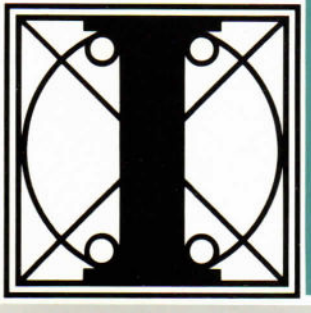
Operating System

MOS

Programmer Guide

MOS



olivetti

PUBLICATION ISSUED BY:

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

Operating System

MOS

Programmer Guide

olivetti

PREFACE

This manual is intended as a guide to users wishing to design and prepare applications to be executed on MOS systems, particularly applications (or parts of applications) written in Pascal+, which has complete visibility of the basic services that MOS offers applications.

SUMMARY

The manual begins with a general introduction; this is followed by a central section divided into three Parts, followed by four Appendices.

The general introduction (Chapter 1) lists the programming languages and the tools (services and programming structures) provided by MOS for the applications.

Part 1 is of particular interest to programmers, and includes:

- an overview of basic MOS services for processes and memory (Chapter 2), files (Chapter 3) and work stations (Chapter 4)

- a detailed description (Chapter 5) of a number of application services, most of which are available in the various programming languages.

Note that while Chapter 5 provides a full description of the services (including the call syntax), Chapters 2, 3 and 4 are only a basic guide to using the services.

Part 2 is a guide to integrating environments, as it:

- specifies the memory segments available to the applications according to the services used by the applications concerned (Chapter 6)

- specifies the possibilities, existing in multifunctional systems, relative to data exchange between different environments (Chapter 7)

- guides the user in the implementation of shared user services, besides those services supplied by Olivetti (Chapter 8).

Part 2 is of particular interest, therefore, to system administrators.

Part 3 comprises Chapters 9 and 10, and describes the execution of user activities.

Chapter 9 is of particular interest to users responsible for designing the activities of the system. It describes the various activation modes for user activities, with particular emphasis on automatic activation.

Chapter 10 specifies a number of programming facilities for the system start-up and shutdown phases.

The Appendices describe the following subjects:

- The format of an executable program (Appendix A) and the M5LDUMP utility, which prints it (Appendix B).
- The DMPRINT utility (Appendix C), which enables files containing application program memory dumps to be directed to the standard output.
- The types of screen formats available for the various environments (Appendix D).

REFERENCES

Read first:

Introduction to MOS (in MOS Features and Functions - Code 4041600 F)

For further information, read:

MOS Basic Architectural Concepts - Code 4002710 J

FORTRAN - Program Preparation and Execution (in FORTRAN Programming Manual - Code 4042040 G)

COBOL - Program Preparation and Examples (in COBOL Programming Manual - Code 4040470 V)

Compiled BASIC - Program Preparation (in Compiled BASIC - Programming Manual - Code 4041840 L)

PASCAL+ - Program Preparation (in PASCAL+ Programming Manual - Code 4041770 U)

C - Program Preparation and Execution (in C Programming Manual - Code 4041790 E)

ZLOC Linker - User Guide - Code 4043900 E

OLINK LInker - User Guide - Code 4042120 X

SHELL Commands - Reference Manual (in SHELL Environment - User Manual - Code 4041630 W)

MCL MOS Command Language - User Guide (in SHELL Environment - User Manual - Code 4041630 W)

MOS System Software Generation and Installation Manual - Code 4041710 W

MOS Full Work Station Emulator - User Guide - Code 01507901 T

MOS Operating Guide - Code 01001300 X

FILE SYSTEM Primitives - Reference Manual - Code 4004700 N

MOS Primitives for distributed Environment - Code 4004680 Q

PMM and DRIVER Primitives - Reference Manual - Code 4004670 F

PGU - Graphics Management Package - Programmer Guide - Code 4004650 D

RS232/CL Interface - Programmer Guide - Code 4004450 H

GSP Graphing Subroutine Package Application Interface - Programmer Guide - Code 4004320 V

VISA Form Management Package - Programmer Guide - Code 4041740 Z

LMS Local Management System Server Component - User Guide - Code 4041720 X

COMMIT Data Protection Service - User Guide - Code 4042060 A

MESSAGE SWITCHING Service - User Guide - Code 4042080 L

DISTRIBUTION: General (G)

FIRST EDITION: June 1988 - Release 6.0

CONTENTS

1. APPLICATIONS PROGRAMMING IN THE MOS ENVIRONMENT

This chapter provides a general overview of applications programming in the MOS environment.

After describing the many services that MOS offers the programmer, it outlines the characteristics of the programming structure (programs and services) that the user can implement in MOS.

MOS PROGRAMMING MULTI-FUNCTIONALITY

The MOS system is multi-functional: programming in MOS is thus characterized by the availability of a wide range of services for specific applications.

In order to offer the most suitable interface for particular applications problems, the following languages are available:

- Compiled COBOL
- Compiled BASIC
- ICE COBOL
- Interpreted BASIC
- Pascal+
- FORTRAN
- C

The programmer may prepare both interactive and batch applications in the Shell environment and execute them there or in such environments as:

- distributed
- business
- scientific
- data entry
- office automation.

As well as the services provided by the languages and the environments, the standard set of MOS services also comprises:

1. applications services (oriented to specialized applications)

2. basic services (for handling system resources)

3. services for handling user's activities.

Any user-written program can access the environment services and those listed above, subject to the visibility restrictions imposed by the language used. More precisely, each language has visibility of a particular set of service libraries (library being understood as a homogeneous subset of services).

Pascal+ has complete visibility of MOS services and may thus be used for the implementation of more specialized applications. By using Pascal+ the user can also prepare customized libraries for providing sets of functions which are not covered by standard libraries.

