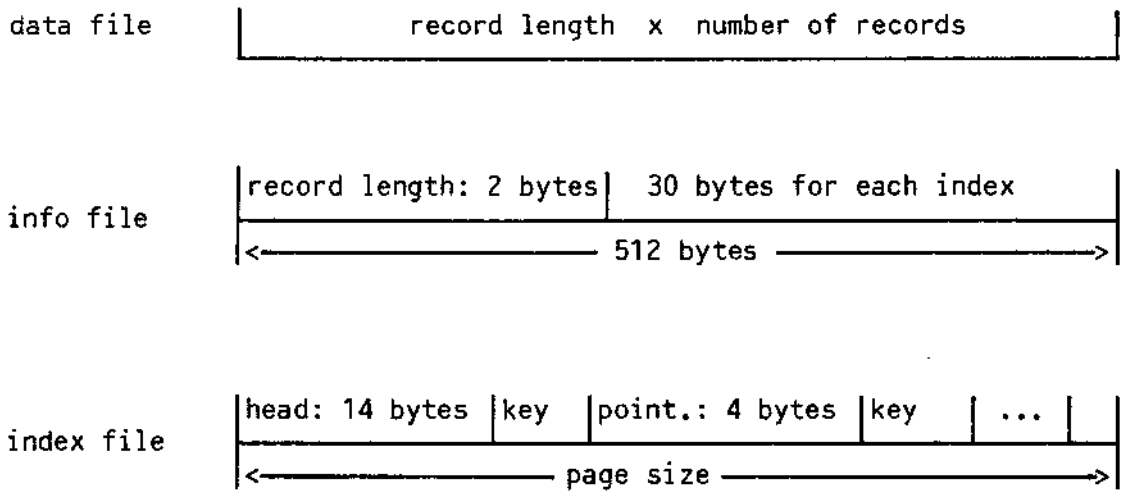
Note: The number of records means the number from the first to the last, inclusive, of the recorded records.

Keyed File

A keyed file physically consists of three files:

- data file: contains the data
- info file: contains information about the indices allowing access to the file
- index file: contains the pages with the indices allowing access to the file

These three files occupy the following bytes:



Note: The number of records means the number from the first to the last, inclusive, of the valid records.

Remarks: The following theoretical calculations represent a "worst case" approximation for record insertion in a data file.

The number of bytes occupied by the index file depends on the selections made when the keyed file is created. One of these is the selection of the page size: it can be 512, 1024, 2048 or 4096 bytes.
The number of elements in a page is:

$$NE = \frac{\text{Page size} - 14}{\text{Key length} + 4}$$

The number of pages of the lower level of the tree would be the integer part of the result of:

$$NP = \frac{\text{Record number} - 1}{NE} + 1$$

The number of pages used by the index file depends on the criteria used for dividing the pages, which has been decided by the user, and can be divisions of 50-50, 90-10 or 100-0 (see the section below "Supplementary Notes on the Key Indices").
The number of pages is:

NP = NP x 2 (with criterium 50-50)

NP = NP x 1.2 (with criterium 90-10)

NP = NP x 1 (with criterium 100-0)

If NP = 1, the number of occupied bytes is NB = Page Size.

If NP > 1, the tree structure in which the indices are stored consists of several levels.

In this case, each level of the tree requires a new page whenever the NP/NE ratio of the lower level is > 1.

It must be remembered that if there is more than one page at one level, there must be a page at a higher level for addressing the lower level pages.

When the total number of pages in the various tree levels has been obtained, the number of bytes occupied by the index file is:

NB = NP x Page Size

The total number of bytes occupied by the keyed files is obtained from the sum of the occupation of the files which it consists of (data, info, index).

Volume Root Directory

512 bytes are initially assigned to this directory.

Each new name which is subsequently created (for example, a file, a directory, a volume) occupies 18 bytes.

Volume Bit Map

This file's size is equal to 1 bit for each block (of 512 bytes) assigned to the volume.

The number of blocks is equal to the size assigned by the user when the volume is created, divided by 512.

The number of bytes occupied by the Bit Map file is therefore:

$$NB = \frac{\text{number of blocks}}{8}$$

Pdd Table

The number of bytes occupied by this table is equal to the number of files (assigned by the user to the volume) multiplied by 156.

Note: When a volume is created, the maximum number of files which can be created in it should be carefully established, so that the Pdd table is not filled when there is still free space in the volume.

Volume Descriptor

This always occupies 512 bytes.

USER VISIBILITY OF FILE OCCUPATION

Information relating to the size of some file system objects can be displayed, using the PRY command in Shell environment.

With regard to the syntax to be used when calling the command, and the meaning of the information displayed, see the manual MOS, SHELL Commands, Reference Manual.

Supplementary details are given below on two items of information which can be obtained from this command: SIZE and BUSY SPACE. Both refer to the size of the file given as the object of the command, but their meaning depends on the type of file in consideration.

Byte-stream File

The SIZE information referring to a byte-stream file indicates the number of bytes between the first and the last written in the file. In this case BUSY SPACE has the same number of bytes.

Positional 'no-deletion' File

SIZE indicates the number of records from the first to the last, inclusive, of the written records. BUSY SPACE has the same number. This is because, for this type of file, enough space is allocated for all the valid records from the first to the last, even if not all have been written.

Positional 'deletion' File

SIZE indicates the number of records from the first to the last, inclusive, of the valid records. BUSY SPACE, however, indicates the number of valid records between the first and the last. The difference between the first and second value represents the number of records deleted or never created between the first and last valid records.

Keyed File

SIZE indicates the number of recorded keys from the first to the last, inclusive, of the valid keys. BUSY SPACE, however, indicates the number of valid keys between the first and the last.

Observations

The information obtained via the PRY command refers only to the "data" file. For files with auxiliary structures (info, bit-map and index files) occupation must be calculated according to the details given in the section "Calculating File Occupation".

SUPPLEMENTARY NOTES ON THE KEY INDICES

All the indices relating to a keyed file are organized in a "tree" structure. This tree is implemented with a byte file which is logically divided into pages by the File System Management.

Sizing the Pages

These pages are organized according to the tree structure. The size of these pages, defined by the user, can be 512, 1024, 2048 or 4096 bytes.

Large pages guarantee less overall occupation and division of the tree but slow down searches for an element in them, whilst smaller pages make searches within them easier but the tree structure is more difficult to scan.

A page contains a heading which gives information about the page and its family: position of the pages of the same level to the left and right, position of the 'father' page (the page on the level immediately above), number of bytes used in the page in question.

The rest of the page contains the "key-elements". Each of these is made up of a pointer and a key. The length of the key is constant for a particular index. If the page is a 'leaf' of the tree (that is, it belongs to the lowest level), the pointer indicates the position of the record of the data file associated to the key.

All the keys in an index are always present in the leaves, and some can also be present in higher level nodes.

The last "key-element" of the right hand leaf is a dummy-element, and only indicates that there are no other keys. The value of its key is always greater than any key which can be inserted by the user.

If the page is not a 'leaf' of the tree, the key element pointer indicates the position of the 'son' page in the tree.

The 'son' page will always contain "key-elements" whose keys will have a value less than or equal to that of the key associated to the pointer of the 'son' page in the 'father' page.

The last "key-element" of a 'son' page always contains the same key as the one pointing to the 'son' in the 'father' page.

Split Strategy

When a new key is to be inserted, the tree is scanned to identify the exact point at which the key must be placed.

This search always ends in a 'leaf'. The new key is inserted and all the larger ones move to the right. If this movement causes the page to overflow, it is divided into two leaves of equal level and the element which divides them is reported to the 'father' page.

The 'father' page normally has enough free space and so a further division is not necessary. If, however, the opposite is true the dividing process is repeated for the 'father' page.

The splitting takes place according to a criterium defined by the user, and which can be 50-50, 90-10 or 100-0.

This criterium indicates the percentage of the contents of the full page which is to be transferred to the two new pages just created.

For example, the 90-10 criterium means that one of the two new pages created will contain 90% of the elements of the old page (new left hand page) and the other 10% of the elements (new right hand page). This means determining how much free space will be available in the index.

If, for example, the keys are generated sequentially and are consecutive integers, the 100-0 division criterium will ensure maximum packing, without leaving unused space.

If, however, the keys are created in random mode it is advisable to use the 50-50 criterium to limit the number of page splitting at run-time.

Finally, if a file is generated in controlled conditions, and therefore not subjected to frequent insertions, the most suitable solution is the 90-10 criterium. The file would be created sequentially, and as each leaf would have 10% free space, the probability of causing page splitting at run-time with random insertions is rather low.

Any subsequent insertion of a new key is carried out in the same way.

The following figure shows a set of indices stored in a tree structure where the size of the pages is such that each of them can contain a maximum of three keys.

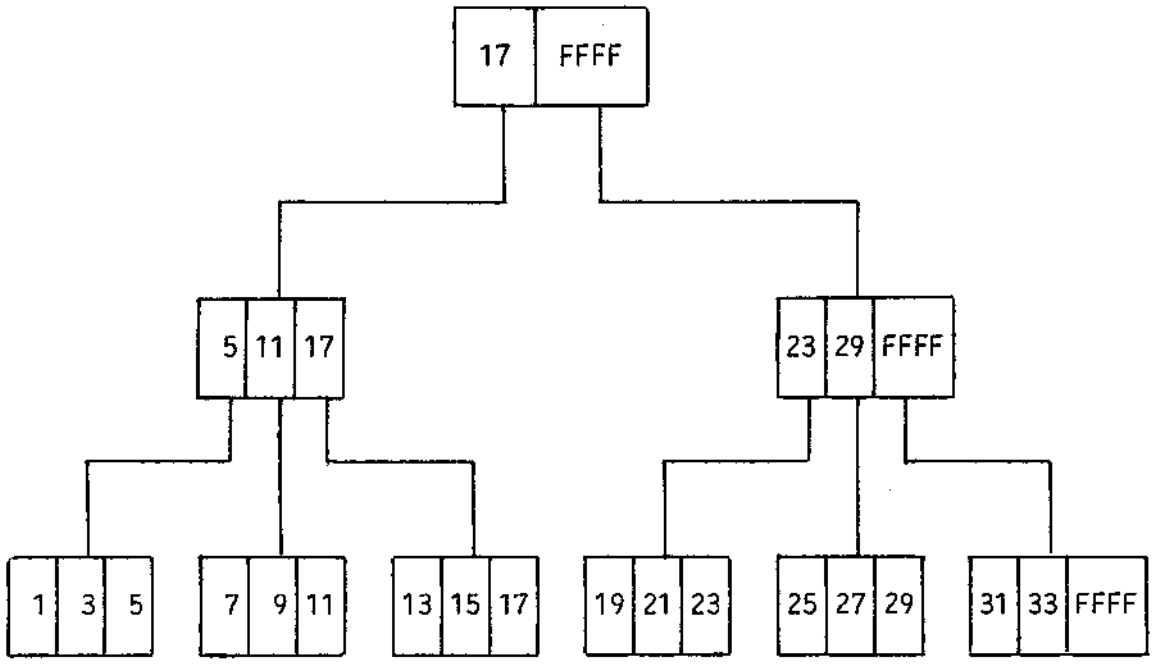


Fig. 5-1 Indices Stored in a Tree Structure

DISK SPACE ALLOCATION STRATEGY

The disk space allocation strategy followed by the File System Management is described below.

This strategy resolves the need to allocate contiguous space to the file with that of creating dynamic file extents.

When the file is created it is allocated a contiguous space on disk. The desired size of this space depends on the allocation unit specified by the user.

When calculating this size the information given in the earlier section, "Calculating File Occupation", might be of help, if it is possible to forecast the number and size of the records which will be handled.

When there is not enough contiguous space available, the File System Management allocates a space as near as possible to the size requested and informs the user of this.

The established size of the file is not, however, definite: if so much new data is added to the file that it overflows, an extent is allocated to the file which is the same, or as near as possible, to the allocation unit. This new extent is probably not contiguous to the original extent.

After a while, a file may become rather fragmentary, with multiple separated disk extents. Such fragmentation means access time to the file is increased.

On the other hand, the allocated space is "dedicated" to the file, in the sense that it can only be occupied by that file (if the file grows) but not by other files. If the size is too big for the "Allocation Unit" parameter, space is allocated on disk which is not occupied.

A good handling of the allocation unit can eliminate these inconveniences, and therefore it is necessary:

- to calculate the probable final size of the file and specify this value when the file is created (initial allocation unit)

- subsequently substitute the original attribute with a reduced value (expansion allocation unit), using the Shell command SETINFO.

If the initial allocated space is insufficient, the file extent will be incremented with this reduced value, thus avoiding creating unnecessary space within the file.

It is also necessary to periodically pack the fragmented files (COMPACT command) in order to group the various extents in a single contiguous space.

Finally, the system administrator can pack the various extents of the file system objects, created in a volume, into contiguous blocks (VCOMPACT command).

There are two commands (PACK and UNPACK) which permit file system objects to be compacted or expanded on floppy disk.

6. SCREEN HANDLING

THE SCREENS

The L1 line systems can be used with various types of screens of differing physical sizes:

- 15"
- 14"
- 9"
- 5"

The 9" screen has a controller which allows it to be used in different formats (the number of characters which can be displayed).

THE FORMATS

The format determines the size of the characters displayed on the screen, according to their physical size. The 9" screen allows the format type to be changed dynamically at the start of an activity, at Grandpa level for activities started automatically at system initialization (this is limited to the master terminal screen) or in Shell environment (SETTERM command). A default format is defined at system initialization time for this screen type, which is used until the user modifies it. This format has 2000 characters.

The formats which can be adopted are:

- 25 rows by 80 columns (2000 characters), for 15", 14" and 9" trivalent screens
- 25 rows by 40 columns (1000 characters), for 9" trivalent screens
- 13 rows by 40 columns (520 characters), for 9" trivalent and 5" screens.

When the screen changes from one format to another the size of the displayed characters changes. By halving the number of columns the characters and spacing become wider. By halving the number of rows the characters become higher.

MOS SUBSYSTEMS AND SCREEN FORMATS

The various MOS subsystems allow all types of screens, which can be connected to the L1 systems, to be handled. Handling the various formats, however, is subject to restrictions imposed by the specific requirements of some of these subsystems. These restrictions are listed below for each of the software environments or the components imposing them.

Shell

The Shell environment can be used with any type of screen (5", 9", 14", 15") and any selected format.

If screen splitting has been requested, the system line is handled correctly using 15", 14" or 9" screens. In all other cases, messages to be displayed on the system line appear sequentially, and if they are longer than 40 characters only the last 40 remain displayed.

If screen splitting has not been requested, the system line functions are not available: the SUSPEND and RESUME commands cannot be called and the messages which normally appear on the system line are not displayed. The KILL command is instead available by entering /CTRL/ /K/ .

The messages displayed by the Shell commands in the upper window of the screen are handled correctly using 15", 14" or 9" screens. In all other cases, messages can be displayed on several lines.

BEAM

The BEAM environment can be used with any type of screen (5", 9", 14", 15") and any selected format.

Editor

The system Editor can be used with any type of screen and any selected format. However, only 24 rows are used at the most.

DMS

This package can only be used with the 25 x 80 format or, if creating the Data Dictionaries, with the 13 x 40 format.

COBOL

COBOL allows program preparation with Screen Section for any format. During the compilation phase the size defined by the Screen Section is checked to make sure it does not exceed 24 rows by 80 characters.

A COBOL program can be executed with any type of screen and format. The language's run-time support checks the congruency between the format requested by the program and that selected on the screen. The screen format cannot be changed dynamically.

Eventual messages are displayed on the penultimate line if the Screen Handling extent is used, or on the cursor line if the program uses the screen in sequential mode.

COBOL ICE

The COBOL-ICE programming environment allows program preparation with the Screen Section for any format.

The CRECOS source generator can only be used with the 25 x 80 format. The error messages are displayed on the penultimate line. The Screen Section in the programs generated by CRECOS is only compatible with the 25 x 80 format.

During the compilation phase, the size defined for the Screen Section is checked to ensure that it does not exceed 24 rows by 80 characters.

Program debugging, using the Symbolic Debugger in the COBOL-ICE environment, must be executed using the 25 x 80 format.

When the program is executed no control is carried out on the congruency between the format requested by the program and that selected on the screen. This is the responsibility of the user. The eventual error messages issued by ICRTS are compatible with any type of screen and format.

The services of the interactive COBOL-ICE environment are only compatible with the 25 x 80 format.

BASIC Interpreter

The BASIC language interpreter allows programs to be written for any type of screen and format. The line editor can be used for this purpose, also on any type of screen and format.

Program debugging, using the interpreted BASIC debugger, is executed on any type of screen and format.

The programs can be executed on any type of screen and format. The interpreter checks the congruency between the format requested by the program and that selected on the screen. Eventual error messages are compatible with any type of screen and format.

The interpreted BASIC graphic features are not compatible with the normal alphanumeric 5", 9", 14" and 15" screen, but require a 15" graphic screen with a 25 x 80 format (2000 characters). The format cannot be changed on this screen.

Compiled BASIC

Compiled BASIC allows program preparation for any type of screen and format.

The format cannot be changed in dynamic mode (but via a statement: MARGIN).

Eventual error messages issued by the language's run time support are displayed on the system line, if present, or on the last line of the screen.

The compiled BASIC graphic features are not compatible with the normal alphanumeric 5", 9", 14" and 15" screen, but require a 15" graphic screen with a 25 x 80 format (2000 characters). The format cannot be changed on this screen.

FORTRAN

FORTRAN allows program preparation for any type of format. This language does not support the concept of format, therefore FORTRAN programs can be executed with any type of screen and format. The format cannot be changed in dynamic mode.

Eventual error messages issued by the language's run-time support are displayed on the cursor line.

The FORTRAN graphic features are not compatible with the normal alphanumeric 5", 9", 14" and 15" screens, but require a 15" graphic screen with a 25 x 80 format (2000 characters). The format cannot be changed on this screen.

ESE

The ESE emulation environment of the P6066 Olivetti systems must be used with the 25 x 80 format.

SORT

The SORT utility program can be used with any type of screen and format.

VISA

The features offered by this package can be used with any type of screen and format. The congruency between the format to be interpreted and that selected on the screen is checked.

The screen format can be changed dynamically.

The format preparation and control phase (using the TFORM utility program) must be executed using the 25 x 80 format on 9", 14" or 15" screens. The VISA formats can, however, be created for all the possible screen formats (520, 1000, 2000 characters).

VISA S6000 Compatible

The features offered by this package can be used with any type of screen and format. The congruency between the format to be interpreted and that selected on the screen is checked.

The screen format cannot be changed dynamically.

The format preparation and control phase (using the TFORM utility program) must be executed using the 25 x 80 format on 9", 14" or 15" screens. The VISA S6000 compatible formats can be created for all possible screen formats (520, 1000, 2000 characters).

Symbolic Debugger

This component, which is used for interactively debugging programs written in COBOL or BASIC (the latter is not yet available), can be used with any type of screen and format.

The programs being debugged use the screen as stipulated by the languages in which they are written.

MTS

Programs in the MTS environment configuration can be executed with any type of screen and format. Formats cannot, however, be changed dynamically: the format must be selected before the program is activated via Grandpa or Shell.

The lowest available line is used as a "message line" irrespective of the screen or format used. The "message line" is used in MTS environments during the LOGON procedure, by the functions "Sending Immediate" and "Broadcasting Immediate", and for displaying anomalous situations.

The format of programs which have been declared in the MTS environment configuration cannot be dynamically changed.

7. M30 / M31 / PERSONAL COMPUTER / VT100-like TERMINALS

This chapter describes the use of some systems and non-integrated terminals connected to L1 MOS systems.

Each of the three sections that make up the chapter deals with a different system or terminal which can be used as a work station in an L1 MOS system. They are:

- M30/M31
- OLIVETTI Personal Computers
- VT100-like terminals

M30/M31 USED AS L1 MOS WORK STATION

This section describes the M30 or M31 system used as a work station in an L1 MOS system.

The description is valid for both systems, apart from where one of them is referred to explicitly, as the only difference between them is found in hardware components which do not, however, influence their behaviour as non integrated work stations in an L1 MOS system.

SOFTWARE REQUIREMENTS

From the software point of view it is necessary, during the L1 MOS operating system configuration phase, to define the M30/M31 as a work station with characteristics that are congruent with the way in which the MOS operating system was configured for the M30/M31; this ensures correct line management in the connection.

The version of the MOS operating system used on the M30/M31 is equipped with all the components necessary for using the M30/M31 itself as a work station in an L1 MOS system: the terminal driver, the RS232/CL driver and the printer driver.

See the manual System Software Generation and Installation, User Guide for greater detail.

USES OF M30/M31 AS WORK STATION

The M30/M31, used as a work station in an L1 MOS system, can perform most of the normal activities possible on standard terminals, bearing in mind, however, the functional limitations which are described below.

Available Application Environments

The following application environments are available via the M30/M31 used as a terminal in an L1 MOS system:

- Shell
- BEAM
- COBOL ICE
- Interpreted BASIC
- DMS
- MTS
- ESE
- OWS 2

as well as applications written in the following languages:

- COBOL
- COBOL ICE
- Interpreted BASIC
- Compiled BASIC
- FORTRAN
- PASCAL+

The PGU and GSP graphic functions, along with those provided by the VISA package, and the utility programs offered by the MOS operating system (for example, Editor, SORT, TFORM, etc.) are also available. For the limitations see the related manuals.

Limitations

The existing limitations on the M30/M31 used as work stations in an L1 MOS system are:

- It is not possible to use the graphic facilities provided by the ESE environment.

- It is not possible to use the non-OLIVETTI Terminal Emulator environment software.

- Connection of a work station using a RS232/CL channel does not allow more than one window to be handled. It is not possible, therefore, to use the split option relative to this work station, and consequently the work station in question must not be defined as the master terminal in the Grandpa configuration file.

- Due to the above reason, limitations relative to Message Switching exist.

- When operating in Shell environment, the non-availability of the asynchronous window renders the execution of the KILL, SUSPEND and RESUME command impossible. The function of the KILL command can however be obtained by typing the /CONTROL/ /K/ keys, but the other two are not available.

- A work station printer, connected to the M30/M31, can only be used to obtain alphanumeric hard-copy.

- Colour is not available on the M31.

- Programs using some of the MOS operating system PASCAL+ primitives for handling the work station (Split, SignalErrTm, WriteInfoWs) cannot be executed, and the ReadInfoTm primitive is limited in use (see the manual PMM and Driver Primitives, Reference Manual for greater detail).

PERSONAL COMPUTER USED AS AN L1 MOS WORK STATION

This section describes the use of an OLIVETTI Personal Computer as an L1 MOS work station.

Only the word PC is referred-to, but the description is valid for the M19, M24, M24SP and M28 Personal Computers.

The PC can function as a work station through the interaction of the following components:

- on the L1 MOS
  - . Work Station Management with or without Ports.
- on the PC
  - . emulation programs L1WSE, WSELAN or OLIEMU.

The emulation programs can:

- emulate the alphanumeric integrated work station of an L1 MOS system
- emulate the monochromatic or colour integrated graphics work station of an L1 MOS system
- handle work station printers connected to the PC
- store a screen page or print hard copy
- handle banking peripherals connected to the PC (PIN Pad, Badge Reader, CA2000, ML700/LP700).

For the limitations for each emulator program, see the next section.

Using the emulator programs byte-stream files can be transferred from an L1 MOS system to the PC and vice versa, by means of the TRANSF program on the L1 MOS, environment can be changed (PC as an L1 MOS system work station and Personal Computer at the same time) by means of the CONTSW program on the PC. For details as to use of the programs TRANSF and CONTSW, see the manual MOS - Operating Guide.

DESCRIPTION OF EMULATOR PROGRAMS

There are usually two work station emulation modes for the PC which depend on the physical connection and the emulating programs as follows:

1. PC connected as static work station (in local or remote mode) by means of the L1WSE program for general purpose functions, or by means of OL1EMU program for banking functions.

2. PC connected as dynamic work station over the network (OL1LAN) using the WSELAN program for general purpose functions.

More specifically:

- When using L1WSE, with connection via RS232 (MUX, SIC) or via the Current Loop (MUX, SIC), emulation is incomplete, because asynchronous window, work station printer and banking peripherals are not handled.

- When using OL1EMU, with connection via RS232 or Current Loop with protocolled MUX, emulation is complete because asynchronous window, work station printer with serial interface and OLIVETTI standard (PR/A) and banking peripherals are handled. The work station printer with parallel interface and IBM standard (PR/B) is not handled.

- When using WSELAN, with connection via the RS232 (MUX, SIC), via OMNINET or via ETHERNET, emulation is incomplete, because the printer PR/A and the banking peripherals are not handled.

The table and diagrams that follow illustrate what has been said above:

	L1WSE	WSELAN	OL1EMU
Asynch. Window	NO	YES	YES
PR/A	NO	NO	YES
PR/B	NO	YES	NO
Banking Periph.	NO	NO	YES

Tab. 7-1 The Difference between Emulating Programs

STATIC WORK STATION EMULATION

Static work station emulation (in local or remote mode) can be done by means of the L1WSE or OLIEMU emulator programs.

L1WSE

Emulation with the L1WSE program is done by means of the connection PC - L1 MOS via the RS232 (MUX, SIC) or via the Current Loop (MUX, SIC), using the following software:

- on the L1 MOS
  - . MOS operating system including the driver for the RS232/CL interface.
  - . TRANSF, command for file transfer (optional).
- on the PC:
  - . MS-DOS operating system.
  - . L1WSE, emulation program.
  - . CONF, configuration program for the L1WSE.
  - . OSKEM or GIOMR or GIOM, driver for the screen/keyboard handling. The module GIOM or GIOMR is only need for graphics emulation.
  - . FONT0,..,FONT7, font files needed for graphics emulation if the GIOM driver is used (optional).
  - . ASYNC2, driver for the serial line handling.
  - . CONTSW, program for changing context (optional).
  - . HCBM or HCBC, driver for graphic hard copy on the monochromatic and colour printers respectively (optional).

Main limitations arising from emulation by means of L1WSE program are:

- No asynchronous window handling.
- No banking peripherals handling (PIN Pad, Badge Reader, etc.).
- No work station printer handling.

The following should also be borne in mind:

1. For definition of the values to assign to the MOS and PC (CONF program) configuration parameters, if the PC is used as an L1 MOS work station, see the manual System Software Generation and Installation, User Guide.

2. For use of the PC as an L1 MOS work station, see the manual MOS - Operating Guide.

3. The EGC board is required for colour graphics on the PC

4. The drivers GIOM and GIOMR which are required for video/keyboards handling when in graphics emulation, differ in that GIOM delegates the PGU "output" functions to the emulator program, so that the MOS CPU does not have to handle them. For further information see the manual PGU, Programmer Guide.

OLIEMU

Emulation with the OLIEMU program is done by means of the connection PC - L1 MOS via the RS232, or via the Current Loop, with protocolled MUX, and with the following software components:

- on the L1 MOS
  - . MOS operating system, including the driver for the RS232 interface.
  - . TRANSF, file transfer command (optional)
- on the PC
  - . MS-DOS operating system.
  - . OLIEMU, emulation program.
  - . OLICONF, program for configuring the OLIEMU.
  - . OSKEM or GIOMR or GIOM, driver for the screen/keyboard handling. The module GIOM or GIOMR is only need for graphics emulation.
  - . FONT0,..,FONT7, font files needed for graphics emulation if the GIOM driver is used (optional).
  - . ASYNC2 or ASYNC3, driver for the serial lines handling. ASYNC3 is needed if three serial lines are handled. This driver then requires the T_PIM module (Polling Interrupt Module).
  - . CONTSW, program for changing the context (optional).
  - . HCBM or HCBC, driver for graphic hard copy on the monochromatic and colour printers respectively (optional).
  - . T_BPP, driver for handling banking peripherals such as the PIN pad, Badge Reader, CA2000, ML700/LP700 (optional). This driver then requires the T_PIM module (Polling Interrupt Module).
  - . T_KEYB, driver for the PB keyboard - 105 key model handling (optional).

  The main limitation resulting from emulation using OLIEMU is that PR/B printers cannot be handled.

The following should also be borne in mind:

1. For definition of the values to assign to the MOS and PC (OLICONF program) configuration parameters, if the PC is used as an L1 MOS work station, see the manual System Software Generation and Installation, User Guide.

2. For use of the PC as an L1 MOS work station, see the manual MOS - Operating Guide.

3. The EGC board is required for colour graphics on the PC

4. The drivers GIOM and GIOMR which are required for video/keyboards handling when in graphics emulation, differ in that GIOM delegates the PGU "output" functions to the emulator program, so that the MOS CPU does not have to handle them. For further information see the manual PGU, Programmer Guide.

DYNAMIC WORK STATION EMULATION

Dynamic work station emulation can be done by means of the WSELAN program. This type of emulation uses the Ports and is carried out by means of the connection PC - L1 MOS via RS232 (MUX,SIC), ETHERNET or OMNINET and with the following software components:

- on the L1 MOS:
  - . MOS operating system.
  - . appropriate interface-driver for the connection mode chosen.
  - . TRANSF command for file transfer (optional).
  - . Modules for the Ports.
  - . OLILAN "Name Server" component for connection via OMNINET and ETHERNET (optional).
- on the PC:
  - . MS-DOS operating system.
  - . WSELAN, emulation program.
  - . WSECONF, program for configuring WSELAN.
  - . Appropriate interface-driver for the connection mode chosen.
  - . Module for the Ports.
  - . OSKEM or GIOMR or GIOM, drivers for the video/keyboard handling. The module GIOM or GIOMR is only required for graphics emulation.

- FONT0,..,FONT7, font files needed for graphics emulation if the GIOM driver is used (optional).

- CONTSW, program for changing the context (optional).

- PRINTER, driver for handling the printer PR/B (optional).

- HCBM or HCBC, driver for graphic hard copy on the monochromatic and colour printers respectively (optional).

- VISA, module for handling the work station logically (optional). This module may require:

  a) VPI, module for the validation program handling on the PC.

  b) PRINTER, driver for handling the PR/B printer via VISA.

The main limitations that result from this emulation program are as follows:

- No banking peripheral handling (PIN Pad, Badge Reader, etc.)

- No printer PR/A handling (the printers PR/B are handled).

The following should also be borne in mind:

1. For definition of the values to assign to the MOS and PC (WSECONF program) configuration parameters, if the PC is used as an L1 MOS work station, see the manual System Software Generation and Installation, User Guide.

2. For use of the PC as an L1 MOS work station, see the manual MOS - Operating Guide.

3. For informations on network services (OLILAN) see the manual OLILAN Local Area Network - User Guide.

4. The EGC board is required for colour graphics on the PC

5. The drivers GIOM and GIOMR which are required for video/keyboards handling when in graphics emulation, differ in that GIOM delegates the PGU "output" functions to the emulator program, so that the MOS CPU does not have to handle them. For further information see the manual PGU, Programmer Guide.

6. The VISA module on the PC is used to relieve the MOS of the logical management of the work station. For further information see the manual VISA, Form Management Package, Programmer Guide.

USES OF THE PC AS A WORK STATION

Connection to an L1 MOS system does not effect the independent processing capacity of the PC. It can be used normally as a stand-alone computer using the hardware and software that has been configured for it.

If the PC is used as an L1 MOS work station, most of the activities available on a normal computer can be carried out, but the emulation causes the exceptions described below.

Application Environments Available

The following application environments can be activated on the PC used as an L1 MOS work station:

- Shell
- BEAM
- COBOL ICE
- Interpreted BASIC
- DMS
- MTS
- ESE

together with the following applications written in the following languages:

- COBOL
- COBOL ICE
- Interpreted BASIC
- Compiled BASIC
- FORTRAN
- PASCAL+

The graphic functions of the PGU and GSP and those of the VISA package can be used, in addition to the utility programs made available by the MOS operating system (e.g. Editor, SORT, TFORM, etc.). For the features and limitations for the emulator programs and the PC, see the corresponding manuals.

Limitations

The limitations for the PC used as an L1 MOS work station:

- It is not advisable to define it as a master terminal.
- The only ruling character available is underling (_).
- Terminal emulators that are not OLIVETTI cannot be used.
- The set of Japanese characters is not available.
- Programs using some of the MOS operating system PASCAL+ primitives for handling the work station (SignalErrTm, WriteInfoWs) cannot be executed, and the ReadInfoTm primitive is limited in use (see the manual PMM and Driver Primitives, Reference Manual for greater detail).
- If a program that calls the PASCAL+ primitive WriteTm is used, there are the following limitations on the commands that can be sent in the TCDS (Terminal Control and Data String):

COMMAND	LIMITATIONS
set scroll scroll	Partial scrolling of attributes is not available. A "scroll pattern" value other than 0 scrolls all the attributes.
set attribute reset attributes write attributes	The attributes not supported are: - upper line - left line - right line These attributes are ignored.
write background	This command (specific for the Kanji keyboard) is ignored.
write data	A "row" parameter value which is less than 0 (feature for the Hebrew field) is interpreted as zero.
cntl led	This command is ignored.
write status ext erase line ext save_restore cursor ext move ext move inside region ext set region ext new line mode ext set tabulation stop ext origin mode ext cntl led ext autowrap mode ext set character mode ext	These commands are ignored.

Tab. 7-2 Limitations on the Commands which can be sent in the TCDS

If the emulation with L1WSE is used there are the following limitations also:

1. Exclusion of screen handling. It is not possible to use the split option and consequently the asynchronous window is not available ("system line"). It is possible to interrupt the execution of a job, (using the key combination /CONTROL/ /K/).