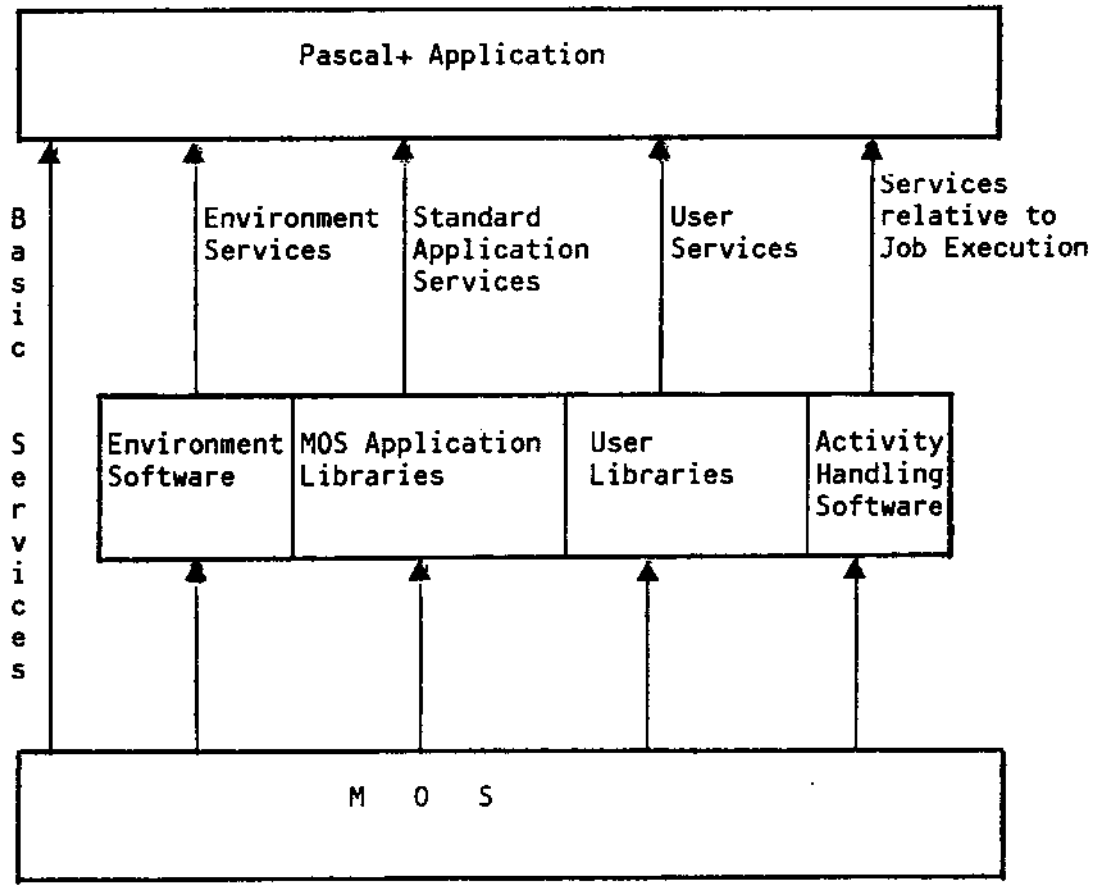


Fig. 1-1 Services Available to a Pascal+ Application

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

The DMS environment offers the programmer the Data Management Language (DML).

The remaining environment software either provides non-programmable interfaces (e.g terminal emulators), or interfaces whose description is outside the scope of this manual such as, for example, that for the ONE environment.

STANDARD APPLICATION SERVICES

The following components provide application services:

- PGU (Package Graphic Unified), which provides a set of facilities for the development of two-dimensional graphics. Its services may be called by compiled BASIC, interpreted BASIC, COBOL, DMS and Pascal+.

- GSP (Graphing Subroutine Package), which enables graphics to be designed. Its services may be called by compiled BASIC, interpreted BASIC, compiled COBOL, and Pascal+.

- VISA (Video Interface System Analizer), which handles work stations, using forms prepared by the user. VISA services can be called by compiled BASIC, interpreted BASIC, compiled COBOL, DMS and Pascal+.

- LMS (Local Monitoring System), which co-operates with the CMS (Central Monitoring System) for controlling the hardware resources of systems inter-connected in geographical networks. It provides services for keeping track of hardware and software failures, events and data, and can be called by compiled COBOL and Pascal+.

- The Commit Service, which safeguards data integrity. Its services may be called by compiled COBOL, Pascal+, and compiled and interpreted BASIC.

- Message Switching Services, which enables messages to be exchanged in a distributed system. These services are visible to the compiled languages COBOL and Pascal+.

- Various services, including signature verification, applications dump and dynamic program concatenation.

The various services are described in the Chapter "MOS Application Services"; descriptions of all the other services can be found in the appropriate manual.

The following applications interfaces to certain MOS components are also provided:

- RS232/CL services may be called by compiled BASIC, interpreted BASIC, and compiled COBOL, as well as FORTRAN, Pascal+
- Line Manager services may be called by compiled BASIC, COBOL, as well as Pascal+.

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

Component	Compiled BASIC (call to external procedure)	Inter-preted BASIC	FORTRAN (run-time)	Compiled 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
PAU (2)					X
Line Manager	X			X	X
VISA	X	X		X	X
MTS				X	X
BEAM	X			X	X
LMS				X	X
ONE				X	X
Message Switching				X	X
Commit Manager	X	X		X	X

(1) The functions available to interpreted BASIC are included in the language itself.
(2) The functions of this component are included in Pascal+.

The user should read the program preparation and execution manuals for the language used by the application for information on how to use the features offered by the libraries available to it. Titles and codes of the relevant languages are provided in the Preface.

Titles and codes of the manuals relative the MOS languages, and program preparation tools are listed in the Preface. Note that the BUILD and ZSTUB utilities are described in the OLINK on ZLOC manuals, respectively.

PROGRAM DEVELOPMENT SERVICES

These are provided by the following components:

- The Editor program, which enables programs in source format to be stored in a MOS system.
- Compilers for Pascal+, COBOL, BASIC and C, which output programs in object format.
- The ZLOC linker, which operates on programs in object format output by Pascal+ or C, and enables l-modules to be produced.
- The OLINK linker, which operates on programs in object format output by all compilers (except the C compiler), and produces a program directory.
- Interpreters of ICE COBOL and interpreted BASIC.
- Debugging tools for the various compiled languages.
- The BUILD utility, which enables the user to create service libraries.
- The ZSTUB utility, which makes it possible to prepare services that can be dynamically linked to a program.

SERVICES FOR HANDLING USER ACTIVITIES

These are provided by GrandPa, or components activated by GrandPa (for example, the Batch Monitor and the spooling system). They include the standard user environment preparation services, common to all the system's users, and those for the specific environment required by the user.

CONCEPTS AND STRUCTURES FUNDAMENTAL TO THE PROGRAMMER

This section provides general information of particular interest to the programmer.

APPLICATION STRUCTURE

As well as realizing independent activities, MOS enables co-operative activities to be created; that is, activities which together achieve a certain objective.

Processes (programs in execution) associated to different user activities "live" in different families; a family can be understood as an entity to which are allocated the resources available to the processes belonging to the same activity. The resources allocated to the family include, for example, the fundamental system resources - CPU and Memory.

As regards memory it should be noted that MOS offers the family a logical space of segmented type in which the programs that determine the activities of the family processes are loaded. Communication between the processes of the family takes place largely by means of Pascal+ monitors.

MOS also allows the families to co-operate between themselves; it permits, for example, one family to control one or more others, and one family to make its own services available to another family, or all the families. These possibilities are exploited, for example, by the environments that control the families hosting the user programs.

STATIC AND DYNAMIC OBJECTS

Like the "family", also the "object" concept is fundamental to MOS. Implementation of MOS as a distributed system is based on objects.

Each system resource, local or remote (e.g. a file), is an object which an application may access only if it knows its name. In MOS there are two types of object:

- static
- dynamic.

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 possibly an alias).

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

This means that the user does not have the problem of handling different 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 different 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).

NON-ACTIVE PROGRAMS: LOAD MODULES AND PROGRAM DIRECTORIES

A source program, object or l-module is a bytestream file belonging to a volume stored on a direct access support.

In order for an l-module to be activated, the volume on which it resides must be "mounted" on the system memory volume. For stand-alone systems, the FS enables a user to access only those objects (files) described in the relative volumes in memory. For distributed systems, the global space of the names recognized by the FS is contained in the name space of the individual memory volumes. What has been stated for l-modules, relative to the requirement to "mounting", applies also for program directories, described later.

LOAD MODULES

An l-module is the load unit in the memory of MOS systems. From the logical point of view, it is a collection of one or more code segments, one or more data segments, and a stack segment. Each l-module has a main entry point.

The load module contains information on each segment such as, for example, its position in the logical address space, and its length.

The format of a load module is described in Appendix A. To print a load module the M5LDUMP utility is available (see Appendix B).

A part of the logical address space of the program is available to the programmer; another part is occupied by the system, and by the environment and the application services.

PROGRAM DIRECTORIES

A program directory is a structure made up of a program (possibly organized in overlays) and by the data files used by that program.

It facilitates the portability of programs from one computer to another in the context of a distributed system, and enables memory use to be optimized.

Program Directory Structure

From the structural point of view it is a normal directory, as it is identified using a name chosen by the user, and groups together a set of files. What distinguishes it from other directories are the constraints imposed on the files contained in it, and the way in which it is created.

The name that the user associates to the program becomes the name of the directory; the files referenced by the program are indicated in the directory. These files include all the data files used, the main program (called MAIN) and the overlays, if any (for information on this aspect, refer to the "Overlays" Section, below.

The names of the files described in the program directory are those with which they are referred to in the program and, if wishing to retain program independence from its location within the tree structure of the file system, it must not reference files using a complete pathname.