2. Due to the above reason, limitations related to the Message Switching functions exist.

3. When operating in Shell environment it is not possible to call the SUSPEND and RESUME commands.

EMULATION OF THE L1 MOS KEYBOARD

The L1WSE, WSELAN and OLIEMU emulation programs allow the use of a standard keyboard (IBM/OLIVETTI PC), selected from the 14 national versions available which can be connected to the PC, or a PB keyboard - 105 key model (only with OLIEMU). The keyboard version used must be specified in the configuration programs CONF, WSECONF and OLICONF, described in the manual System Software Generation and Installation - User Guide. To use the PB 105 key keyboard the driver T_KEYB must be loaded.

Standard PC Keyboard

The following table describes the correspondency between the function keys of the L1 MOS system keyboard and those of a standard PC keyboard.

L1 KEYBOARD	PC KEYBOARD
→ |	SHIFT F10
CNTL → |	ALT PGUP
| ←	SHIFT F9
CNTL | ←	ALT PGDN
CNTL ↓	CTRL F10
CNTL ←	CTRL F7
CNTL →	CTRL F8
CNTL ↑	CTRL F9
CNTL CL_ERR	CTRL F6
CNTL HOME	ALT HOME
CHANGE WINDOW	ALT F7
CLEAR	ALT F6
DC	DEL
DL	SHIFT F8
ERASE	ALT F5
EXIT	END
CNTL EXIT	CTRL END
HALT PRG	ALT →
CNTL HALT PRG	ALT ←
HARD COPY	ALT F10 - SHIFT SCR PRT

(Cont.)

L1 KEYBOARD	PC KEYBOARD
IC	INS
IL	SHIFT F7
SEND	CTRL F5
SKIP	SHIFT TAB
F5	F5 - PGDN
F6	F6 - PGUP
F11	SHIFT F1
F12	SHIFT F2
F13	SHIFT F3
F14	SHIFT F4
F15	SHIFT F5
F16	SHIFT F6
F17	ALT END
F18	ALT 5
F19	ALT ↓
F20	ALT ↑
F21	CTRL F4
F22	ALT F4
F23	CTRL F3
F24	ALT F3
F25	CTRL F2
F26	ALT F2
F27	CTRL F1
F28	ALT F1
P1	ALT ↓
SHIFT P1	CTRL PGDN
P2	ALT ↑
SHIFT P2	CTRL PGUP
P3	ALT PGDN
SHIFT P3	CTRL HOME
P4	ALT PGUP
SHIFT P4	CTRL →
P5	ALT HOME
SHIFT P5	CTRL ←
S2	CTRL F4
CNTL S2	ALT F4
S3	CTRL F3
CNTL S3	ALT F3
S4	CTRL F2
CNTL S4	ALT F2
S5	CTRL F1
CNTL S5	ALT F1

Tab. 7-3 Correspondency between the L1 MOS and PC Keyboards

Notes

1. The keys that are not specified have the same function on each keyboard.

2. The keys ALT F8 on the PC save the screen page in a MS-DOS file called "SCRPRT" which is present under the current directory. This file is automatically created the first time that the program is called, or is written in append mode if it exists already. This feature is not available for graphics.

3. Under CONTSW the keys SHIFT SHIFT cause control to pass from MS-DOS to the emulator and vice versa.

4. The keys ALT F9 on the PC are used to exit from the emulator program and return to MS-DOS. This cannot be done under CONTSW.

5. The BREAK key on the PC has no effect with the emulator programs L1WSE and WSELAN, while with the OLIEMU program it causes an error condition.

6. The keys ALT F10 on the PC make alphanumeric hard copy using the PR/A (OLIEMU program) or PR/B (WSELAN program) printer.

7. The keys for MS-DOS hard copy (SHIFT SCR PRT) have the following functions:

   - If the HCBC or HCBM driver has been loaded they execute the alphanumeric or graphic (if in graphic mode) hard copy using the /B printer.

   - If the HCBC or HCBM driver has not been loaded they execute the standard MS-DOS hard copy using the /B printer.

PB Keyboard - 105 Key Model

The following table describes the correspondency between the function keys of the L1 MOS system keyboard and those of a PB keyboard - 105 key model.

L1 KEYBOARD	PB KEYBOARD
F1	F1, SHIFT F1
F2	F2, SHIFT F2
F3	F3, SHIFT F3
F4	F4, SHIFT F4
F5	F5, SHIFT F5
F6	F6, SHIFT F6
F7	F7, SHIFT F7
F8	F8, SHIFT F8
F9	CONTROL F1
F10	CONTROL F2
F11	CONTROL F3
F12	CONTROL F4
F13	CONTROL F5
F14	CONTROL F6
F15	CONTROL F7
F16	CONTROL F8
F17	JUMP, SHIFT JUMP
F18	CONTROL JUMP
F19	ER FIELD, SHIFT ER FIELD
F20	CONTROL ER FIELD
F21	HOME, SHIFT HOME
F22	CONTROL HOME
F23	ALT F3
F24	ALT F4
F25	ALT F5
F26	ALT F6
F27	ALT F7
F28	ALT F8
/	F9, SHIFT F9, CONTROL F9
*	F10, SHIFT F10, CONTROL F10
-	ER IMP, SHIFT ER IMP, CONTROL ER IMP
+	CLEAR, SHIFT CLEAR, CONTROL CLEAR
HALT PRG	WRITE, SHIFT WRITE

(Cont.)

L1 KEYBOARD	PB KEYBOARD
CNTL HALT PRG	CONTROL WRITE
EXIT	CLOSE, SHIFT CLOSE
CNTL EXIT	CONTROL CLOSE
IC	PASTE, SHIFT PASTE
IL	CONTROL PASTE
DC	CUT, SHIFT CUT
DL	CONTROL CUT
|←	IC, SHIFT IC
CNTL |←	CONTROL IC
→|	DC, SHIFT DC
CNTL →|	CONTROL DC
CHANGE WINDOW	|←, SHIFT |←
HOME	→|, SHIFT →|
CNTL HOME	CONTROL →|
CLEAR	ERASE EOF, SHIFT ERASE EOF, CONTROL ERASE EOF
,	-, SHIFT -, CONTROL -
SKIP	COPY, SHIFT COPY, CONTROL COPY
ENTER	SKIP, SHIFT SKIP
HARD COPY (alphanumeric)	CONTROL |←, SHIFT PRTSC

Tab. 7-4 Correspondency between the L1 MOS and PB - 105 keys Model Keyboards

Notes

1. The keys that are not specified have the same function on each keyboard.

2. The keys ALT JUMP on the PC save the screen page in a file called "SCRPRT". This file must be under the same directory as the emulator program and is automatically created the first time that the program is called, or is written in append mode if it exists already. This feature is not available for graphics.

3. Under CONTSW the keys SHIFT SHIFT cause control to pass from MS-DOS to the emulator and vice versa.

4. The keys ALT WRITE on the PC are used to exit from the emulator program and return to MS-DOS. This cannot be done under CONTSW.

5. The keys ALT F1 on the PC have the function of BREAK, and cause an error condition.

6. The keys CONTROL |← on the PC make alphanumeric hard copy using the PR/A printer.

7. The keys for MS-DOS hard copy (SHIFT PRTSC) have the following functions:

   - If the HCBC or HCBM driver has been loaded they execute the alphanumeric or graphic (if in graphic mode) hard copy using the /B printer.

   - If the HCBC or HCBM driver has not been loaded they execute the standard MS-DOS hard copy using the /B printer.

WORK STATION PRINTER HANDLING

In case of emulation with the WSELAN or OLIEMU programs it is possible to connect a printer to the PC as a work station printer, with, respectively, an IBM standard parallel interface (indicated as PR/B) or with an OLIVETTI standard serial interface (indicated as PR/A). The user must declare the connection of the printer to the PC in the emulator program configuration phase (WSECONF for WSELAN and OLICONF for OLIEMU).

In emulation with OLIEMU there are no limitations regarding PR/A printers, whereas in emulation with WSELAN the following limitations and methods of use must be remembered:

- Printer operations which are defined as FRONT FEED (for handling special printings which permit an interaction with the operator) are not handled. (For example savings books)

- The "accessmode" field in the "OpenPr" primitive which starts the printing session must be defined as "DEFAULT".

- The "WritePr" primitive must be used with "TRASPARENT" printing mode. In particular the "TURN_PAGE" and "MAGNETIC" modes are not handled.

- The "ReadPr" primitive is not available and it returns a "UNIT DOWN" warning.

- The printing session is disabled only by means of the "ClosePr" primitive.

- Control codes related to graphics features are not handled.

- The "ReadInfoUnit" primitive does not return the printer type but returns the 40H value in the "PrType" field, which indicates that the printer is working in FREE RUNNING mode. The "PrConf" field codes are not meaningful.

The following table gives a list for the /B printer of the printer control functions which are emulated and their limitations.

FUNCTION	NAME (Standard 12)	LIMITATION
BOTTOM OF FORM	BOF	The ESC 0 sequence is sended to cancel the BOF.
DELETE	DEL	Cancels the DSS command too.
REVERSE INDEX	DRI	The value of the third parameter is calculated on the basis of the basic English and American line spacing value.
DOUBLE SIZE SET	DSS	The command is automatically cancelled when a LF, CR, VT, or FF is received.
HORIZONTAL POSITION INCREMENT	HPI	None
HORIZONTAL POSITION RELATIVE	HPR	The positioning is obtained by sending 'n' blanks.
HORIZONTAL SPACING 10 CRT	HS10	The command is automatically cancelled when a SI, ESC SI, ESC @ or ESC M is received.
HORIZONTAL SPACING 16 CRT	HS16	The command is automatically cancelled when a DC2, ESC M, ESC @ or ESC P is received.
HORIZONTAL SPACE PROPORTIONAL	HSPR	None
HORIZONTAL TABULATION PROGRAM	HTP	The first parameter of the command is ignored.
INCREASE INTENSITY RESET	IIR	None
INCREASE INTENSITY SET	IIS	None

(Cont.)

FUNCTION	NAME (Standard 12)	LIMITATION
LEFT MARGIN SET	LMS	None
PARTIAL LINE DOWN	PLD	It does not cancel a PLU previously defined.
PARTIAL LINE UP	PLU	It does not cancel a PLD previously defined.
RESET TO INITIAL STATE	RIS	None
SHEET LOAD	SHL_P	None
UNDERSCORE RESET	USR	None
UNDERSCORE SET	USS	The underscores are not recognised.
VERTICAL POSITION RELATIVE	VPR	The positioning is obtained by sending 'n' LF
VERTICAL SPACE 2I VERTICAL SPACE 3I VERTICAL SPACE 4I VERTICAL SPACE 5I VERTICAL SPACE 6I	VS2I VS3I VS4I VS5I VS6I	The value of the third parameter is calculated on the basis of the basic English and American line spacing value.
VERTICAL SPACE 8L	VS8L	None
VERTICAL TABULATION PROGRAM	VTP	The maximum number or tabulation which can be defined in a single command is 16. The maximum value of interlines per module is 127.

Tab. 7-5 Control Functions, Names and Limitations for the PR/B Printer

If the printer is connected to a L1 MOS system instead of a PC, all characters and only some control codes (LF, CR, etc..), related to the USA ASCII table code, will be recognised by the printer.

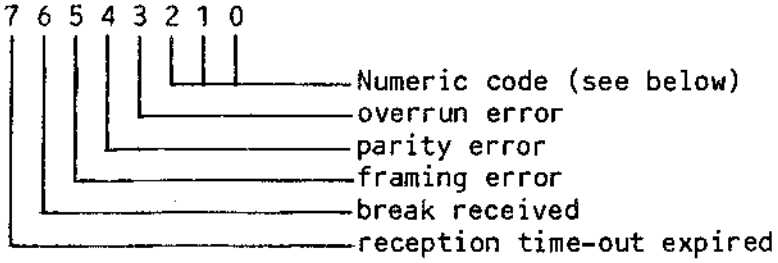
ERROR MESSAGES ON THE CONNECTION PC - L1 MOS

The serial interface-drivers ASYNC2 and ASYNC3 generate a set of error messages when anomalies are detected in the line activity between the PC and the L1 MOS system.

The error messages are displayed on the last line of the PC screen and have the following structure:

Line error XY print any character

XY are two hexadecimal digits whose binary value must be interpreted as following:



The numeric code can have the following values:

010	incorrect parameters (*)
011	line down
100	incorrect length (*)
101	DSR disabled
110	CTS disabled
111	non-existent channel (*)

(*) These errors occur with the OLIEMU program only.

Any combination of the previous errors is possible. For example:

If the XY value is 08 (00001000 as binary code) an overrun error has occurred.

If the XY value is 20 (00100000 as binary code) a framing error has occurred.

If the XY value is A2 (10100010 as binary value) the following error situation has occurred: reception time-out expired + framing error + incorrect parameters.

The user must bear in mind that to correct possible errors on the line he must check the parameters values into the CONF, WSECONF and OLICONF configuration programs.

VT100-like TERMINALS USED AS L1 MOS WORK STATION

This section deals with terminals whose interface is compatible with that of the Digital Research VT100 terminal (for example, the Olivetti WS584 terminal) and which can be used as a work station in an L1 MOS system.

These types of terminals will be referred to as "VT100" in this section.

SOFTWARE REQUIREMENTS

From the software point of view, it is necessary, during the L1 MOS operating system configuration phase, to define the VT100 terminal as a work station with characteristics that are congruent with those of the terminal itself. These (and its configurability) are described in the documentation provided by the supplier.

For greater detail on how to configure the MOS operating system refer to the manual System Software Generation and Installation, User Guide.

FUNCTION KEYS

The function keys of VT100 terminals have the task of issuing certain ESC sequences. The effect that these sequences have (on the terminal itself) are described in the supplier's documentation.

The use of these keys for executing activities in the MOS environment is, therefore, only meaningful if the the program being run is expecting to receive the ESC sequence generated by these keys.

In other circumstances the use of the function keys is neither meaningful nor influential.

USES OF VT100 AS WORK STATION

The VT100 terminal used as a work station in a L1 MOS system performs all the normal activities possible on a standard terminal, apart from a few exceptions which are outlined below.

Available Application Environments

The following application environments can be used on a VT100 terminal used as a work station in an L1 MOS system:

- Shell
- MTS

as well as applications written in the following languages:

- FORTRAN
- PASCAL+

It should be noted, with regard to an application program's handling of the screen, that not all the accepted ESC sequences are common to both the standard L1 MOS terminals and the VT100 terminals (see the manual PMM and Driver Primitives - Reference Manual in the "Terminal Driver" section for the ESC sequences accepted by the L1 MOS work station). Therefore, it is very important to refer to the documentation provided by the supplier of the VT100 terminal in order for the application program to be able to send ESC sequences acceptable to the aforementioned terminal.

The management of the screen by a program written in the FORTRAN language cannot use the FINFIELD function. It must use the appropriate ESC sequence.

Limitations

The limitations that characterize the use of a VT100 terminal are imposed by the inherent differences between it (the VT100 terminal) and the standard L1 MOS terminal.

The following features are not available:

- The graphic functions and colour.
- The bivalence and trivalence functions (and therefore it is not possible to change, dynamically or otherwise, the video format).
- The subdivision of the screen into windows (it is not possible to use the split option relative to the VT100 terminals; therefore the terminal in question must not be defined as master terminal in the Grandpa configuration file).

When operating in Shell environment, it is not possible to invoke those commands which display a video format. These commands are:

- BATCH
- CLEAR
- DISKCHECK
- HEXED
- KILL
- MORE
- MTA
- RESUME
- SETTERM
- SHTERM
- SPOOL
- SUSPEND
- USERMAN

The function of the KILL command can however be obtained by typing the /BREAK/ key.

It is not possible to invoke the utility programs offered by the MOS operating system (for example, Editor, SORT, TFORM, etc.).

Limitations relative to Message Switching functions non-availability of the window handling feature.

Programs which require the MOS operating system PASCAL+ primitives for handling the work stations cannot be executed, with the exception of the following:

OpenTm
CloseTm
ReadTm
WriteTm
GetWsName
ReadInfoWs

8. SEGMENTS, FAMILIES AND PROCESSES

This chapter describes how a program is executed in the L1 systems, illustrating the elements (segments, families, processes) on which program execution is based under the control of the MOS operating system.

The memory is seen as a set of segments, some of which are reserved for the system and some for the programs which, from the system's point of view, are users. Each segment is a continuous area of memory not greater than 64 Kbytes.

The segments are the basic units which provide a logical address space.

The programs are sets of codes and data which are loaded in memory, starting with the files (which have a loadable format and are known as l-modules) created by the linker.

When a series of program statements are executed this is known as a process. The processes are potentially independent of each other, and are the primary agents of the processing procedures.

One of the system components (the PMM, Process and Memory Management) has to prepare the program execution environment. This means handling the logical address space where the program resides, loading the program in the physical memory, activating the program's execution by creating the relative process and assigning it control of the CPU.

FAMILIES

The processes are grouped into two categories:

- Coresident

- Disjointed

A family is formed by one or more processes. The family is assigned control of the processing, which means that it has visibility of the logical segments. The coresident processes share the family's address space (except for the stack - each process has its own). They cooperate in executing the family's activities: they can pass control among themselves (when the running process suspends) without causing the rescheduling of the activities to be executed.

Control of the processor among the various active families (those which have at least one ready process) is assigned for a time slice.

A series of information (size, physical location and attributes of the segments used) is stored for each process in appropriate registers handled by the MMU (Memory Management Unit).

Note: Each L1 MOS system is equipped with one MMU, except the M60 which may be equipped with either one or two MMUs. The following description applies to systems with one MMU. See the Section "M60 With Two MMUs" for further information about this specific case.

When control of the processor is taken from a process and passed to another belonging to the same family, the contents of the two registers are saved in a table (handled by the PMM) so that that process can correctly restart its execution at a later date (Partial Context Switch).

Disjointed processes belong to different families whose activities are not strictly connected. Control of the central unit is assigned to these processes according to the priority of their family.

When, however, the new process must take control from another family, the contents of all the MMU registers are saved with information on the user segments used by the family to which the process which has lost control belonged to (Context Switch).

The table in which this information is saved (PMM Segment Table) thus contains information relating to the 30 user segments used by each family, plus that relating to the 2 segments characterizing each process.

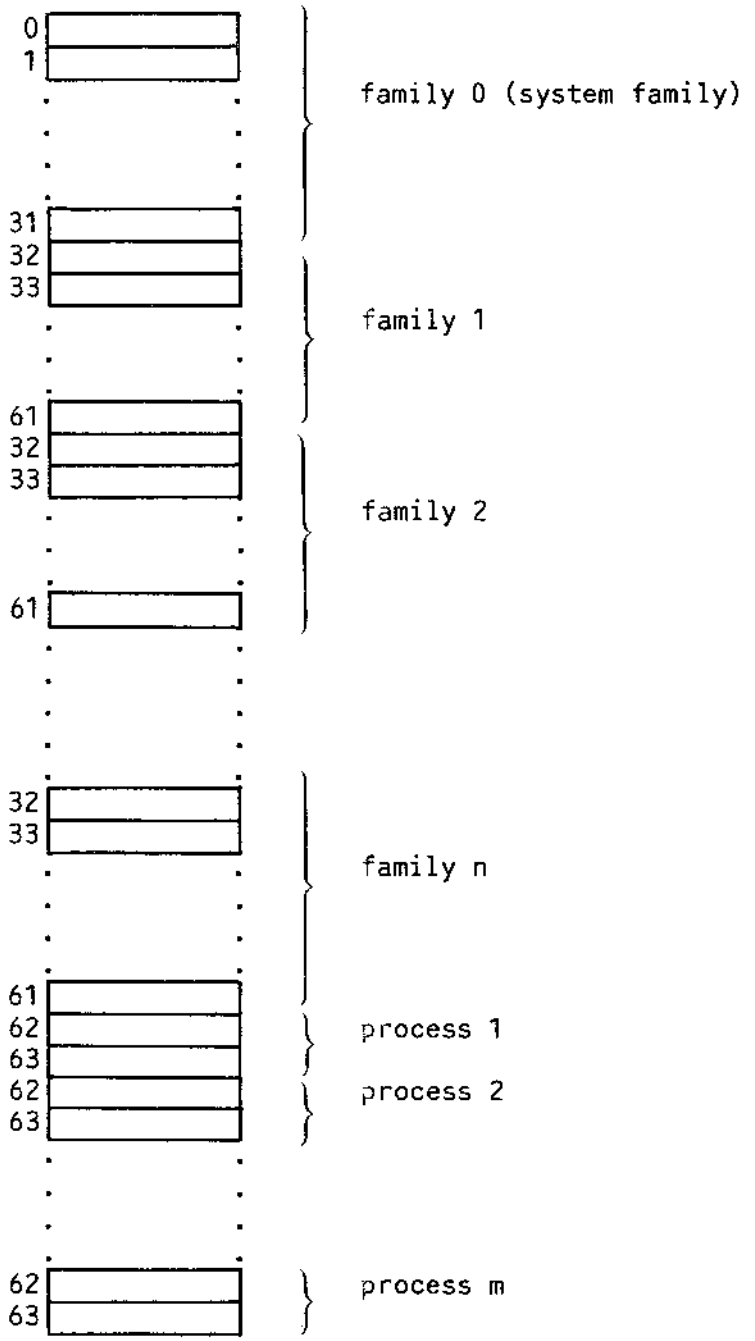


Fig. 8-1 PMM Segment Table (1 MMU only)

The information contained in this table allows the PMM to handle parallel execution of separate families (multiprogramming).

See the section "Assigning CPU Control to the Families" later in this chapter for further details.

SEGMENT ALLOCATION TO THE FAMILIES

The families are created according to a hierarchic structure. A "father" family exists for all the others which creates "son" families. Each of these can create others, which will be their "sons", and so on.

When a family has control of the central unit (when it is active) it uses a total of 64 logical segments for its address space.

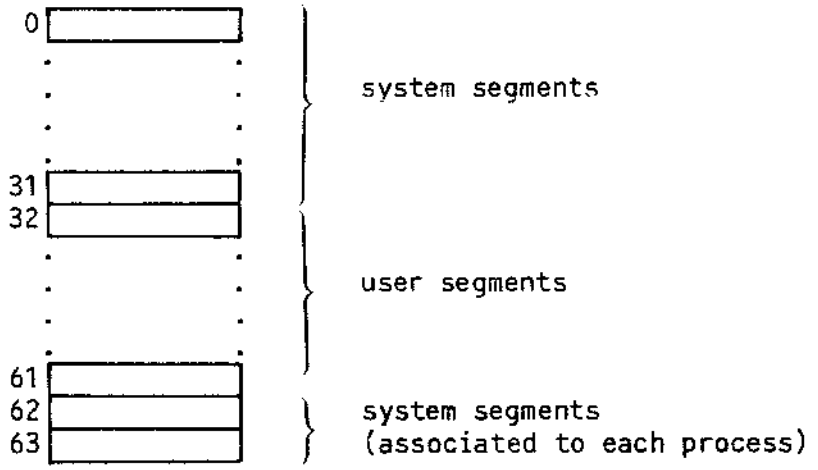


Fig. 8-2 The Segments (1 MMU only)

The segments from 0 to 31 inclusive are allocated to the operating system, and contain the system's code and data area. These segments are shared among all the families, including those which are created during a processing.

Segments 32 to 61 are known as user segments, and are allocated to a family. They are shared between all the coresident processes of this family and contain the data and code of the user programs (for example, the Shell command interpreter, the BASIC interpreter, an application program, etc). One of these programs can occupy more than one segment. The allocation of particular user segments to the code and data area of the application programs can be requested by the user during the linking phase, via a suitable linker option (OLINK or ZLOC).

Segments 62 and 63 belong to each process (each coresident process of the family has a copy of these two segments). Segment 62 is reserved for the system, whilst segment 63 is the process's stack.

M60 WITH TWO MMUs

The M60 system may be equipped with a second MMU. In this case the number of segments is doubled: the total number of logical segments is 128. They are grouped as shown in the following figure.

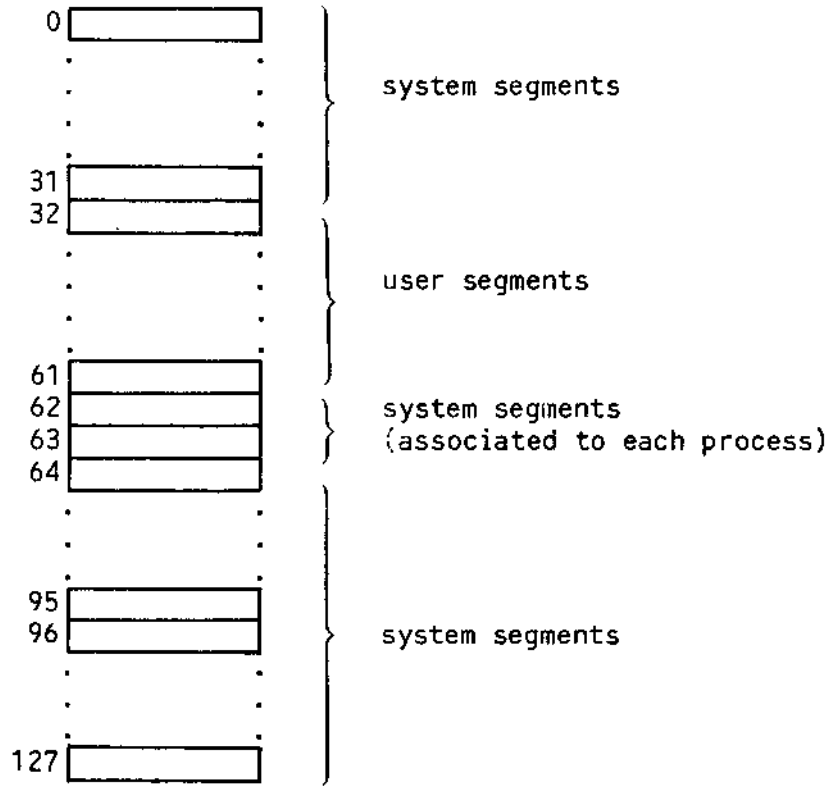


Fig. 8-3 The Segments (M60 with two MMUs)

The second MMU adds a set of logical segments (64 - 127) which are used by the system and which expand the configurability features.

FAMILY ATTRIBUTES

When a new family is to be created, the father family can assign a series of attributes to the son which determine its execution environment:

priority: is the priority of all the processes of the new family.

time slice: is the time slice assigned to the new family.

budget: is the number of memory pages (each with 256 bytes) to be allocated to the new family.

private segments: is the number of segments which will be considered private property of the new family (apart from segments 62 and 63, which are always private property of a process).

The new family's address space is initially empty, except for the system segments (0 - 31) which all the families have.

The segment area which has been defined as private property of the new family is that included between the highest available user segment (61) and the highest private segment of the father family.

The segments which are lower than this last are shared among the father and its son, therefore the son is not given exclusive control of them.

A series of detailed information can be obtained on the segments, families and processes using the NOSE utility program, which can be activated in the Shell environment.

ASSIGNING CPU CONTROL TO THE FAMILIES

The time-sharing policy is used for sharing the CPU between the families.

A family is defined as a set of processes which reside in the same address space. A family is active when at least one of its processes is carrying out an activity of the family.

CPU control is assigned to a family by the system: the family's processes have equal priority and can use the CPU without being interrupted by another process of the same family. When a process suspends to wait for a resource or an event, CPU control is assigned to another process of the same family.

The decision not to interrupt a process in favour of another of the same family is made because the two processes cooperate in executing the family's activities and so there is no reason to consider them as competing for CPU control.

The time-sharing policy is used for handling the families. When a new family is created, it is given a time-slice value. This value is a multiple of the system's time unit, which is equal to 100 msec.

This value indicates the time for which a family can control the CPU, unless other families with a higher priority become available in the meantime (see the section below "FAMILY PRIORITY") with at least one "ready" process.

When a family's time-slice expires, CPU control is taken from the process currently using it and given to another family.

If the time-slice value has been defined as less than zero, the family cannot be interrupted. In other words, when the CPU is assigned to a ready process of this family, the process only releases it when it has finished its activity or when it suspends to wait for a resource or an event.

This possibility allows families executing activities in real time, where the time-slice policy cannot be applied, to be handled simultaneously with families whose activities allow this policy to be used. The latter type of activity is executed when there are no real time activities in progress.

FAMILY PRIORITY

When a family is created, it is assigned a priority. This priority is used to identify all the processes belonging to that family.

A new family is created by an existing family and the latter is then known as the father: in order to guarantee that the father family always has control of its sons, the new family's priority is the value assigned to it plus that of the father family.

The highest priority value is 1, and the lowest is 32767.

The family with the highest priority is the one which executes Grandpa's activities. It receives CPU control as soon as the IPL phase has finished and it creates the families which will execute the activities forecast in Grandpa's configuration file (assigning them the priority decided by the user): this family is, therefore, known as the father of all the families. When a family dies, because its activities have finished, control is returned to its father family (the family which requested its creation).

The user can determine the families' execution priority during the normal system activities.

The PRIOR command is available for this purpose, and can be activated in the Shell environment. It lowers the priority of all the programs activated after it has been d called.

The value supplied by the user is added to the values of the family priorities that are to be created. In this way they are penalized, from the point of view of execution priority, compared with the activities that were already in progress when the command was carried out.

To return to the previous situation, in order to eliminate any alterations to the priorities assigned by the user, this last can simply call the PRIOR command with a value of 0 as its parameter.

SEGMENT TYPES AND ATTRIBUTES

Each segment is characterized by a type and series of attributes which can be assigned during the linking phase, via the ATTRIBUTES command of the linkers (OLINK or ZLOC).

The segment "type" informs the system of its contents and how to handle it. The table below lists the possible types.

TYPE		SEGMENT CONTENTS DESCRIPTION
1		XQTCODE : execute-only code
2		RDCODE : code which can be executed and/or read
3		RWCODE : code which can be read and/or written
4		RDDATA : read-only data
5		RWDATA : data which can be read and/or written
6		STACK : process stack
7	(*)	RLRDCODE : relocatable read-only code
8	(*)	RLRWCODE : relocatable code which can be read and/or written
9	(*)	RLRDDATA : relocatable read-only data
10	(*)	RLRWDATA : relocatable data which can be read and/or written

(*) The relocatable segments are reserved for the interpreted programs.

Tab. 8-4 Segment Types

Types 1, 2, 4, 7 and 9 identify the segments which can be shared among several families. That is, if the same l-module is loaded by more than one family, segments of the type 1, 2, 4, 7 and 9 are loaded only once.

A segment's "attribute" determines the type of hardware protection to be set. The following table lists the possible attributes.

ATTRIBUTE	DESCRIPTION	
0	RDWRT	: the segment can be accessed for execution or for read and/or write operations
1	RDONLY	: the segment can be accessed for execution or for read operations
8	XQTONLY	: the segment can be accessed for execution only
32	STCKATTR	: the segment is used as stack

Tab. 8-5 Segment Attributes

The segments which should have different "type" or "attribute" values from those indicated in the previous two tables, or which might present incongruencies between these two values, could not be loaded into memory. It is the responsibility of the user, whenever he does not use the linker automatically, to guarantee their correctness and compatibility.

Modifying the Size of the Segments

The system analyst can alter the size of one or more user segments. This is done by giving a signed value, when the SetSegment primitive is called (see the manual "PMM and Driver Primitives - Reference Manual"), which expresses the unit number (each of 256 bytes) by which the size of the segment must be increased or reduced (according to whether the value is positive or negative).

Only the size of segments 3, 5, 6, 8 and 10 can be modified. The size of the segments reserved for the system cannot be modified.

ASSIGNING AND USING THE USER SEGMENTS

When the system has been initialized, control is passed to the process which must start all the forecast activities. This process, known as Grandpa, is the "father" of all the families. A detailed description of its functions is given in the Chapter "Activating the Programs and User Subsystem" later in this manual.

When Grandpa has loaded all the user packages indicated in its configuration file (for example, the QUEMAN queue manager, the Message Switching functions package of a Transaction Handler, if forecast, etc), it establishes which is the segment with the highest number currently being used in its address space, apart from the segments used by Grandpa itself.

The segments with lower numbers will be shared among all the families. Therefore, all the private segments allocated to the families created by Grandpa, and their eventual descendents, will have higher numbers than this.

It is possible that some of the user segments included among those established by Grandpa as shared among the families are, in fact, not used. This is the ideal segment for loading an eventual functions package written by the user (and to be indicated during the linking phase of that package).

The user identifies which eventual free segments will be used for this purpose, bearing in mind that the system packages loaded by Grandpa are located as indicated in the table below. The following table shows which user segments are allocated to the various components.

Information is not given on segments reserved for components resident in private spaces and which therefore do not influence the program preparation activities.

COMPONENT	ALLOCATED SEGMENTS
COMMIT	40, 41
Login Program	55, 56, 57
User Packages	
QUE_LMS (QUEMAN & LMS)	32
QUEMAN	32
NMS	34, 35, 39
MSWMAN (Message Switching)	33
MTSCTLG (MTS)	34
MTSCTLG & CSCHEMA (MTS)	34
BEAMMON (BEAM)	34, 35
SLAM (ONE)	36
EEAUP (NEMOS)	36
WBF (Term. Emul.)	37, 38
LUINTERFACE1 (Term. Emul.)	37, 38
LUINTERFACE2 (Term. Emul.)	37, 38
MTS	
SMAN	39, 40, 41
LMAN	39, 41
GMAN	39, 45
TUMAN	42, 45, 46
GTSMAN	42, 45, 46
COMAN	40, 41
TBMAN	40, 41

(Cont.)