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

The program directory is a static structure, unique for each program (even if activated a number of times). It is stored on disk as the relationships between the program and the files that it uses remain valid for the entire life of the program concerned.

Program Directory Creation

The OLINK linker produces an output program directory; ZLOC produces an l-module. In the latter case, 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).

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

Overlays

The program directory optimizes system use as programs with an "overlay" structure can be implemented. A program using overlays optimizes memory use, because parts of executable code that are stored in separate files, known as "overlays", can be grouped together. Another file, with the pre-defined name of MAIN, contains the code which controls the overlay's 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 pre-defined names. The user must guarantee their congruency with the names used in MAIN for calling them.

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

The program is structured, optionally, in overlays by the system automatically during the linking phase. The user wishing to divide his program into overlays must split the program code into separate files, compile these files individually, and link them, specifying the appropriate option when calling the OLINK linker. Refer to the OLINK Linker, User Guide for further details on overlay creation.

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.

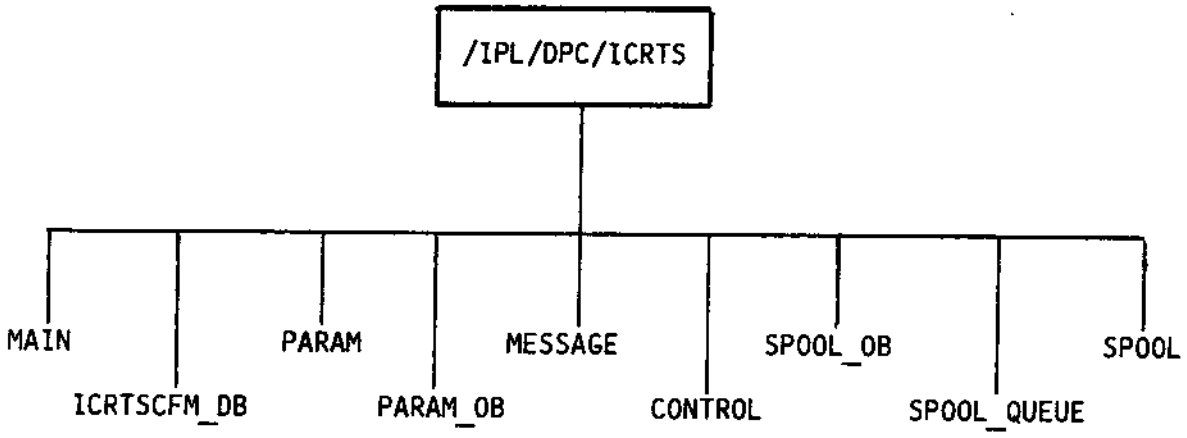


Fig. 1-2 COBOL ICE with Program Directory Structure

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.

ACTIVE PROGRAM STRUCTURE

An active program comprises two logically distinct parts:

- the static part, made up of the l-module and the program directory
- the dynamic part, made up of the program execution context.

The latter is a MOS dynamic object (structurally, a table) which is associated to each active program, and contains the dynamic global names (known as system-ids) of the family resources, including:

- the standard input unit
- the standard output unit
- the working directory
- the program directory (if any)
- the files connected before program activation (by means of the Shell CONN command).

If the same program is activated simultaneously by more than one user, a number of contexts will be provided, and a single program directory, which will be automatically linked to the relative context when loading the program into memory.

The context provides a mapping mechanism between the names of the resources used within the program (usually logical names) and the corresponding system-ids, thus making them usable by the program.

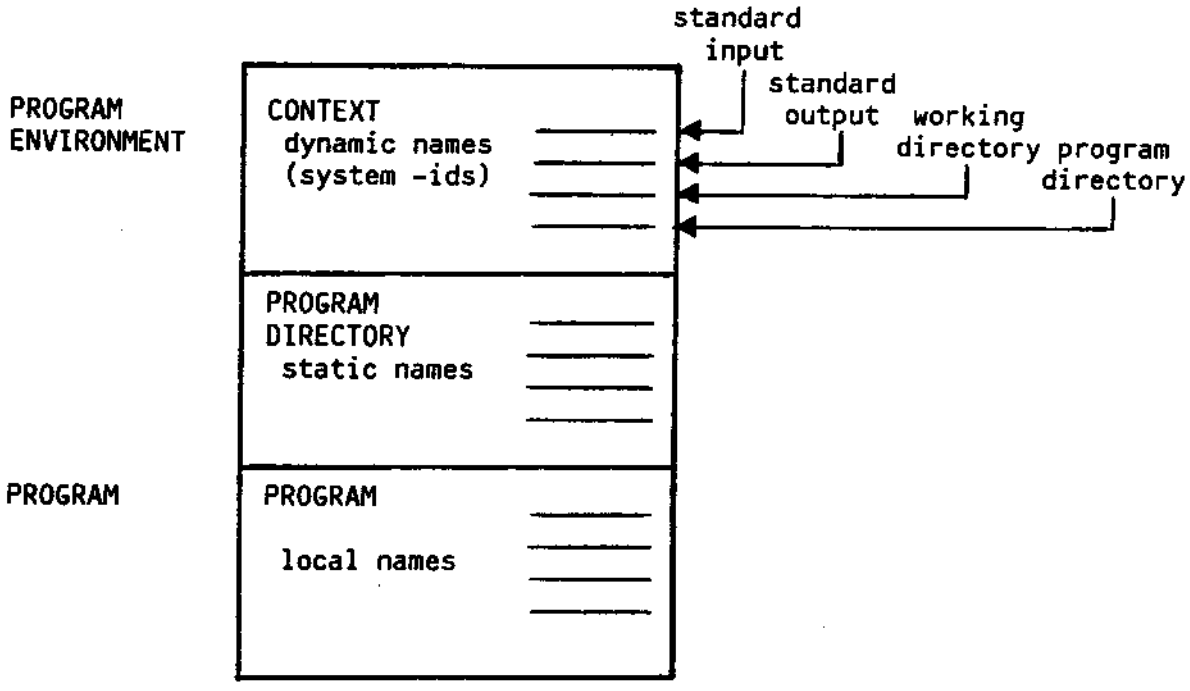


Fig. 1-3 Mapping Mechanism for System I/O Resource Names

USE OF STANDARD SERVICES

To simplify usage of standard services, Olivetti provides the Pascal+ programmer with files containing the definitions of the data types used by the services, and the declaration of the Pascal+ functions/procedures that implement them.

These files must be included in the source code of the program using the services.

Olivetti also provides the Pascal+ programmer with special structures known as "interfaces", which are to be linked to the object programs using the standard services. These structures derive their name from the fact that they normally do not contain the code and data of the called services, but various elements enabling the services to be called.

The major part of the standard services (among which are the basic MOS services and the environment services) are linked to the calling program dynamically; few are linked statically.

When a service is linked dynamically, the links between the calling program and the called services are definitively established only when the program is loaded in memory at run-time, and loading of the services into memory is handled by the system itself immediately after IPL (when they become available to all the users of the system). These characteristics free the user from having to repeat the linkage operation for programs using the services each time that Olivetti modifies a service.

When a service is linked dynamically, the links between calling program and called services are definitively established at the linkage phase; loading and unloading of services into/out of memory is closely connected with that of the program using them; these services are "private" to the program.

The static and dynamic linkage mechanisms are described in the Chapter "PMM Primitives and Related Commands"; the Chapter "Assignment of User Segments" contains information on the private and shared services.

Pascal+ programmers have visibility of two types of interface structures towards the standard services: interface files and (interface) libraries.

•When linking an interface file to an object program the entire contents of the file are included in the l-module output at the linkage phase.

When linking a library to an object program, only the elements of the library relative to the services called are included in the l-module output at the linkage phase.

ZLOC makes it possible to link to an object program both the libraries and interface files; OLINK only enables the libraries to be linked. Refer to the Manuals COBOL, Program Preparation and Examples, Compiled BASIC, Program Preparation, and C, Program Preparation and Execution for details on how a COBOL, Compiled BASIC or a C program calls the functions provided by a user library.

Refer also to the OLINK Linker, User Guide for details on how to link a Pascal+ user library to a COBOL or a compiled BASIC program.

Refer to the ZLOC Linker, User Guide for details on how to link a Pascal+ user library to a C program.

Note also that in MOS systems, the term "user package" describes all the software components providing shared services automatically loaded by MOS, including the software components supplied by Olivetti. This is due to the fact that also Olivetti user packages are "users" of the basic MOS functions.

CREATION AND USE OF USER SERVICES

The user may create both packages of services that can be shared by all users (user packages), and packages of private services; both types can be organized in libraries.

Preparation of user packages requires particular care, and is reserved to Pascal+ programmers as it implies the creation of not only the package but also the interface file. The ZSTUB utility enables the latter function to be effected.

The BUILD utility makes it possible to create user libraries containing:

- private user services

- interfaces towards user package services.

BUILD can in fact create collections of files in object format output by compilers or by the ZSTUB utility.

Information on the various steps involved in creating a user library is given in the Manual PASCAL+ Program Preparation. Refer also to the Chapter "Preparation of a User Package" in the current manual.

ZLOC makes it possible to call both interface files and libraries (output by the BUILD utility): in the former case the called services are linked dynamically, in the latter case they are linked statically or dynamically depending on the contents of each individual object obtained from the library. More precisely, those called objects that are the output of a compiler are linked statically; those output from ZSTUB are linked dynamically.

Both ZLOC and OLINK also make possible links between programs in object format and symbol tables output by a preceding link operation relative to another l-module. In this way it is possible to link statically to one program the services offered by another. These services thus take up belonging to the relative l-module.

PART 1 - PROGRAMMING

INTRODUCTION TO PART 1

This part describes the use of the interface primitives and structures provided by the following basic MOS components:

- File System Management
- Program and Memory Management
- Work Station Management

It also gives a complete description a number of application services most of which are available in more than one language.

2. FILE SYSTEM MANAGEMENT INTERFACES

After briefly summarizing the characteristics of the FS component, this chapter describes its interface objects and primitives. A list is then given of the Shell commands relative to the FS objects, together with a number of logical usage outlines for primitives.

The reader should note that the Section "FS Primitives: Logical Usage Outlines" may be ignored by system administrators, as it contains information of interest mainly to programmers.

A description of the Shell commands relative to FS objects can be found in the Manual SHELL Commands - Reference Manual; a description of primitives can be found in the File System Primitives Reference Manual. The programming environment in which they are used is described in the Manual PASCAL+ Program Preparation. A complete description of the relevant hardware I/O features can be found in M30/M40 Hardware - Architecture and Functioning.

INTRODUCTION

File System Management provides the user with the following services:

- It virtualizes the peripherals and provides the user with (user-named) collections of data objects (the files)
- It provides a name handling facility
- It provides a facility for controlling concurrent access to records and files

PERIPHERALS SUPPORTED AND FILE VISIBILITY

File System Management implements for the user the **file** concept, which is defined as a collection of data objects.

Two classes of peripherals are supported by FSM (which in both cases provide file visibility): direct access devices (disks of all types), and sequential access devices. To the latter class belong both those devices that allow record blocking and those that do not allow it (work stations, printers, and RS232 lines used in free-running mode).

The access classes are virtualized by access methods which are called the Direct Access and the Sequential Access Method, which provide the programmer with the access primitives to the files.

FSM also allows access to the files allocated in main memory, which is treated as a direct access support, capable of storing temporary files.

ACCESS ORGANIZATIONS AND METHODS

There are different kinds of files depending on the kind of data objects of which they are made up, and on the way the data objects are identified and operated on within the file.

The data objects may be organized into groups of records or streams of bytes. The records may all have the same length within a given file (fixed-length records) or they may have different lengths (variable-length records).

The Direct Access Method provides fixed-length records; the Sequential Access Method provides variable-length ones.

The files on disk or in main memory may be made up of simple sequences of bytes, or they can be structured in records. The other files (resident neither on disk nor in main memory) are made up of a succession of records (sequential organization). **The Direct Access Method makes it possible to access files supported by direct access devices.**

In this Access Method there are three ways (**modes**) of identifying and operating on the data objects.

In the **byte** mode the file is viewed as a sequence of bytes which are identified by their position (a non-negative integer). Any number of bytes can be read or written from a particular position in a single operation.

In the **positional** mode the file is viewed as a set of records of fixed length. The record is identified by its position in the file (a non-negative integer) and only one record can be read or written in a single operation.

In the **keyed** mode the file is again viewed as a set of records of fixed length, but the way of identifying the records is more extensive than in the positional mode. Each record is, however, identified by its **primary key**, i.e. the value of a user defined field of the record itself or an external name. The user can also declare the file as having additional (not necessarily unique) **secondary** keys corresponding to fields of the records. Thus, a more flexible content addressing scheme is provided for these files, since a written record can be retrieved directly by means of various keys. In other words, a keyed file is a positional file on which a different "view" of the record has been imposed.

The **Sequential Access Method** allows the user sequentially to write a variable-length record (possibly one byte only) and to read any number of bytes. This method is specifically designed for sequential devices, but its primitives can also be used for files on direct access devices.

Thus, although sequential organization is not available on disks, it is possible to access sequentially a file resident on disk; with no restrictions for bytestream files, but with certain limitations for positional or keyed files.

The table below shows the access modes allowed in the various types of file organization.

ACCESS MODE	FILE ORGANIZATION ON			
	DISKS			OTHER DEVICES
	byte	pos	keyed	seq
byte	yes	yes	yes	no
pos	no	yes	yes(*)	no
keyed	no	no	yes	no
seq	yes	yes(*)	yes(*)	yes

* with certain limitations

It should be noted that also the direct access method provides primitives for making sequential access to files possible. However, while sequential access carries out reading of data in the order in which it is physically written, direct access allows reading data according to the logical way in which it is stored.

It is also possible to gain an idea from the above diagram of the large degree of device-independence provided by FSM. This is not only internal to each device class, but applies to other classes as well. For example, if a printer has failed, a sequential file may be written on a direct access support instead of the printer, without changing the program while waiting for the file concerned to be subsequently printed. Further information on "device independence" can be found in the Section "I/O Primitives: Sequential Access".

NAME SPACE FACILITIES

The name space can be structured according to a hierarchy (a tree structure) as the user wishes, using the FS objects, i.e. files, directories and volumes.

File System Management provides a uniform name handling facility, which allows the user to assign names to FS objects and to access files by name.

Files and Directories

The "leaves" of this hierarchical name space are the user files, while the nodes of the "tree" are called **directories.** The user is given primitives for creating and naming objects within a specified directory, and the objects can be either files or directories. The File System distinguishes between the local name of an object (i.e. its name within the directory it belongs to) and the **pathname** (i.e. the sequence of local names extending from the root of the tree down to the named object).

The directory facilities thus allow the user to group his files together **logically.**

Volumes

The user is also allowed to group his files together physically by means of the concept of **volumes.** A volume is a file which physically contains other files (to be accessed via the directory mechanism). The root directory of the entire volume is known as the "Root Directory". As a direct access device can hold more than one physical volume, the File System volumes virtualize the physical (removable) volumes on direct access peripherals, i.e. storage areas that are physically contiguous. Sequential devices which are inherently contiguous do not need such a virtual layer and the sequential file is already a complete virtualization of them.