COMPONENT	ALLOCATED SEGMENTS
ESCHEMA	43, 44
OVLSMAN	43, 44
OVLGMAN	46, 47
BUFSEC	42
BUFTU	43
BUFITSC	44
Graphics	
PGU	43, 44, 45
RTGSP	43, 44, 45, 46, 47
VISA	
MONITOR	42
INTERPRETER	55, 57, 59, 61
COBOL	
RTS	56, 57, 58, 59, 60, 61 (*)
BASIC	
RTS	57, 58, 59, 60, 61
FORTRAN	
Default allocation for user programs	48, 49, 50, 51, 52, 53, 54 55, 56, 57, 58, 60, 61
PASCAL+	
Default allocation for user programs	59, 60, 61

(*) Segment 56 is only occupied if the SORT utility is used. Segment 60 may be not occupied if a reduced configuration of the Run Time is present.

Tab. 8-6 User Segment Allocation

In order to avoid the danger of a family attempting to create a son family and allocating it a segment which is already occupied, when the user-written programs are linked the user must ensure that the segments are allocated in descending order, starting from the highest allowed (61), using the BLOCK DESCRIPTOR command of the linkers (OLINK or ZLOC).

Supplementary Notes on the Use of the User Segments

From now on, the first free segment whose number is immediately above those occupied by all the user packages will be referred to as "GPASEG". GPASEG has a variable value depending on which packages have been loaded by Grandpa (from the system packages listed in the table above, or other user-written packages - see the section "The User Packages").

The created families can use the segments between GPASEG and 61 (this is the segment with the highest number available to the user).

In order to pass CPU control between the families, each one must have customized segments available, according to the segment visibility given to them by the 'father'. This means that a segment's contents may vary according to the family in execution.

This concept allows alternated execution of unconnected families which are dedicated to different activities, and each family is guaranteed a large number of available segments.

Segment availability for certain application environments or programs, activated by Grandpa, is summarized below.

The diagrams should provide a simple graphic explanation of how the user segments are assigned and used. Each vertical column represents a family. The name of the activity carried out by this family is given at the top of the column, with indications in the column on the contents of the segments and the family's visibility of them.

Family creation is indicated by an arrow (————>).

The value given in brackets in the arrow indicates the number of private segments assigned by the creator family to the new family. The value indicated by (X) is the result of the expression (30 - the number of segments reserved for the user packages). If the value is given as (S), it means that the creator family assigns the same number of private segments as it has to the family which it creates.

The components which are indicated in brackets are optional: they may not be present.

Programs activated with the INIT and TERM commands have a family available whose address space goes from 32 to 61. These programs exist only when system activities are initialised and terminated and they are therefore not considered as contending for segment use.

The user packages loaded with the CALL or PCALL command are allocated in the Grandpa family. The loading of the user packages determines the range of the segments that Grandpa assigns to the interactive and non-interactive programs. The interactive programs are allocated in the address which goes from GPASEG to 61.

Programs activated with commands TTYx, ALLT and all non-interactive programs activated with commands FG, START, BG and PFG have a family available whose address space goes from GPASEG to 61.

Shell

The Shell program is activated in a family which goes from GPASEG to 61. The application program activated by Shell is resident on a family, daughter of the Shell family, which goes from GPASEG to 61.

Graphics

The PGU graphics package is loaded into the same address space as the program which uses it and is allocated to the segments in the range from 43 to 45.

RTGSP, on the other hand, is allocated to the segments in the range from 43 to 47.

MTS

The MTS software environment requires the CSCHEMA and MTSCTLG packages loaded by CALL in the Grandpa configuration file in segment 34.

To permit the activation of transactional and/or interactive environments GPASEG must be < = 39.

The family executing the controller program of the interactive activities (consisting of either SMAN or LMAN) can use the segments from GPASEG to 61.

The interactive program is executed in the same family as the controller program and, therefore, can use the segments from GPASEG to 61 with the exception of those occupied by SMAN (or LMAN), ESCHEMA, COBOL Run Time Support, graphics and VISA.

The OVLSMAN module, which is the transient part of the CE (Client Environment), is loaded by SMAN into the segments 43 and 44 and unloaded before the ESCHEMA uses them.

The application program cannot occupy the segments occupied by OVLSMAN because this module executes the application program loading before ESCHEMA unloades it.

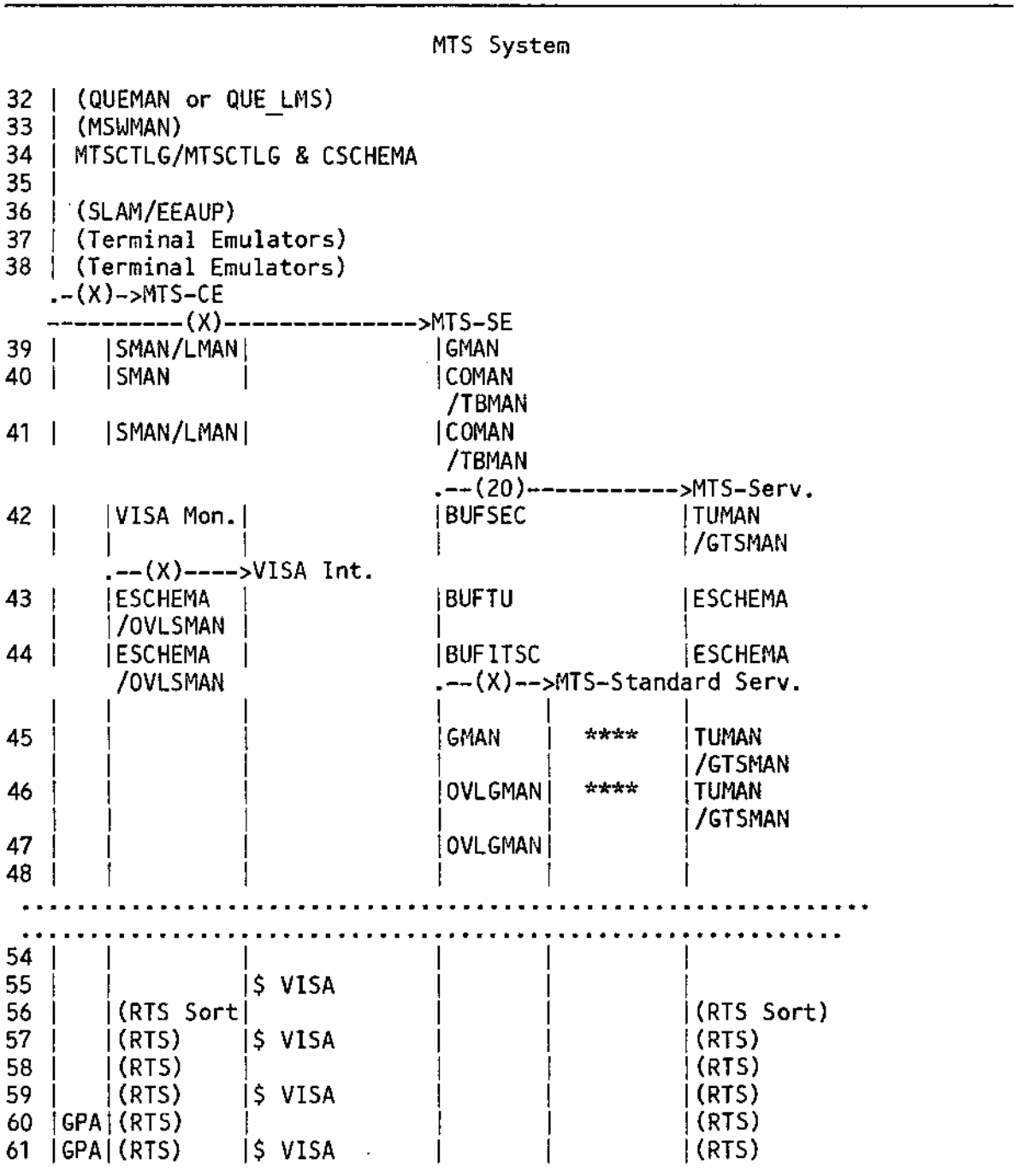
In case of program structured in overlays, only the MAIN module related to them cannot be loaded in the segments occupied by OVLSMAN which, however, can be occupied by overlays loaded subsequently.

In case of PASCAL+ programs, the program loaded by OVLSMAN cannot occupy the segments occupied by this module, but these segments can be occupied by other programs activated by the first one.

The 'main' program of the transactional environment (consisting of MAIN) can use the segments from GPASEG to 61. It is activated by START or FG in the Grandpa configuration file. This 'main' creates the families which execute the Servers; these families can use the segments from 47 to 61, with the exception of those occupied by COBOL Run-Time Support (for Server Programs written in COBOL).

The OVLGMAN module, which is the transient part of the SE (Server Environment), is loaded (and unloaded) by GMAN into the segments 46 and 47.

It must be remembered that an incompatibility exists between graphics and the Chained Data Base.



where **** can be every combination of STDSR (Standard Server), ITSC (Inter Transactional System Communication) or DUALLOG (secondary log update handler).

Fig. 8-7 MTS System

COBOL Programs

When a COBOL program is activated, the language's Run-Time Support is loaded into the segments from 56 to 61 of the family executing the program.

The program may therefore be loaded in the segments from GPASEG to 55, and also in segments 56 and 60 if the SORT utility is not used, and if a configuration with reduced Run Time is present.

If graphics are used, GPASEG must be < = 43 and the program cannot use the segments reserved for the graphics.

If VISA is used, GPASEG must be < = 42 and the program cannot occupy the segment occupied by the VISA Monitor.

If the program is executed under COMMIT, GPASEG must be < = 40 and the program cannot occupy the segments occupied by COMMIT.

If the Debugger is used, GPASEG must be < = 41.

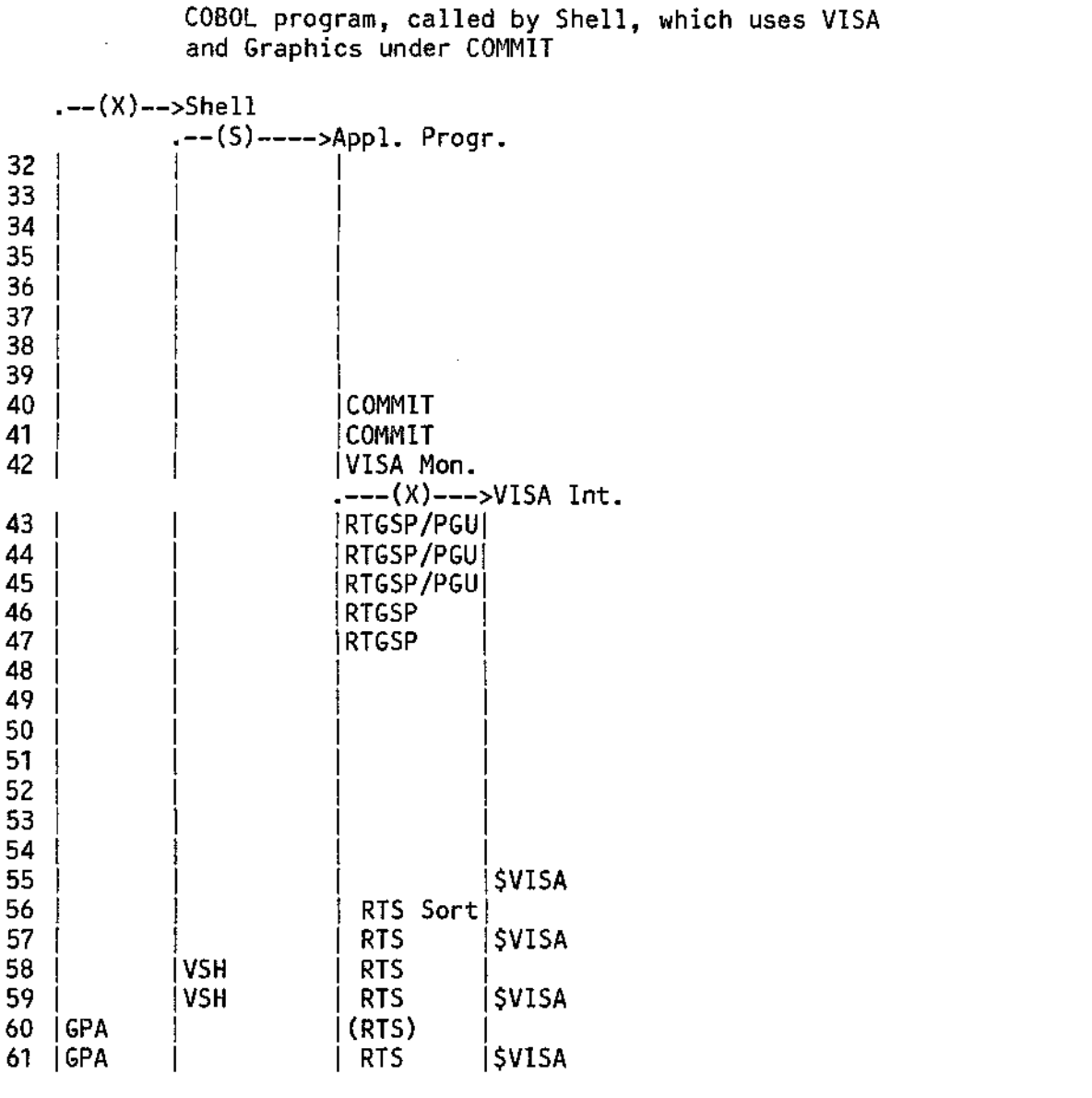


Fig. 8-8 System with a COBOL Program which Uses VISA and Graphics Under COMMIT

Compiled BASIC Programs

When a Compiled BASIC program is activated, the language's Run-Time Support is loaded into the segments from 57 to 61 in the address space of the family executing the program.

The program must, therefore, be loaded into the segments from GPASEG to 56.

If graphics are used, GPASEG must be < = 43 and the program cannot occupy the segments occupied by the graphics.

If the program is executed under COMMIT, GPASEG must be < = 40 and the program cannot occupy the segments occupied by COMMIT.

If the Debugger is used, GPASEG must be < = 41.

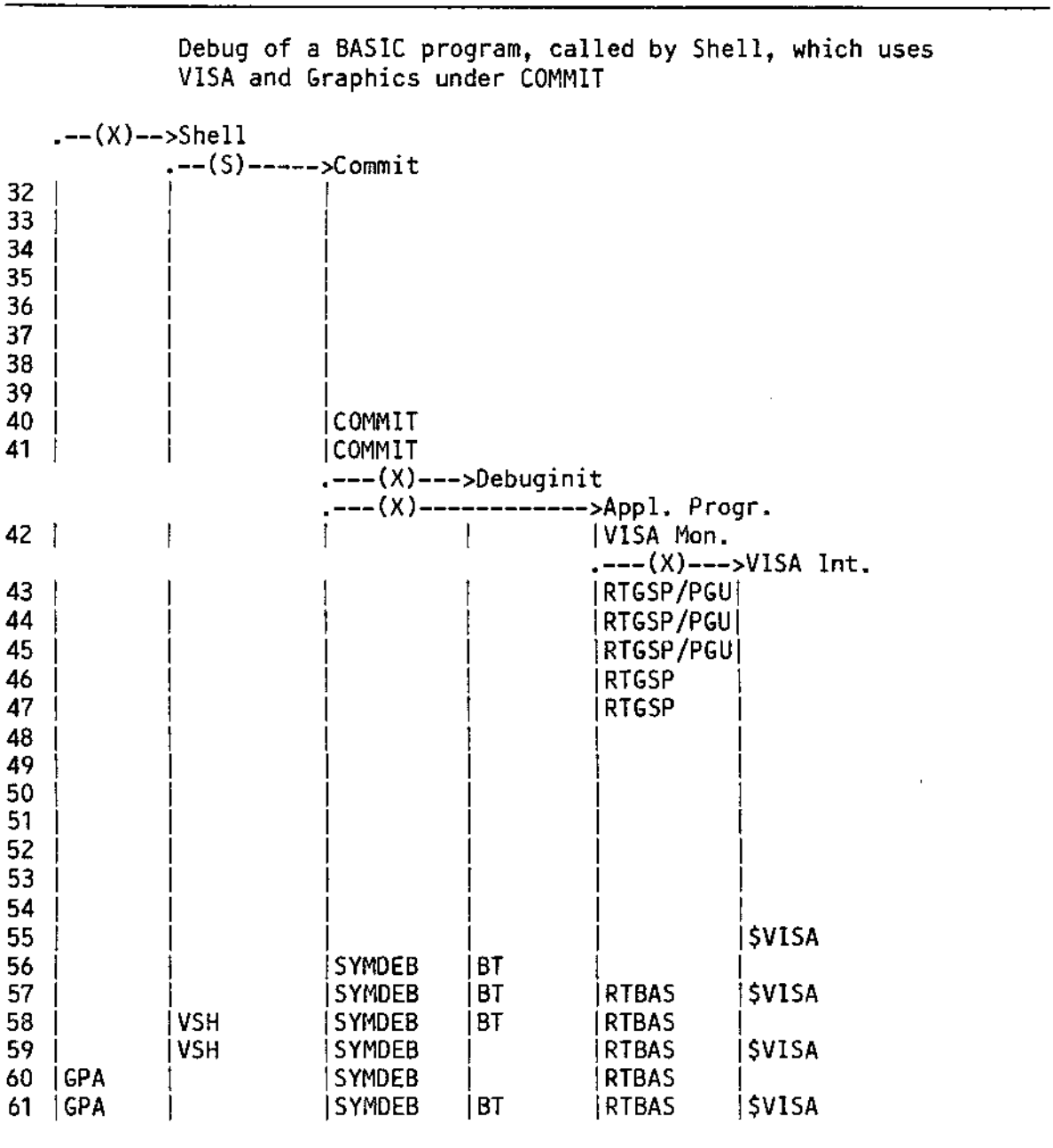


Fig. 8-9 Debug of a BASIC Program which Uses VISA and Graphics Under COMMIT

PASCAL+ Programs

The program can be loaded in the segments from GPASEG to 61. Default allocation is in the segments from 59 to 61.

If graphics are used, GPASEG must be < = 43 and the program cannot occupy the segments occupied by the graphics.

If VISA is used, GPASEG must be < = 42 and the program cannot occupy the segment occupied by the VISA Monitor.

If the program is activated under COMMIT, GPASEG must be < = 40 and the program cannot occupy the segments occupied by COMMIT.

If the Shell commands are activated, GPASEG must be < = 43.

If the PASCAL+ Debugger is used, GPASEG must be < = 42.

FORTRAN Programs

The program can be loaded in the segments from GPASEG to 61. Default allocation is in the segments from 48 to 58 and in the segments 60 and 61.

If graphics are used, GPASEG must be < = 43 and the program cannot occupy the segments occupied by the graphics.

Batch

The batch function is performed by the BTCHGPA module activated by Grandpa with a START, and by the QUEMAN module loaded in segment 32 with a PCALL by Grandpa. BTCHGPA then generates a family in which the UNSPOOL program executing the batch activities is activated.

Spool

The spool function is performed by the SPGPA module loaded by Grandpa with a START, and by the QUEMAN module loaded in segment 32 with a PCALL by Grandpa. SPGPA then generates a family in which the UNSPOOL program executing the spooling activities is activated.

BEAM

The BEAM is composed of two modules: BEAMMON and BEAM. The BEAMMON is loaded by Grandpa with a CALL in the segments 34 and 35. The BEAM is loaded in the segments 54 an 55 of a family created by Grandpa whose address area goes from GPASEG to 61.

Programs that can be activated by BEAM are loaded in the address space that goes from GPASEG to 61 of a family created by the BEAM module.

If the activity requires an interpreter the BEAM module creates a family

whose address space goes from GPASEG to 61, in which the interpreter is loaded.

Symbolic Debugger

"Symbolic debugger" "occupied segments" When the Debugger is activated, the MAIN module is loaded in the address space of the family which executes the program which invoked the Debugger. This module creates two families and loads in their address space the program to be debugged and the BT module which initialises the Debugger.

VISA

The VISA component consists of a monitor and an interpreter. Segment 42 of the family executing the application program which has called the VISA function is allocated to the monitor. The interpreter is loaded into segments 55, 57, 59 and 61 of a new family, created by the application program which is executing the requested VISA functions. As both the interpreter and the application program have to access the monitor, segment 42 must be shared between their families and it cannot be used by the application program.

Message Switching

The Message Switching service is executed by the module MSWMAN, which is loaded by means of a CALL in the Grandpa configuration file at segment 33, and by the modules MSWDIS, MSWROUTER or MSWLOCAL, activated by Grandpa in a START operation.

LMS

The LMS service is executed by the module SYSLOG, which is activated by Grandpa with a START operation, and by the module QUE_LMS, loaded with a call from Grandpa at segment 32.

NMS

The NMS user packages are loaded by a CALL from Grandpa in segments 34, 35, and 39. These are only used by the CMS component (Central Monitoring System) and are only installed on the machine that controls the network.

SLAM

The ONE user package (SLAM) is activated by a call from Grandpa, in segment 36.

NEMOS

The Network Monitoring service of the SNA network, is controlled by the user package EEAUP activated by a CALL from Grandpa, in segment 36.

Conclusions

An understanding of the organization of segment occupation means that the correct segment in which to load a user-written function package may be identified (see the Section "Notes on Writing a User Package").

Some general conclusions can be made on the basis of the information given so far, bearing in mind that segment occupation in a system must be evaluated by the user according to the components which are used.

For **stand alone** or **application/terminal server** systems, an incompatibility currently exists between:

- Terminal Emulators
- BEAM and MTS
- ONE network and NEMOS (network monitoring SNA).

This means that these application environments are mutually exclusive and cannot coexist simultaneously in a system: they can, however, both be used separately, each time initializing the system with different Grandpa configuration files. Each file requests the desired environment to be loaded.

The MTS application environment can coexist on a system with graphic functions if these are not called by the interactive program of the transaction application. (They are, instead, available to other programs which are not part of this application) The Server Environment of the transaction application may use neither graphics nor I/O towards the screen.

For **LAN multiserver** systems, an incompatibility currently exists between:

- BEAM and MTS application environments if the respective servers reside on the same system
- BEAM and NMS (they are mutually exclusive if reside on the same system)
- NMS and MTS (they are mutually exclusive if reside on the same system)
- ONE network and NEMOS (they are mutually exclusive if reside on the same system)

Furthermore, the terminal emulators must reside on the same system (Identified by the logical name 128) as the Line Manager.

USER VISIBILITY OF THE MEMORY OCCUPATION

The user can find out the memory occupation value of a program via the CSIZE command, which can be activated in Shell environment. It provides information relating to:

- the memory occupation of one or more l-modules (files in loadable format created by the linker) indicated by the user calling the command

- the segments allocated to these l-modules.

The occupation value given by this command refers to the memory occupation of the specified program, and not the space on disk occupied by the file containing the program.

A program's memory occupation can be found out without that program being loaded in memory: in other words the CSIZE command can be executed for a program which is not currently in memory but is resident on disk.

9. ACTIVATING THE PROGRAMS AND USER SUBSYSTEMS

The MOS operating system carries out the requested activities according to the concepts mentioned in the chapter "Segments, Families and Processes".

The application environments (Shell, BASIC interpreter, COBOL ICE, etc.), the interactive and non-interactive programs, the applications (programs or packages) written by the user are the activities to be indicated to MOS when it is initialized.

All these activities are executed in the user address space and are known as "user activities" because they use the operating system's services.

They can be executed in the following ways:

- Automatically and in operator-transparent mode, after the system has been initialized.

- On operator request, after system initialization, when a menu is displayed on the screen of the multifunctional work stations.

- On operator request, via interactive use of the Shell or BEAM environments, or one of the interpreters (BASIC, COBOL ICE,..).

The user can decide which activity or application environment is to be activated as soon as the system initialization phase has ended. He can also decide which activities are to be available on certain work stations (known for this reason as multifunctional, and on which an activity selection menu is displayed).

These selections are specified in the Grandpa configuration file, which is the system process which creates the user environment according to the directives given in this file.

The syntax and conventions to be respected when creating the Grandpa configuration file are described in detail in the manual MOS, System Software Generation and Installation, User Guide.

Substituting the System Disk

An option can be inserted in the Grandpa configuration file, reserved for the master terminal, to substitute the system disk before activating any activity.

Whilst MOS is being loaded, the magnetic support on which the system is loaded is logically connected to the memory volume, under the /IPL directory belonging to this volume.

The system modules, indicated in the system configuration file (named $CON) are loaded in memory from this disk and initialized.

If the NEWVOL function has been inserted in the Grandpa configuration file, the user can, at this point, unmount the system disk from the /IPL directory and:

- logically connect the disk to another directory (without physically replacing it)

- replace the disk with another and logically connect the new one to the /IPL directory

- replace the disk with another and logically connect the new one to another directory.

If the disk is physically replaced, the new support must contain another Grandpa configuration file ($CONFGP), as well as the components necessary to the system (the files with the programs to be activated such as, for example, the Shell commands).

If the new disk is logically connected to another directory, it must contain a suitably prepared Grandpa configuration file. The first part of the path name of the programs in this file which are to be activated is the name of the volume identifying the new disk during the "NEWVOL" phase. The system will identify these files by this path name.

To make this procedure clearer, we can examine a case in which system initialization starts from a floppy disk in a hardware configuration with diskette and hard disk.

For the sake of simplicity, the Grandpa configuration file makes the activities equally available to all the work stations (Shell and interpreted BASIC application environments, possibility of requesting system shutdown) as follows:

```
ALLT:NEWVOL!/IPL/SYS/$VSH!/IPL/DPC/BASIC!SHUTDOWN;
INIT:/A/DPC/CMD/MNT,<>HD1<>/IPL;
CALL:/A/$QM/CODES/QUEMAN,<>1;
START:/A/SP/SPGPA;
BG:/A/BE/BATCHGPA;
```

When a request is made on the master terminal (which, in this case, is the first terminal to be accessed as no specific one has been indicated) to replace the system disk (NEWVOL), one possible strategy is the following:

1. The floppy disk is not replaced.

2. It is logically connected as a new volume, with the name /A.

Grandpa continues its processing, which includes initializing and activating certain programs. The programs are identified in the Grandpa configuration file with the path name starting with /A, because this is the name with which the volume containing them has been logically connected. The user is responsible for ensuring their congruency.

When the select activity menu is displayed on all the terminals the user can:

- logically disconnect the volume resident on disk, which is used for the IPL phase, with the command UNMNT /A

- physically replace this disk with another, on which the system modules and program already activated (already loaded in memory) are no longer present. The new disk must contain the programs and/or data necessary for its activity

- connect the new volume at the desired point of the file system tree.

Note: The login data base, if present, must always be contained in the /IPL/ETC directory.

The Login Program

The login program is the program activated by Grandpa for starting the user-recognition operations.

It is executed as a normal user program, and indicates the start of a work session.

This program gives Grandpa the necessary information about a user and the request whether the screen is to be split into two windows or not.

Two default login programs are available, which can split the screens on all the work stations (LOGS) or, alternatively, on none (LOGN). The one selected must be stored on disk, at installation time, with the path name /IPL/SYS/$LOG.

It must be remembered that the master terminal (the terminal to which the keyword MASTER or GMASTER is associated in the Grandpa configuration file) requires the split option in the login program associated to it.

The user can write a customized login program, which can then be associated to one or more work stations (appropriately using the LOGIN label in the Grandpa configuration file, following the detailed description given in the manual MOS, System Software Generation and Installation, User Guide).

One of the following alternatives can, therefore, be used:

- All the system work stations can be used with the split option, with the LOGS login program as default. In this case, there are no problems for the master terminal.

- All the system work stations can be used without the split option, with the LOGN login program as default. In this case, a login program which will split its screen must be associated to the master terminal. This can either be the LOGS program, or a program written by the user following the rules given below.

- Besides the default login program, some user-written login programs can be associated to some (or all) of the work stations.

The user login program must include the following files:

PROT.d
PROT.i
ulog.d

and must link one of the following files:

aux_ui.obj
aux_ui.lib

It is necessary also that this program must give Grandpa the following information (using the "stdenv.quit" PASCAL+ function - see "Programs Activation" in the next section):

- A reply code, which must have a value of 127 if the login program has been correctly executed, or a value of 2 if not (in which case the following data is meaningless).

- A series of data with the following structure:

```
info = record
       case integer of
            1 : ( string : packed array[1..13] of char );
            2 : ( identity : T_identity;
                      name : T_login;
                     split : char )
       end;
```

where:

info.identity is the identity (consisting of two integers) which the system associates to the user

info.name is the user's login name

info.split indicates whether the terminal to which the login program is associated is to be split into windows (the character is S) or not (the character is N).

Remarks: If splitting has not been requested for any of the system terminals, the warning function of the master work station will not be available. The non integrated terminals which will be connected via the RS232 interface must not be indicated as master terminals as they cannot handle more than one window.

AUTOMATICALLY STARTING THE ACTIVITIES

EXECUTION CLASSES

MOS uses various classes for executing user activities and each of these classes has a different priority and a specific time slice.

Non-Interactive Programs

The non-interactive user programs, which are those with no interaction with any work station, are automatically activated after the IPL phase. Grandpa is responsible for their activation, and receives control when all the system's initialization procedures have finished and before any user process can be activated.

Grandpa starts its activities, reading the information in its configuration file, and the user can place the non-interactive user programs in four classes:

- server
- foreground
- standard
- background

Server

The activities which must be executed quickly and without interruption are inserted in this class. This class has the highest priority: as has been mentioned, only Grandpa is executed with a higher priority. The priorities which can be assigned to the programs belonging to this class are between 500 and 999. The CPU time slice assigned to this class's activities is infinite: when these activities are called they continue execution until they have finished. Note that these activities cannot access the procedures defined in user packages functions, because these packages are are loaded after the activation of server programs. A typical example of activities which should belong to this class are the Server Programs (both system and user) which constitute the Server Environment of a Transaction Handler, that is, the programs which provide the services called by the interactive programs constituting the Transaction Handler's Interactive Environment.

The keyword to be used in the Grandpa configuration file for inserting an activity in this execution class is "SERV".

Foreground

The activities which must be executed quickly and without interruption are inserted in this class. This class has an high priority: The priorities which can be assigned to the programs belonging to this class are between 1000 and 1999. The CPU time slice assigned to this class's activities is infinite: when these activities are called they continue execution until they have finished.

The keyword to be used in the Grandpa configuration file for inserting an activity in this execution class is either "FG" or "PFG".

Standard

This class has a lower priority than Foreground: the priorities which can be assigned are between 2000 and 2999. The time slice assigned to the activities executed in this class is 100 msec. The activities which are commonly indicated as system services are inserted in this class, for example the spooling system.

The keyword to be used in the Grandpa configuration file for inserting an activity in this execution class is either "START" or "PSTART".

Background

This class has the lowest priority of the four execution classes. The priorities which can be assigned to activities executed in this class start from 3000, and the time slice is 100 msec. The activities which do not have to be executed particularly quickly are inserted in this class, for example the Shell environment batch activities, or the statistics on the system's activities, etc.

The activities placed in this class are executed only when there are no more executable activities (families with at least one ready process) belonging to the Server, Standard or Foreground classes.

The keyword to be used in the Grandpa configuration file for inserting an activity in this execution class is either "BG" or "PBG".

Observations

Correct distribution in the available classes of the various activities which the user wishes to be activated at system initialization is of utmost importance for a "balanced" functioning of the system itself. In other words, it is the user's responsibility to define all the programs to be activated by Grandpa and to place them in the correct execution class, in order to avoid a class being penalized by higher priority activities which monopolize the CPU. The Grandpa configuration file permits the user to correctly define and place these activities.

Interactive Programs

The interactive user programs, which are those associated with a work station, are always placed in the Standard execution class.

The keyword to be used in the Grandpa configuration file for associating an interactive program with a work station is "TTYx", where x indicates the work station in question.

The keyword to be used in the Grandpa configuration file for associating an interactive program to all the work stations handled is "ALLT".

The keyword "RTTY" it also available for OLILAN configurations. It enables one or more activities to be associated to the work stations, which can execute a "remote login" to the L1 MOS system. In this case the set of activities is common to all work stations of this type.

GRANDPA'S FUNCTIONS

As has already been said, both the interactive/non-interactive programs and the packages which are loaded in the user space can be activated automatically and with user-transparency by a system process known as Grandpa.

PROGRAMS ACTIVATION

When Grandpa activates the programs executed in the user space:

- it creates a family for each process activated, assigning each one a priority and time slice according to the specifications in its configuration file
- it loads a program from a file in the address space of the new family
- it creates a process to execute this program.

During this phase, Grandpa must provide the program to be activated with two types of information:

- Context parameters
- Activation parameters

Context Parameters

The information contained in this series of parameters guarantees the correct link between the objects necessary for the program, referred to by logical names, and the physical names of these objects by which they are referred to by the system.

This information is stored in a structure known as "execution context", which is created by the father program before activating the son program which will use it (in this case the father program is Grandpa). An

execution context exists for each program activation: the same program can be executed in different contexts (for example, when it is executed on different work stations). The information present in the context execution provided by Grandpa for each program is listed below in detail:

- **root** is the name of the root directory of all the systems present in the distributed configuration. It represents the highest point of the whole global file system, including all the local file systems of the connected systems.

- **localroot** is the name of the memory volume, created in memory when the operating system is loaded, present in the local system. It is the only non-removable volume in the system.