A physical volume space can be assigned entirely to the volume, or partitioned into logical volumes which helps in confining the various name space subtrees in predefined disk space areas, as well as physically grouping the files. The subdivision of a physical volume in a number of logical volumes may be useful for recovery reasons, for checking the handling of free zones on disks, etc.

The concept of the volume is very useful in systems where efficiency and reliability are important. Dumping a complete volume to another secondary storage unit is much more efficient and reliable than dumping the contents of the directories.

Memory Volumes

The **memory volume** is the highest point in the FS hierarchy. In this volume the memory is treated like a direct access peripheral. The memory volume contains the IPL volume (IPL), the **device directory** (DEV) and the **program contexts directory** (CONTEXT$$) for the system. It may also contain a directory (TMP) for the creation of temporary **memory files.**

Multi-user systems include, in the IPL volume, the USR directory, to which are connected as many files as there are system user login names.

The fact that also peripheral devices are considered files (having the pre-defined names TTY, HD, ...) facilitates uniformity, and thus simplifies programming, and also makes device independence possible.

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 to this memory volume before they can be referred to, using the MNT command or the Mount primitive.

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 logically connected to the memory volume).

Extension of Name Space and Path Names

The root of the global file system tree is a directory known as the **root** directory (or **root** for short). A unique file name must be specified in local or distributed systems, defining the complete route to be taken in order to reach the file, starting from the root of the FS (local) or distributed tree.

The global **pathname** of an object in the name space is a string of characters composed of the names of the directories passed through in order to reach the object from the global root directory. In order to separate the various component names, the "/" character is used. Thus, "/../mach1/IPL/usr/bin/refer/MARS" is the name of a file in a distributed system, reached by visiting the following objects:

- the directories ".." (this notation indicates the father directory of the current one; in this case the global root), and "mach1" (within the global root, which identifies the system to which the object belongs)
- the volume "IPL" (within "mach1")
- the directories "usr" within "mach1", "bin" within "usr" and "refer" within "bin"
- the file "MARS" within "refer".

The root of the subtree on each system is a directory called the **local root** directory and it is denoted by "/". The **local pathname** of an object is similar to the global pathname, but starts from the local root directory. Thus, the name "/IPL/usr/bin/refer/MARS" would be the local pathname of the file used for the above example.

There are three restrictions on the structure of the name space, as follows:

- The name space must be a single (although possibly huge) tree. Allowing more complex graphs would introduce unnecessary complications.
- The files resident in a particular system must form a complete subtree.
- The files contained in a removable volume must form a complete subtree.

CONCURRENT ACCESS CONTROL

FSM allows controlling concurrent access to the same file (i.e. access by more than one process), thus ensuring that the processes control competition for files and communicate smoothly via files.

FS PRIMITIVES

They are grouped according to their purposes as follows:

- Primitives for an entire object.
- I/O primitives for direct access.
- I/O primitives for sequential access.
- Concurrent access control primitives.
- Primitives for I/O options.
- Security primitives.
- Utility primitives.

PRIMITIVES FOR AN ENTIRE OBJECT

These may be further subdivided into primitives for:

- creating/destroying an object
- setting an object's characteristics
- connecting/disconnecting an object
- opening/closing an object

Creating and Destroying an Object

The **MakeVolume** primitive has been provided for the creation of volumes; this is described in the Section "Creating and Destroying Devices and Volumes", as are the other primitives relative to volumes.

A file or a directory is brought into existence by a Create primitive. A file, a Directory or a Volume can be renamed by the **Rename** primitive, and destroyed by the **Remove** primitive. When a process creates a file, it decides its external name (which remains valid up to subsequent Rename calls) and the kind of file it wants.

There are various create primitives, namely: **CreateByte, CreatePos, CreateKeyed, CreateDir, CreateSeq** and **CreateAlias.** The first three are for the creation of empty files (respectively: bytestream, positional and keyed); CreateDir creates an empty directory. All four operate on direct access devices.

The structures they create on these devices are described in the Section "FS Structures on Disk". CreateSeq creates a sequential file (operating on sequential access devices). It is described in the Section "Creating and Destroying Devices and Volumes". CreateAlias creates an "alias" file; that is, a file that contains the name of another file, called the "aliased" file.

When a keyed file is created, it has only a primary key definition. File System Management can be told to add a secondary key definition to the file by means of the **Attach** primitive. Any number of secondary key definitions can be attached. The **Detach** primitive allows the user to remove a secondary key definition, possibly in order to save the amount of disk space File System Management uses in handling secondary indices.

The name given to a file at create time can be changed by means of the **Rename** primitive. The services provided by **Rename** cover various cases that might occur when a file already exists with the new name, whether it is of the same or of a different type. Problems arise when the user wants to **Rename** a file across volumes. As already mentioned, volumes belong to the name space and their names are directory names, with the constraint that the whole subtree under the directory must be allocated within the volume. Thus, Renaming a file "under" a different volume requires copying (which is of course avoided if the file is renamed under the same volume).

Setting an Object's Characteristics

As regards object characteristics, the user can discover a file's **type** as recognized by File System Management by means of the **GetType** primitive, which returns a code for all possible types and subtypes of files in the system. For further information refer to the Section "FS Structures on Disk".

In the case of files on direct access supports, additional information attached to the file at create time (stored in PDD, as described in the Section "FS Structures on Disk") can subsequently be read by means of the **ReadInfo** primitives and can also be changed by means of the **WriteInfo** primitives. In this way File System Management permanently records file characteristics on behalf of the user; in this way also File System Management can be informed that the user process wants it to modify handling of the file.

Primitives are provided for each type of direct access file to read/write file characteristics: namely **WriteInfoByte, ReadInfoByte, WriteInfoPos, ReadInfoPos, WriteInfoKeyed, ReadInfoKeyed.** The one part that is common to all file type descriptions contains **allocationUnit, createTime, lastAccess, lastModify, userType.** The field **userType** contains an integer value which has no meaning to File System Management. The user process can use it to store a conventional use it envisages for the file (File System Management file types do not exhaust all possible file usages). The **userType** field helps in building user-defined file types.

At Create time the user process specifies an **allocationUnit,** i.e. the number of bytes the user wants File System Management to try and allocate each time some space is needed for the file. The user requirement for

the first allocation (initial file size) may be different from his subsequent requirements. Thus, the user can change the desired **allocationUnit** after the first allocation by using the WriteInfo primitives to write the new **allocationUnit** value. As regards the technique for the allocation of space, it should be noted that File System Management tries to provide the benefits of both contiguous storage allocation and dynamic file size.

With a little help from the user good results can be achieved.

When a file is first created, a contiguous space on disk is allocated for it. The user must suggest the size of such a space via the Create parameter **allocationUnit**, and File System Management will try to satisfy this hint. The size of the actually allocated **extent**, as close as possible to that of the **allocationUnit**, is returned to the user - during the file creation phase - via the parameter **allocatedSize** (so that the user realizes if the disk is becoming congested).

As a consequence of their variable size, files have **bounds** which change during the lifetime of the file. More precisely they have a **beginning-of-file** and an **end-of-file** bound (an ordering is always imposed on the objects in the file). By "decreasing" the end-of-file bound or by "increasing" the beginning-of-file bound, the file can be shrunk and this is achieved by means of the **SetBoundsByte, SetBoundsPos or SetBoundsKeyed** primitive. An extreme case of shrinking a file is that of emptying it. This is achieved by setting the end-of-file boundary before the beginning-of-file boundary.

For full details on how to empty a file, refer to the "I/O Primitives: Direct Access" and the "FS Primitives: Logical Usage Outlines" Sections. The size of the file is not, however, fixed for ever, and if the user asks for a larger size simply by writing outside the current **bounds** of the file, the necessary new extents - of the size requested by **allocationUnit** - are allocated for the file. Most probably these new extents will not be contiguous to the original extent. Similarly, if the file is shrunk (how this done is explained later) all the completely empty extents are freed and returned to the free disk space pool.

As might be expected, multiple extents have a cost, both because they require allocation and because they have to be located during file accesses. Thus, a **Compact** utility is available to compact files that may have become highly fragmented into as few extents as possible.

To summarize, good utilization in general is obtained at the price of a little cooperation from the user, who should:

- estimate the final size of the file and request this space at Create time as a single extent
- change **allocationUnit** to a relatively much smaller value (using the WriteInfo primitives described above)
- periodically compact the file.

If the user can foresee the number of records to be handled and their size, the information given in the Section "FS Structures on Disk" may be useful in estimating file sizes.

Creating and Destroying Devices and Volumes

With the direct access method, volumes and devices are distinct concepts.

As already stated, devices are given standard **filenames** at boot time so that they can be accessed by File System Management primitives.

It is the **CreateSeq** primitive that makes it possible to assign sequential devices a name in the file system name space (TTY, for example).

Sequential devices can then be accessed by means of the Sequential Access Method primitives; direct access devices can be accessed by means of the byte mode primitives of the Direct Access Method (see below).

Alternatively a volume must be created on a direct access device before it is possible to create a file on the device (if it is required to have files on the device). This is the normal way of using direct access devices.

Since there is no point in creating a direct access structure on a sequential device, File System Management does not recognize the concept of volume in this case. There exist, therefore, only standard input/output devices: terminal (ot_), printer (op_), free-running connection to another system (fr_) or magnetic tape (mt_).

The seqDev parameter of the CreateSeq primitive tells File System Management the device type, and the dev parameter is the system-id of the particular device, which is obtained using the **GetDevId** primitive. The **filename** becomes the File System name for that peripheral. Thus, **CreateSeq** is meant to be used by programs that define the operating environment.

A volume is created by means of the **MakeVolume** primitive, which is told in which parent volume to make the new volume. When the user wants to make a volume out of a (removable) disk volume (that is, the user requires the new volume to occupy an entire device), it specifies as parent volume the name of the device where the physical volume resides.

The only non-removable volume of a system is the **memory volume** created at boot time.

The **MakeVolume** primitive is meant to be used only by programs that define the operating environment for other programs.

When a volume is created, File System Management will always create on the support the structures which are necessary for storage management and for holding the directory which is the root of the subtree allocated on the data structures. The root directory may be given a name at **MakeVolume** time which acts as the label of the physical volume. This name could be provided to the **Mount** primitive when the volume is logically mounted (in order to check it if required with the one stored on the medium).

As already explained, the "Mount" primitive appends the sub-tree contained in the volume to be mounted onto a directory of the memory volume. The **Unmount** primitive removes it.

The purpose of unmounting a physical volume will generally be to allow physical removal of the disk volume.

Connecting and Disconnecting an Object

Handling of file names is very convenient for users (who might want to give names to files to help in remembering what kind of information they contain). However, if these names were to be used for each read or write operation on the file the overhead would be very high because such a naming scheme involves tree traversing (and thus additional I/O operations). Therefore, the File System requires the user process to first **Connect** to the existing file to be read or written - even if it is empty - by means of its name.

The **Connect** primitive returns to the process a system name for the file, which is to be used for subsequent operations on the file. Such a system name is called the **system-id** of the file, and its value is of no concern to the user process, as it need only store it in a variable.

In addition, the tree traversing from the root according to the path name might be difficult or impossible for the process. Thus, names are always expressed as **pathnames** relative to a directory which is specified by means of a system-id. In this way the directory does not have to be the root directory (whose id is obtained via the program context) and any known and convenient directory can be used as the starting point from which the tree is traversed. Thus, in MOS the files are always named by means of the pair:

---

directory system-id, relative pathname

It should be noted that:

- The **Connect** primitive effects connection to any system object and thus to the program context, the volume and the directory, as well as to the file.

- Connection to an object of a lower hierarchic level requires knowing the system-id of an object of a higher order.

With the above in mind, connection to the file /AAA/BBB/CCC/DDD belonging to a volume already connected may be achieved in the following way:

- Connection to the context of the program in order to obtain the system-id of the local root.

- Connection to the directory AAA/BBB/CCC, by means of the system-id of the local root.

- Connection to the file DDD by means of the system-id of the above directory.

An example of creation of a byte stream file can be found in the Section "FS Primitives: Logical Usage Outlines".

When using the **Connect** primitive it is necessary to specify the **access rights** the user process requires to have on that file. The access rights are **bound** to the returned system id and the user process will not be allowed to perform any operation which is not included in the access rights of the system-id.

Access rights specified by the user at the time of the Connect are checked with those stored on the file in order to guarantee data security. Further details can be found in the Section "Security Primitives", in which the data security mechanism provided by MOS is briefly described.

The access rights are in force until the Connect session is terminated by means of the **Disconnect** primitive. Thus, the Connect session allows the programmer to structure his programs conveniently by separating the procedures that determine the access rights from those that actually operate on the files. The former are concerned with the actual pathnames, while the latter receive the system-ids they have to operate upon without needing to know file names.

When the object to be connected is a file system object, namely a file, a directory or a volume, the available access rights are **nil_**; **read (r_)**, **write (w_)**, **execute (x_)** or **append (a_)**; or certain combinations of these except nil_. The **append** access right is the only one requiring explanation. A file whose system-id carries the a_ access right can only be written outside the file boundaries (because, as will be explained later, File System Management will automatically enlarge the file). This access right prevents the file being accidentally overwritten. Only new information can be added to the file.

The access rights the process specifies at **Connect** time can be seen as a kind of contract the process stipulates with itself about the future use of the returned system id (connect-id). Should the process violate the contract by issuing a file operation which is not implied by the specified access rights, File System Management will return a security violation error code. The converse is not true, i.e. file operations implied by the access rights are not guaranteed always to be valid, because hardware limitations would make their execution impossible. For example, one could **Connect** to a printer with read-only access rights but an invalid operation error code would be obtained if any attempt were made to read from the printer system-id.

On a connect-id it is possible to perform certain generic "information" requests, such as getting the time of creation, length of file and so on.

Connect merely performs a name mapping and ensures that the caller program may perform this mapping. Thus even if a file is locked exclusively (see the Section "Primitives for Concurrent File Access") this will not prevent another user from connecting to the file.

If it is required to design programs that are "portable" from one system to another, it is necessary to use a program directory that is connected when the program is activated by a software environment (for example, the Shell interpreter).

When a **Connect** primitive is performed on an alias, the **Connect** primitive returns the process a system name for the aliased file. For all other primitives handling pathnames, the alias mechanism only functions if the alias is an intermediate directory in the pathname. If the alias name is the last element in the pathname, the primitive works on the alias itself, and not on the aliased file.

In order to obtain a system-id for an alias, the **ConnectLiteral** primitive must be used, as using the **Connect** primitive on an alias would cause the system-id to be associated with the aliased file. A system-id can be obtained for the parent directory of a file by means of the **ConnectFather** primitive. A copy of a system-id may be obtained through the **ConnectAgain** primitive.

Opening and Closing an Object

The Connect sessions are themselves structured as sessions (use sessions) initiated by an **Open** primitive and terminated by a **Close** primitive. The use sessions are distinguished from the Connect sessions because they cover other aspects of file usage (namely, concurrent file usage). An additional good reason to keep the Connect session concept distinct from the use session concept is that the Connect session is good accessing policy and is thus recommended in distributed systems.

The use session belongs more properly to the file operation domain and as such it will be described in later sections. Certain points are worth noting here. A use session (initiated by an **OpenByte, OpenPos, OpenKeyed or OpenSeq** primitive and terminated by a **CloseByte, ClosePos, CloseKeyed or CloseSeq** primitive) is characterized by an **access mode** and a **lock level.**

The access mode requested by the user must be a subset of the access rights specified at **Connect** time. Checking of the access mode provides early signalling that the program is trying to use the file incorrectly.