- **workdir** is the name of the working directory, which is defined as:

  . the directory specified by the user in the Grandpa configuration file

  . the localroot directory if the user has not indicated any directory in the Grandpa configuration file.

- **stdin** and **stdout** are parameters which are interpreted:

  . by the initialization, interactive and termination programs as names of the work stations to which they are connected. The work stations are seen as a sequential file (for input and output respectively)

  . by non-interactive programs as input and output byte-stream files. The name of these files is /DEV/BKG<n>, where <n> indicates the nth file created. In the first non-interactive program present in the Grandpa configuration file <n> has a value of 1, in the second 2, and so on. Eventual anomaly signals are recorded in the output file, which refer to the non-interactive program's execution. These files can be sequentially accessed. They are deleted during shutdown.

- **workst** is the name of the work station associated to the program. This value does not exist for non-interactive programs.

- **auxterm** is the identifier of the lowest window (25th line) of the work station, if splitting has been requested for it. This value does not exist if splitting has not been requested for the work station associated to the program.

- **sysprt1, ... , sysprt<n>** are the names of the n system printers available to the user.

Activation Parameters

The information contained in these parameters is provided by Grandpa for the program without carrying out any control.

The parameters must be specified in the format in which the program is

set up to receive them. The parameter list must not be longer than 160 characters.

The program activated by Grandpa can:

- acquire the activation parameters
- use all the files contained in the workdir directory in the execution context, as well as all the directories which this contains
- be ended at any moment, as it is Grandpa's responsibility to deactivate the work station to which the program is linked.
- communicate to Grandpa the system clousure.

To communicate the system clousure the program must give Grandpa one of the following codes:

. 128 : shutdown

. 129 : shutdown + restart (system-shutdown command with automatic restart)

. 130 : shutdown with timeout (command for system-shutdown after waiting for a given number of seconds).

To pass these values, and for code 130 to express also the "waiting time" before shutting down the system, the PASCAL+ procedure "stdenv.quit" must be used by inserting the following instructions in the program:

```
Import quit from stdenv;
procedure stdenv.quit (var msg : packed array [ 1..u ] of char;
                           cc : integer); definition;
```

The array "msg" contains the waiting time (passed as a positional parameter), while the variable "cc" contains one of the above codes. If "msg" is empty, wating time is defaults to 90 seconds. Remember that the shutdown procedure may only be requested by a program activated for a user belonging to the SYSTEM group. If the shutdown request is triggered from a non interactive program or from a program triggered by the keyword PINIT, after the system has closed down a "blinking" number "77" is displayed on the screen.

PACKAGE ACTIVATION

A package is a set of procedures which can be logically divided into:

- an initialization procedure
- a termination procedure
- a set of one or more procedures which provide the services offered by the package itself.

Activating a package means calling the initialization and termination procedures, as well as providing all the calls to the procedures, which guarantee the package's functions, for any program during the period of time between initializing and terminating the package.

Grandpa guarantees correct execution of these phases for the packages activated at system initialization time.

The keyword to be used in the Grandpa configuration file for activating a package is either "CALL" or "PCALL".

Each package is activated twice by Grandpa. The first, when the system is initialized, allows the package to initialize its own global variables. The second allows the package to correctly end its processing when system shutdown has been requested.

Grandpa provides a series of information for each package, using the Context and Initialization Parameters described below.

Context Parameters

The procedures which provide the services, as they can be called by any program, are executed in the Program Context associated to the requesting program. Their availability of files, work stations, printers, etc., therefore depends on which program is using them. The initialization and termination procedures of the package are executed in the Program Context of the program calling them, which is Grandpa.

Initialization Parameters

The information contained in these parameters is provided by Grandpa for the package initialization procedure.

The parameters must be specified in the format in which the program is set up to receive them. The parameter list must not be longer than 160 characters.

When the package termination procedure is activated the parameter string provided by Grandpa is empty.

An Example of the Grandpa Configuration File

An example is given of the Grandpa configuration file which is configured for a system with 4 work stations, one of which is the master, where activities can be carried out in diverse application environments, the system spooler is used and batch activities are executed.

The configuration file is the following:

```
CALL:/IPL/$QM/CODES/QUEMAN,<>PAR;
START:/IPL/SP/SPGPA;
BG:/IPL/BE/BATCHGPA;
TTYA:MASTER!DATE!NEWVOL!SHUTDOWN!MCL=/IPL/SYS/$VSH!
     BASIC=/IPL/DPC/CMD/BASIC;
TTYB:BASIC=/IPL/DPC/BASIC;
TTYC:ICE=/IPL/DPC/ICRTS!MCL=/IPL/SYS/$VSH;
TTYD:MCL=/IPL/SYS/$VSH!BASIC=/IPL/DPC/CMD/BASIC;
```

With this information in its configuration file, Grandpa carries out the following operations (listed in the order in which they are executed):

- It associates the following activities to the master (TTYA):
  - Definition of the date and time for the system clock.
  - Possible substitution of the system disk.
  - Possible system shutdown request.
  - Display of warning messages on the 25th line of the screen.
  - Shell application environment.
  - Interpreted BASIC application environment.
- It activates the default login program (which must contain the 'split' request for all the terminals, when a specific login program is not provided in the Grandpa configuration file for that TTYA, and when the TTYA must handle the display of warning messages if it has been associated the MASTER keyword).
- It loads the queue manager (giving it the PAR parameter list).
- It activates the spooling activity manager in the Standard execution class.
- It activates the batch job manager in the Background execution class.
- It associates the interpreted BASIC application environment to the terminal identified as TTYB.
- It associates the COBOL ICE and Shell application environments to the terminal identified as TTYC.
- It associates the Shell and interpreted BASIC application environments to the terminal identified as TTYD.

When the system is switched on, the user's login prompt will be displayed on all the terminals, if the login mechanism is present. A menu allowing the user to select the desired activity from those allowed for each terminal will be displayed on the screen of the multifunctional work stations (TTYA, TTYC and TTYD) after the user has logged in and started a work session.

When a multifunctional work station user has used one of the available application environments and decides to close it, Grandpa redisplays the select activity menu on the terminal, allowing the user to enter another (new or the same) application environment.

In order to free the terminal so that it can be used by other users, the user must select the LOGOUT activity. This is automatically inserted by Grandpa and available for all terminals.

When this activity has been selected, the user ends his work session on the system, and the user login prompt is redisplayed on the screen.

If the login mechanism is not present, the activities (or the select activity menu) associated to the terminals are automatically activated by Grandpa.

An application which is written by the user and inserted in Grandpa's configuration file is treated as any of the application environments used in this example.

INTERACTIVE ACTIVATION OF USER-WRITTEN PROGRAMS

As has already been said previously, each time a program is activated it receives the Context Parameters from the father family.

The user-written programs which have not been activated at system initialization time by Grandpa can be activated in the Shell environment.

Operating in this way, the user can specify the Context Parameters relating to the program which he intends to activate via appropriate tools offered by Shell.

For example, in order to activate a program and provide it with a set of files the user only has to position himself under the directory containing the command SETWDIR. The program will be able to access the files contained in this directory by referring to their partial path name. It will also be able to access any other file by referring to the complete path name (the first character of which is "/").

If the program uses support files which are referred to in the program via logical names (that is, known only to the program), these names must be connected to the physical names (via which the system identifies the files) using the CONN command.

The standard input and output can also be redirected (thus altering the values of the stdin and stdout parameters for execution of the program being activated), using the symbols < and > respectively. For example, a program (contained in the TEST file) which normally reads the keyboard

input data and transmits the output on the screen can be activated so that it reads the input data from one file (called IN) and writes the output in another file (called OUT), using the command:

```
TEST < IN > OUT
```

If the OUT file does not exist it is created; the output of the TEST program is written, starting from the beginning of this file. If the >> characters (TEST < IN >> OUT) are used instead of the > character, the program's output is written after the previous contents of the OUT file.

All or part of the Context Parameters so far described can be specified i a procedure written in the system's command language (MCL), and when this is called the program will be executed in the desired context.

The parameters which are not explicitly modified when the program is activated keep their value attributed by the father program of the environment in which it is activated. In this case they keep the values given to them by Grandpa activating the Shell environment.

The programs which are activated in Shell environment and whose execution is requested in batch mode (BM command) are placed in the Background execution class.

NOTES ON WRITING A USER PACKAGE

The user packages are sets of functions and/or procedures which must be available to any system user.

They are loaded and initialized by Grandpa when the system is switched on: this means that useful functionalities can be shared by all users, without having to link the code to each user program or reload the package each time it is to be executed.

A user package consists of a set of procedures which allow the package to be used, and two procedures for initialization and termination respectively.

There are no particular restrictions concerning the procedures, which are available to all the programs while the system is switched on, but there are some rules which the user must respect when preparing the package.

Firstly, a user package must be a PASCAL+ "module" or "monitor". This is because, although the PASCAL+ run-time support requires both the standard input and output files for executing a "program" ("stdin" and "stdout", which are normally associated to a work station and are part of the program's execution context - see the section "Program Activation"), a "module" or a "monitor" does not.

Furthermore, the initialization and termination procedures must be a single procedure of the module or monitor.

This procedure is called twice by Grandpa: the first time at system initialization when the list of parameters indicated by the user are passed to it, and secondly when system shutdown has been requested, and an empty string is passed as parameters.

The parameters are passed by Grandpa to the package initialization procedure via the "lineParam" variable, which must be declared in the procedure's heading as follows:

```
procedure init-end (var lineParam: packed array
                   [lineParamLow..lineParamHigh: integer] of char);
```

When the procedure is activated at system initialization time, the lineParam variable always contains a not-empty string ("lineParamLow" is less than or equal to "lineParamHigh"). The contents of this string are taken by Grandpa in its configuration file. The string must not be longer than 160 characters.

When the procedure is activated after a shutdown request, no parameters are passed ("lineParamLow" is greater than "lineParamHigh").

The initialization/termination procedure must, therefore:

- check the "lineParamLow" and "lineParamHigh" values to identify whether it has been activated for initialization or shutdown, and if the first case is true it must receive the lineParam variable's contents

- end its execution using the "Halt" PMM primitive in order to return control to Grandpa. Eventual parameters that the initialization/termination procedure wants to pass to Grandpa cannot exceed the length of the parameters passed by Grandpa to the procedure. In other words their length cannot be greater than (lineParamHigh - lineParamLow + 1). These parameters, however, are ignored by Grandpa.

This structure, however, means that the procedure cannot be called within a monitor (as the monitor's lock would not be released when the "Halt" primitive was executed, thus preventing subsequent access).

The call to the package's initialization procedure must therefore be contained in another procedure which is not a monitor procedure and is contained in a separate module, and whose name is the entry point which must be declared, when the package is linked, using the ZLOC linker's command ENTRY. This is the procedure which is activated twice by Grandpa (as described above) and it must terminate (in both cases) by calling the PMM's "Halt" primitive.

A symbol table must be generated for the other procedures in the package when this last is linked (using the linker's SYMBOL command).

This means that the package's procedures can be called by any program to which the symbol table has been linked to (via the INSYM command of the linkers).

Greater detail on the use of the linker and its commands is given in the manuals PASCAL+, Program Preparation and Execution and MOS, Program Development Tools, Reference Manual

A. DMPRINT UTILITY

This appendix describes the DMPRINT utility.

DMPRINT

This utility interprets, displays and prints the contents of a dump file. The dump file contains the image of all the user segments, and is generated after the execution of the "EnableDump" and "MemoryDump" primitives described in Chapter 4.

ACTIVATION OF DMPRINT

The DMPRINT utility is activated in the Shell environment by entering:

DMPRINT filename

---

where:

filename is the name of the dump file.

On activation the following question is displayed:

DO YOU WANT PRINT DUMP (Y/N)?

If **Y** is entered the following message is displayed:

ENTER PRINTER NAME :

and the user must enter the printer name. In this way the contents of the dump file are displayed and printed.

If **N** is entered the contents of the dump file are only displayed.

The following information is subsequently displayed:

- A header containing:
  - . the date (day/hour/minute) of the dump
  - . both the data segments and data/code segments used by the program.
- A segment descriptor for each segment containing:
  - . the segment number
  - . the segment type
  - . the segment length.
- The contents of the segment in both hexadecimal and in ASCII format. In ASCII format unprintable characters are replaced by the character "." .

The file is displayed in screen pages. When the screen is full the following question appears:

MORE(Y/N/Q)?

If **Y** is entered the next screen page within the same segment is displayed.

If **N** is entered the first screen page of the next segment is displayed.

If **Q** is entered the display and print of the dump file terminates.

The following message is displayed at the end of the file:

```
**    END OF FILE    **

DO YOU WANT TO RESTART (Y/N)?
```

N must be entered to exit from the utility, **Y** to restart the display of the file.

B. DISPJOUC COMMAND AND RECOVERY UTILITY

This appendix describes the Shell command DISPJOUC which displays the Joucman file and the RECOVERY utility which rebuilds files.

DISPJOUC

The **DISPJOUC** command displays the Joucman file which stores all messages from activities performed under Commit Manager control. (See Chapter 3).

DISPJOUC

This message is then displayed:

```
ENTER COMMIT OR MACHINE NAME: (DEFAULT IS LOCAL)
```

The user can enter:

xxxx/CR/	Where xxxx is the logical name of the Commit Manager (COMA, for example) or the physical name of the machine (NOM128, for example) where the Joucman file which is to be read resides.
/CR/	Refers to the Joucman file of the local system.

CONTENTS

The contents of the Joucman file are displayed in the following way:

```
ERR DATE HOUR ENV WS REPL ERTYPE ACTION INFORMATION
```

---

where:

ERR is a 3-digit code for the event or error that has occurred.

DATE is the day, month and year on which the event or error occurred.

HOUR is the time, in hours, minutes and seconds when the event or error occurred.

ENV is the environment in which the event or error occurred. The environments are the following:

- CMAN (Commit Manager)
- DUAL (Dual Log)
- INT (interactive environment)
- SEC (End Commit)
- WARM (Warm Restart)

WS is the work station where the program that caused the event or error was activated.

REPL is the error code returned by the operating system when an event or error occurs.

ERTYPE is the type of error. "NR" is a fatal error, "YR" is a recoverable error.

ACTION indicates l'anomalia verificatasi. Le anomalie possono essere le seguenti:

- ABORT (abort for a raison which is different with the following)
- APPLY ABRT (abort during the EndCommit)
- CMAN FAIL (the Commit Manager is failed)
- DUAL ABORT (abort during the writing of the Dual Log file)
- INT ABORT (abort in the interactive environment)
- MSG LOST (abort during the sending a message to client environment)
- SEC ABORT (abort during the EndCommit)
- WARM ABORT (abort during the Warm Restart phase)

INFORMATION is a comment string of the ERR field

ERROR MESSAGES STORED IN THE JOUCMAN FILE

This section gives the error codes (ERR field) and their description (INFORMATION field) which are stored in the Joucman file.

The following is given for each error:

- An explanation of the error.
- The error class: warning, recoverable and fatal.
- The system component (if any) that caused the error.
- The result of the error.
- The action that the user can take (if this is possible), to remove the cause of the error.

CMJ000 = 'COMMIT MONITOR NOW ACTIVE '

The Commit Manager has been activated properly.

Class : Warning.

Result : The Commit Manager is active and can be used.

Action : None

CMJ012 = 'ERROR IN CONN APPLICATION PROG '

The name of the interactive program is incorrect.

Class : Fatal.

File System error.

Result : The Commit Manager is active.

Action : Change the name of the application.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ013 = 'ERROR IN LOAD APPLICATION PROG '

An error has occurred when linking an application program (probably caused by segment conflict) or the file is damaged, and the program cannot be loaded.

Class : Fatal.

PMM error.

Result : The Commit Manager is active.

Action : Check the segments.

Check the MOS code field and see the part of the System Software Maintenance, User Guide manual (PMM section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software System Maintenance Service.

CMJ015 = 'ERROR IN GETKEYSTR PROG '

No name, or an incorrect name was given to the PROG parameter (which allows under the Commit Manager program-activation).

Class : Fatal.

Result : The Commit Manager is not active.

Action : Change the physical name given to the parameter PROG.

Check the MOS code field and see the part of the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ016 = 'ERROR IN GETKEYSTR WFIL '

The physical name given to the parameter WFIL (to activate the program under the Commit Manager) has been omitted or is wrong.

Class : Fatal.

Result : The Commit Manager is not active.

Action : Change the physical name given to the parameter WFIL.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ017 = 'ERROR IN CONNECT WFIL '

The physical name of the Transaction file is incorrect.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Change the name declared in the Grandpa configuration file.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ018 = 'ERROR IN CALL APPLICATION PROG '

A system error has occurred.

Class : Fatal.

PMM error.

Result : The Commit Manager is active.

Action : Check the MOS code field and see the System Software Maintenance, User Guide manual (PMM section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software Maintenance Service.