The requested lock level specifies the degree of concurrent file use for the session and the **timeout** parameter input to the **Open** primitives tells the File System how long the user is prepared to wait to have the Open operation on that file completed. Thus, if the desired lock level cannot be obtained within **timeout,** the Open activation terminates with a TIMEOUT error (concurrent access facilities will be discussed below, but obviously it is impossible to grant an exclusive lock to more than one process at a time). The file lock level is granted for the whole session and to change it the file must be closed and re-opened, thus inducing a program structure which is consistent with the file usage.

The Open-type primitives require a connect-id in input and returns an open-id. Every open-id is different, even when the open is being performed on an already opened file. An open-id is valid until a corresponding Close primitive is requested. The Close also returns the value of the associated connect-id.

I/O PRIMITIVES: DIRECT ACCESS

This section describes the primitives for reading/writing data (according to the various modes provided in direct access), and those primitives closely allied to them. To the latter group belong those primitives that make it possible to achieve a logical sequential read **(scan)** of files organized in records. This type of read is different from that obtained using sequential access, which offers the data just as it was stored in the file, regardless of its logical order (determined by the value of the position or key) or any logical deletion of records. The scan is achieved using:

- The primitives **BeginByte, BeginPos, BeginKeyed, EndByte, EndPos and EndKeyed** which provide the user with both the position and the **address** of the bounds of the file.

- The Compute class primitives **(ComputeByte, ComputePos, ComputeKeyed),** which allow the user to scan the file because it returns both position and address of a valid record which is some number of valid records ahead of or behind a specified record (provided by Begin or End primitive class).

- The Transform class primitives carry out analogous functions to those of the Compute class, as described below.

Compute is also used to obtain an address that is meaningless except when it is beyond one of the current bounds of the file. By setting the other bound to this address the file may be completely emptied. This is the only Compute use meaningful in the case of byte stream organization.

Byte Stream Type I/O Primitives

The byte mode has been designed to provide byte-level access to the data in the file. A variable number of bytes can be transferred in a single I/O operation via the **ReadByte, WriteByte** or **Append Byte** primitives.

The byte being accessed is identified by its **position** (a non-negative integer) in a conceptually infinite stream of bytes. This (infinite) byte stream is therefore not all allocated on the disk. Only a number of extents - sufficient to encompass the stream from the beginning-of-file bound to the end-of-file bound - must be kept on disk (and note that immediately after creation the first byte is in position zero). Moreover, when dealing with bytes there is no point in stating whether a byte between the bounds exists or not. Thus, all bytes included within the byte mode bounds are **valid,** even if they have never been written.

Since all byte positions within the bounds are considered by the FSM to be valid, writing a byte within the bounds is always taken as

overwriting. When the **WriteByte** primitive is used, the user must specify the parameter **WriteMode.** This can be rewrite_, newwrite_ or alwayswrite_. The **WriteMode** rewrite_ only allows writing inside the current file bounds and the file must have been opened with **AccessMode** including w_. The **WriteMode** newwrite_ can be used only with **AccessMode** including w_ or a_ or both, and only to write outside the current file bounds.

To achieve the latter function the **AppendByte** primitive is also available. This differs from WriteByte (newwrite_) only in that it guarantees atomicity of the writing operation. It thus makes it possible for the user to avoid using the file lock (see the Section "Primitives for Concurrent File Access".)

The **WriteMode** alwayswrite_ does not constrain the write action in any way, so the only limitations are those imposed by the access mode. As explained earlier, the **ComputeByte** primitive (used in conjunction with BeginByte or EndByte) is only needed in order to shrink or empty the file, since File System Management addresses do not help the user in accessing his bytes.

Positional Mode I/O Primitives

The file is seen instead as a sequence of records. One record only can be transferred in an I/O operation. Unlike byte stream organization, positional makes it possible to distinguish between whether a record exists in the file (because it has been previously written) or not (because it has never been written or has been deleted from the file). In the positional mode, the concept of **valid** or existing record is meaningful.

The positional mode provides a simple means of access: the records are identified by their **position** (a non-negative integer) just as the bytes are in the byte mode. The space on disk for all the records between the file bounds is allocated (just as for the byte mode) but FSM keeps track of which records are valid, so that the user can **scan** the file (for how this is done, see below), reading only valid records.

There are many situations in which the positional record concept is useful but the user does not require help from File System Management to check whether a record exists. This might happen if the records are never deleted once written and the last record is identified in an application dependent way. The FS can offer a more efficient implementation of the positional mode if record deletion support is not required. This option is specified by the record deletion parameter input to **CreatePos.**

When this parameter is nd_ then use of the **DeletePos** primitive will not be allowed and, since all records will be taken as valid, all writes inside the bounds will be counted as overwrites. Thus, the observations made about **WriteMode** and **AccessMode** in the case of byte mode apply here as well. When record deletion is handled, however, the value of newwrite_ for the **WriteMode** parameter input to **WritePos** does not necessarily imply that writing will be outside the file bounds. In this case, the **AccessMode** with which the file has been opened determines the feasibility of the operation.

To achieve the function of Append the **AppendPos** primitive is also available. This differs from WriteByte (newwrite_) only in that it guarantees atomicity of the writing operation. It thus makes it possible for the user to avoid using the file lock (see the Section "Primitives for Simultaneous File Access".)

As far as the storage allocation is concerned, there are many situations in which a positional file is very **sparse** , i.e. there are a lot of non-existent records within the file bounds. In those cases the storage allocation policy adopted for positional files is not the best possible. Adopting the keyed file policy, which does not suffer from the same problems, the user would still incur the overhead of a general purpose interface which is designed for variable-length keys. When a record is deleted, it is marked as non-existent and the space is recovered when the **Compact** primitive is executed.

As far as **scanning** a positional file is concerned, a few considerations can be added. First, if record deletion is not handled, then **ComputePos** serves only for emptying the file (just as for the byte mode). When record deletion is handled, the description of file scanning given in the Section "I/O Primitives : Direct Access" applies.

ComputePos permits jumping from an existing record to another existing one. The problem might arise when locating an existing record which is "close" to a given position. The **TransformPos** primitive efficiently solves the problem by providing the position and address of such an existing record. The record is selected according to the **exactness** parameter input to **TransformPos.** The **exactness** parameter can be ge_ (greater than or equal to), eq_ (equal to), gt_ (greater than), next_ or previous_. The meaning is obvious but notice that a ge_ is performed before taking the next or previous record (for next_ and previous_).

Keyed Mode I/O Primitives

When a keyed file is created by **CreateKeyed** the primary key of the file must be specified by the **primaryCoords** parameter. The parameter says where the key is located in the file record and how many bytes long it is. If the specified key starting position is negative, then the key is considered to be completely external to the record. The primary key can be seen as the "name" of each record of the file. Thus, there can be only one primary key definition for a file, the keys cannot be duplicated (names must be unique) and the key cannot be changed when updating a record (update is an operation that should only change the contents of a record, not its name). The primary key concept is a generalization of the position concept. Since the key is chosen by the user, File System Management cannot use it directly to locate the record within the allocated extent. Auxiliary data structures (called indexes) are used to do this.

The keyed mode provides a powerful content addressing scheme by means of the primary and secondary keys (described in detail in the next paragraph). Since any field of the records can be used to access them, as well as the external "names", different orderings of the file records are available depending on which key is used.

The keyed file provides a complex view of the record. Consequently, even if the file has two physical bounds, different bounds can be seen using different keys because of the different ordering imposed by the different key definitions. Analogously, the scanning of the file using different sets of keys provides the records in a different order (and also provides different records because of duplicate keys).