CMJ019 = 'ERROR IN GETKEYSTR CMAN '

The physical name given to the parameter CMAN for activating the program under Commit, has been omitted or is incorrect.

Class : Fatal.

Result : The Commit Manager is not active.

Action : Change the physical name given to the parameter CMAN.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ032 = 'CANNOT OPEN MAIN LOG FILE '

A system error has occurred.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ033 = 'CANNOT READ MAIN LOG FILE '

A system error has occurred.

Class : Fatal.

File System error.

Result : The Commit Manager is active, but the Log file cannot be accessed.

Action : Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ034 = 'CANNOT OPEN DUAL LOG FILE '

A system error has occurred.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ035 = 'CANNOT WRITE DUAL LOG FILE '

A system error has occurred.

Class : Fatal.

File System error.

Result : The Commit Manager has failed.

Action : Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ036 = 'ERR IN CONNECT SECURE FILE '

The physical name of the Log file is incorrect.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Change the name declared in the Grandpa configuration file.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ041 = 'ERR IN CONNECT SECURE FILE '

The physical name of the Log file is incorrect.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Change the name declared in the Grandpa configuration file.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ042 = 'ERR IN CREATE TOTAL FILE '

The Transaction file cannot be created.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the MOS code field and see the part of the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ043 = 'ERR IN CONNECT TOTAL FILE '

The physical name of the Transaction file is incorrect.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Change the name declared in the Grandpa configuration file.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ044 = 'ERR IN INSERT ALIAS CMAN NAME '

A system error has occurred. The same name has probably been assigned to two Commit Managers.

Class : Fatal.

Result : The Commit Manager is not active.

Active : Change the name identifying the Commit Manager.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ045 = 'ERR IN MAKE INSERT ALIAS '

A system error has occurred. The same name has probably been assigned to two Commit Managers.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Change the names of the Commit Managers.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software Maintenance Service.

CMJ049 = 'ERR IN OPENRCV OF SECURE PORT '

A system error has occurred.

Class : Fatal.

Port error.

Result : The Commit Manager is not active.

Action : Check the port configuration in the file $CON.

Check the MOS code field and see the System Software Maintenance, User Guide manual (Ports section) for the action to be taken when an error with this code occurs.

CMJ058 = 'ERR IN GET PARAMETERS '

A system error has occurred when getting parameters.

Class : Fatal.

Result : The Commit Manager is not active.

Action : Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software Maintenance Service.

CMJ059 = 'ERR IN CREATE SECURE FILE '

The Log file cannot be created.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ060 = 'ERROR IN SWITCH P_TABLE '

A System error has occurred.

Class : Fatal.

PMM error.

Result : The Commit Manager is not active.

Action : Check the MOS code field and see the System Software Maintenance, User Guide manual (PMM section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software Maintenance Service.

CMJ061 = 'ERR IN CREATE SERVER CHANNEL '

A system error has occurred.

Class : Fatal.

PMM error.

Result : The Commit Manager is not active.

Action : Check how many channels, families and processes have been configured in the $CON file.

Check the MOS code field and see the System Software Maintenance, User Guide manual (PMM section) for the action to be taken when an error with this code occurs.

CMJ062 = 'CANNOT CREATE DUAL LOG FILE '

The Dual Log file cannot be created.

Class : Recoverable.

File System error.

Result : The Commit Manager is active, but the Dual Log functions cannot be used.

Action : Make space on the disk by removing objects that are not used.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ063 = 'CANNOT CONNECT DUAL LOG FILE '

The physical name of the Dual Log file is incorrect.

Class : Recoverable.

File System error.

Result : The Commit Manager is active, but the Dual Log functions cannot be used.

Action : Change the name declared in the Grandpa configuration file.

Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ064 = 'CANNOT CONNECT DUALLOG MODULE '

The physical name of the Dual Log module is incorrect.

Class : Recoverable.

File System error.

Result : The Commit Manager is active, but the Dual Log functions cannot be used.

Action : Check the MOS code field and see the System Software Maintenance, User Guide manual (File System section) for the action to be taken when an error with this code occurs.

CMJ065 = 'CANNOT SPAWN DUALLOG MODULE '

Not enough families and processes have been configured.

Class : Recoverable.

PMM error.

Result : The Commit Manager is active, but the Dual Log functions cannot be used.

Azione : Change the configuration of families and processes in the $CON file.

Check the MOS code field and see the System Software Maintenance, User Guide manual (PMM section) for the action to be taken when an error with this code occurs.

CMJ066 = 'CANNOT LOAD DUALLOG MODULE '

A system error has occurred. There may not be enough memory to load the Dual Log file.

Class : Recoverable.

PMM error.

Result : The Commit Manager is active, but the Dual Log functions cannot be used.

Action : Check the segments.

Check the MOS code field and see the System Software Maintenance, User Guide manual (PMM section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software Maintenance Service.

CMJ067 = 'CANNOT START DUALLOG MODULE '

A system error has occurred.

Class : Recoverable.

PMM error.

Result : The Commit Manager is active, but the Dual Log functions cannot be used.

Action : Check the MOS code field and see the System Software Maintenance, User Guide manual (PMM section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software Maintenance Service.

CMJ100 = 'ERR IN RECEIVE MSG FROM INTER '

A system error has occurred, probably because of a timeout or an interruption on the line.

Class : Fatal.

Port error.

Result : The Commit Manager is active, but the customer environment cannot communicate whith the server.

Action : Check the state of the line.

Check the MOS code field and see the System Software Maintenance, User Guide manual (Ports section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software Maintenance Service.

CMJ101 = 'ERR IN SEND ANSWER TO INTER '

A system error has occurred, probably because of a timeout or an interruption in the line.

Class : Fatal.

Port error.

Result : The Commit Manager is active, but the server cannot communicate which the customer environment.

Action : Check the state of the line.

Check the MOS code field and see the System Software Maintenance, User Guide manual (Ports section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software Maintenance Service.

CMJ102 = 'ERR IN SEND ANSWER TO INTERACT '

A system error has occurred, probably because of a timeout or an interruption on the line.

Class : Fatal.

Port error.

Result : The Commit Manager is active, but the server cannot communicate which the customer environment.

Action : Check the state of the line.

Check the MOS code field and see the System Software Maintenance, User Guide manual (Ports section) for the action to be taken when an error with this code occurs.

Or call the Olivetti Software Maintenance Service.

CMJ150 = 'ERR IN OPEN B.I. FILE '

A system error has occurred.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ152 = 'ERR IN WRITE B.I. FILE '

A system error has occurred.

Class : Fatal.

File System error.

Result : The transaction cannot terminate correctly.

Action : Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ153 = 'ERR IN CLOSE B.I. FILE '

A system error has occurred.

Class : Fatal.

File System error.

Result : System performance is degraded and cannot operate correctly.

Action : Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ154 = 'ERR IN TRUNCATE SECURE FILE '

A system error has occurred.

Class : Recoverable.

File System error.

Result : The Commit Manager is active, but the transaction does not terminate correctly.

Action : Check the Log file.

Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ155 = 'ERR IN WRITE TOTAL FILE '

There is not enough space on disk, or the parameter CommitId of the primitive ENDCOMMIT is incorrect.

Class : Warning.

File System error.

Result : The Commit Manager is active, but the transaction does not take place.

Action : Recall the primitive ENDCOMMIT with the correct CommitId parameter, or make more space on disk.

Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ156 = 'ERR IN PREPROCESSING A.I. '

An error has occurred when files are being physically updated, after an ENDCOMMIT primitive.

Class : Fatal.

File System error.

Result : The transaction is not executed.

Action : Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ158 = 'ERR IN PREPROCESSING A.I. '

An error has occurred when physically updating files, after an ENDCOMMIT operation.

Class : Fatal.

File System error.

Result : The transaction is not executed.

Action : Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) try a cold restart, or call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ160 = 'ERR IN PREPROCESSING A.I. '

An error has occurred when physically updating files, after an ENDCOMMIT operation.

Class : Fatal.

File System error.

Result : The transaction is not executed.

Action : Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) try a cold restart, or call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ161 = 'ERR IN PREPROCESSING A.I. '

An error has occurred when physically updating files, after an ENDCOMMIT operation.

Class : Fatal.

File System error.

Result : The transaction is not executed.

Azione : Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) try a cold restart, or call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ165 = 'ERR IN READ HEADER SECURE FL '

An error has occurred when reading the Log file.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the Log file.

Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ166 = 'ERR IN READ BEFORE IMAGE '

An system error has occurred when reading the Log file.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the Log file.

Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ167 = 'ERR IN OPEN D.B. <filename> '

A file cannot be found during a warm restart.

Class : Fatal.

File System error.

Result : The Commit Manager is active, but the user files may be inconsistent.

Action : Check the state of the Data Base.

Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) try a cold restart, or call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ168 = 'B.I. DOES NOT FIT IN WORK BUF '

Work buffer is too small for the transaction.

Class : Warning.

Result : The ENDCOMMIT operation has not terminated properly.

Action : Divide the transaction into several parts, or increase the size of the "stage area".

CMJ170 = 'ERROR IN APPLICATION A/B.I '

An error has occurred when physically updating files, after an ENDCOMMIT operation.

Class : Fatal.

File System error.

Result : The transaction does not take place.

Action : Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 the value 103 (hardware error) try a cold restart, or call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ171 = 'ERROR IN APPLICATION A/B.I '

An error has occurred when physically updating files, after an ENDCOMMIT operation.

Class : Fatal.

File System error.

Result : The transaction does not take place.

Action : Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) try a cold restart, or call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ181 = 'ERROR IN APPLICATION A/B.I '

An error has occurred when physically updating files after an ENDCOMMIT operation.

Class : Fatal.

File System error.

Result : The transaction does not take place.

Action : Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) try a cold restart, or call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ182 = 'ERROR IN APPLICATION A/B.I '

An error has occurred when physically updating files after an ENDCOMMIT operation.

Class : Fatal.

File System file.

Result : The transaction does not take place.

Action : Check the state (and the consistency, if possible) of the user and Log files.

Check the MOS code field: if it contains the value 103 (hardware error) try a cold restart, or call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ207 = 'ERR IN READ FOOTER SECURE FL '

A system error has occurred when physically updating the Log file.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the Log file.

Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ208 = 'ERR IN READ HEADER SECURE FL '

A system error has occurred when reading the Log file.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the Log file.

Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ209 = 'ERR IN READ BEFORE IMAGE '

A system error has occurred when reading the Log file.

Class : Fatal.

File System error.

Result : The Commit Manager is not active.

Action : Check the Log file.

Check the MOS code field: if it contains the value 103 (hardware error) call the Olivetti Field Maintenance; if it contains 104 (software error), call the Olivetti Software Maintenance Service.

CMJ210 = 'WARM RESTART IN PROGRESS '

An error has occurred when closing the Commit Manager.

Class : Warning.

Result : The Commit Manager automatically carries out a warm restart, to resume work.

Action : None.

CMJ999 = 'COMMIT MONITOR ENDED '

The Commit Manager has been closed correctly.

Class : Warning.

Result : The Commit Manager has been closed correctly.

Action : None.

Note

In a distributed configuration, the utility DISPJOUC can be also be run on a system where the Commit Manager is not used.

**** ERROR IN CONNECT **** XXYY

Joucman file not connected.

**** ERROR IN OPEN **** XXYY

Joucman file not opened.

See the Message Book , Appendix A, for the meaning of XXYY

RECOVERY

This utility recovers one or all the files used in the Commit Manager transactions rebuilding them from the Log File.

RECOVERY

The parameters are requested interactively.

Note

1. To be able to rebuild files, the Log file must be written in append mode. (See Chapter 3).

2. If there are several Log files, this utility must be run once for each file. For each run, input will consist of the current Log file, plus the Image Copy of the file produced by the preceding run.

3. If the Dual Log file has been configured, it can be used instead of the Log file (See Chapter 3). This can be useful if there are hardware errors on the physical devices.

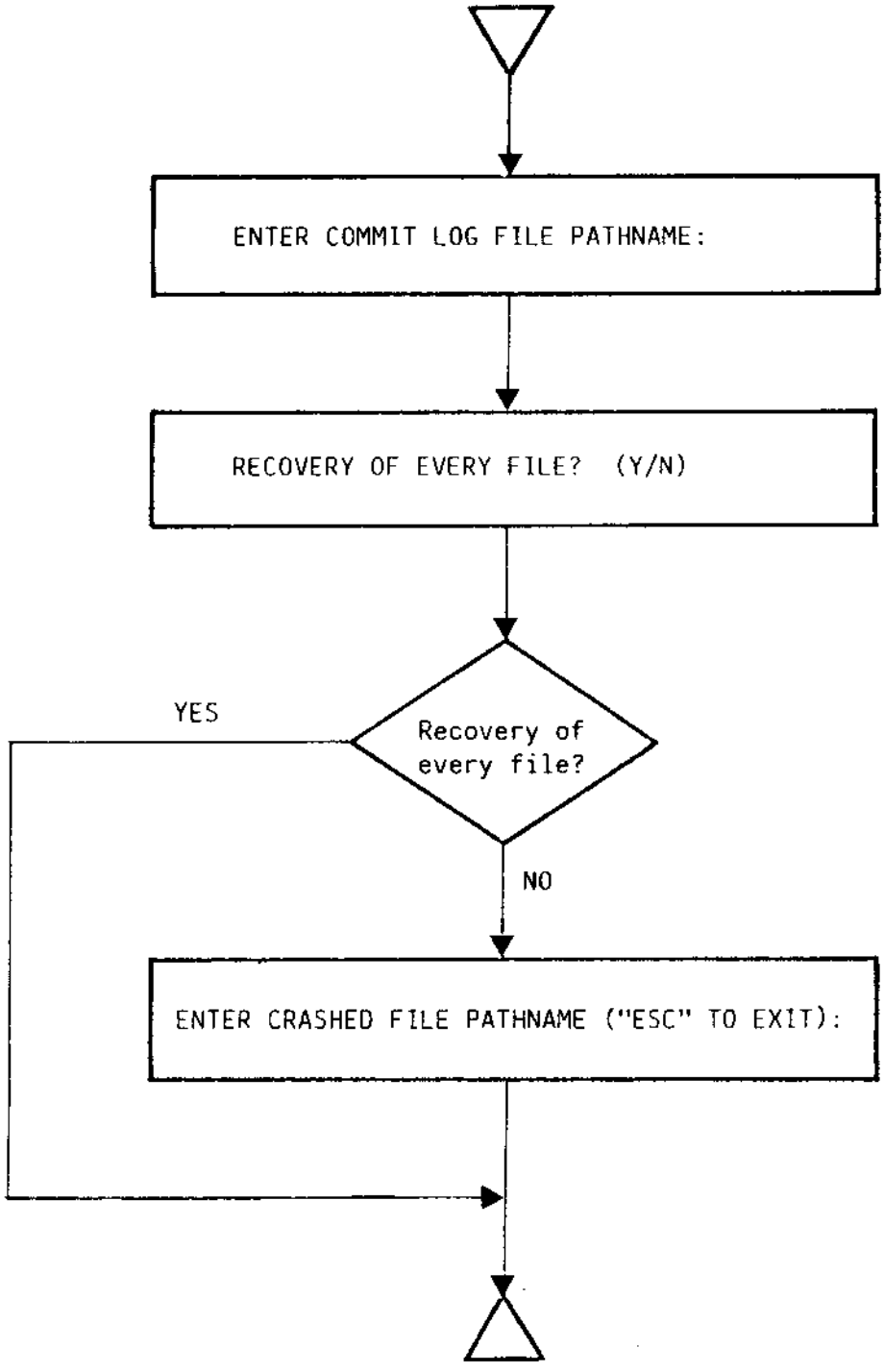


Fig. B-1 RECOVERY - Operating Interface

The following messages are displayed when the utility is activated:

****

UTILITY RECOVERY

****

ENTER COMMIT LOG FILE PATHNAME:

Type the whole of the Log file pathname used by the Commit Manager. This pathname must be the same as the one specified in the parameter LOGF, in the Grandpa configuration file.

INCORRECT PATHNAME

This message is displayed if the pathname of the Log file is incorrect. Such case causes an exit from the utility.

RECOVERY OF EVERY FILE ? (Y/N):

"Y" recovers all the files.

"N" indicates that not all files are to be recovered.

ENTER CRASHED FILE PATHNAME ("ESC" TO EXIT):

Type the full pathname of the damaged file that is to be recovered, or press "ESC" to leave the utility.

****

RECOVERY RUNNING

****

****

RECOVERY ENDED SUCCESSFULLY

****

The files have been recovered successfully.

****

RECOVERY FAILED ERROR_CODE=ZZ SYS_CODE=XXYY

****

Recovery has failed, because of errors ZZ and XXYY where:

- ZZ can have the following values and meanings:

  02, 03, 04 - error when opening the damaged file
  08 - error when opening the Log file
  09,10 - error when reading the Log file
  11 - error during recovery of damaged file

- XXYY is an error message emitted by the operating system (for its meaning, see Appendix A in the Message Book.

I. INDEX

U

V

W

Code 4002570 L (7)
Printed in Italy