When the keyed file is created, the records can only be seen "through" the primary key, in the sense that its value must be provided as a parameter to all record accessing primitives: the record **view** consists of the primary key definition only. The user may wish to be able to see the records through additional fields of the record as well. **Secondary** key definitions can be added or removed by means of the **Attach and Detach** primitives (the complete record view can be obtained by the primitive **GetRecordView).** If secondary keys are present in the record view, **WriteKeyed** and **DeleteKeyed** (which always need the value of the primary key) have more work to do because they must update the indices of the secondary keys. Since the secondary keys are part of the content of the record, their values can be changed when updating the file record but they can only be internal to the record. Secondary keys can be duplicated, provided they have been so declared at **Attach** time. The different sets of keys are identified by a **fieldNumber** returned by the **Attach** primitive (the set of primary keys has number zero).

The method described above for scanning a positional file applies for keyed files as well. Of course, a file is scanned using one of the keys as access specification. The **ComputeKeyed** primitive starts from a pair **fieldNumber and presentAddress** and returns the address and the key of the record that is displaced **nrKeys** away from the specified one. The **presentAddress** could have been obtained from another **ComputeKeyed** call or from an **EndKeyed** or **BeginKeyed** call. Note that the address values (which should not be interpreted by the user in any way) are key-dependent, i.e. if an address has been obtained by using a certain key definition it cannot be used in conjunction with another key definition (or the result will be meaningless).

The order in which the records are obtained from **ComputeKeyed** is dictated by the keys. The keys are ordered byte by byte and the records are ordered in key ascending order. Since **ComputeKeyed** returns also the position of the next record, the user can read the record using the **ReadPos** primitive rather than the **ReadKeyed** one, thus saving some time. When a duplicated key definition is used to scan the file, **ComputeKeyed** will return only one record for each key. The user is thus assured of always obtaining records with different keys.

The user might, however, prefer to scan the file completely, obtaining all records with the same key, even if he cannot determine the order in which the records with the same key are returned (they are returned in arrival order). **TransformKeyed** can be used (among other purposes) for this task. In fact, the inputs to **TransformKeyed** are a **fieldNumber,** a **searchKey** and a **searchPos,** and it returns the key, the position and the address of a new record. As for the positional case, an **exactness** parameter specifies the criterion to select the record sought. The important point is that File System Management can receive suggestions (by the **searchPos** parameter) as to where to start the search from.

Thus the next record can always be found if the record with the same key already returned is skipped by providing its position as **searchPos.**

TransformKeyed is also to be used for searching for records with partial key values. When the size of **searchKey** is less than the actual record key size, then searching is performed assuming that the actual key is truncated to the **searchKey** length.

I/O PRIMITIVES : SEQUENTIAL ACCESS

This section describes the primitives **ReadSeq, WriteSeq and TapeIo.**

As previously mentioned, the sequential access primitives provide variable record length.

Unblocked Sequential Access Devices

ReadSeq and WriteSeq make it possible to access unblocked sequential devices i.e. printers, terminals and free running communications lines. These primitives, (as well as OpenSeq and CloseSeq) are device independent, and the same operation may be performed on all sequential files with the same results (except for hardware limitations). If the user wants to perform a **device dependent** operation on a terminal or a printer, then the Work Station Manager should be used. The resulting programs are consequently less portable and flexible, e.g. their output cannot be directed to another device.

It is the sequential device itself that determines what a record is (e.g. a line, a card and so on). **WriteSeq** is not concerned with the organization of the data to be written and just appends the requested bytes to the file. Analogously, **ReadSeq** reads the requested number of bytes from the file and informs the user of the position of the first "linefeed", if any. If the file is the terminal, the record is considered to be the whole line. For example, if the line typed consists of 50 characters of data closed by a carriage return, then after **ReadSeq** the **lengthRead** output parameter will specify 51 and a "linefeed" character will be in the buffer after the data.

Blocked Sequential Access Devices

The **TapeIo** primitive makes it possible to access blocked sequential devices, i.e. magnetic tapes. Depending on the value of the parameters **ReadOpt** and **DataSize** it:

- Reads a block of data from a physical support into a private buffer defined by the user
- Transfers a block of data from a private buffer to a user area
- Writes a block of data from a private buffer onto a physical support
- Transfers a block of data from a user area to a private buffer.

Full information on private buffers can be found in the Section "Primitives for I/O Options and Buffers".

The Tapelo primitive maintains visibility of the block of data on the support; in this respect it differs completely from bytestream mode primitives.

It is the responsibility of the user, before opening the device, to set up a private buffer pool consisting of a single buffer (see the Section "Primitives for I/O Options and Buffers") and to define the length of this buffer (carrying out these operations using the SetBufferOption). Note that the length requested by the user will be rounded up to the nearest multiple of 512 bytes.

It is up to the user to handle the records existing in the block. It is thus the user who must request a physical operation on a block only when he has processed the last record of the preceding block. Some examples of using Tapelo are given in the Section "FS Primitives: Logical Usage Outlines".

The primitives OpenSeq and CloseSeq, if referring to magnetic tape devices, automatically effect the tape rewind.

If wishing to effect an append operation it is necessary to operate in one of the following ways:

- read the file up to the End of File (this event is signalled by Tapelo itself) and then write on it starting from that point

- read the file up to the last block (when the last block is incomplete this condition is signalled by Tapelo itself); complete that block, write it to tape and then proceed with construction and writing of the following blocks.

Given the physical characteristics of the tape device, it is necessary to lock the file when opening a session of use with magnetic tapes.

Direct Access Devices

The ReadSeq and WriteSeq primitives can also be used to read from or write to byte files on disk sequentially. When a byte file is opened in sequential mode using the **OpenSeq** primitive, a single current position is maintained on a use session basis, i.e. for each user separately. The user can find out the value of the current position using the **SenseP** primitive and can set the current position using the **SeekP** primitive.

The ReadSeq can also be used to read sequentially the record of a positional or keyed file. In fact, as explained in the Section "FS Structures on Disk", the data part of a keyed file is a byte stream file. The records are read in the sequence in which they are written, irrespective of their logical order, which is determined by their positional value or by the key.

PRIMITIVES FOR CONCURRENT FILE ACCESS

File System Management allows the user to access files concurrently. This facility is very powerful and convenient because file sharing can be used very neatly as an interprocess communication tool. However, the various processes must have tools available to regulate concurrent access, otherwise highly undesirable results might arise from the various process activities. The user can choose between two different styles of locking, namely file locking and record locking.

This section describes the files and record locking facilities and the related primitives: **UnlockAllRecords, LockRecordPos, LockRecordKeyed.**

File Lock

When a file is locked exclusive by a process at the time of opening, no other process can Open the file (not even with no lock - i.e. a shared file). The Open primitive terminates if after **timeout** seconds any existing file lock is not released, and the file is not opened. Analogously, in order to acquire an exclusive file lock, no other process must have the file open.

When file locking is adopted, the "granularity" of the objects that the processes can exchange is quite large, i.e. the processes can exchange whole files only. All the relevant file operations are to be included within one single **critical region,** i.e. a program region such that the executing process is sure it will never be interrupted (once it has entered it) by other processes that declare they want to enter the same critical region.

The Open exclusive on a file by a process does not inhibit the Connect operation by another process; it inhibits, however, the Open following the connect. The Open exclusive on a file guarantees possession by a process of the only Open id valid relative to that file. If, however, the process decides to pass its own open-id to another process, then that other process may freely access the file, transfer data and even Close the file, just as if it were the original process.

If the file is to be shared the calling process must specify this at Open time, and then many different open-ids for the file may exist concurrently. Any calling process in possession of an open-id is able to read all the data in the file (if it has the necessary access rights). The possibility of writing records on the file also depends on the existence of any record locks. If there are none, consistency of data between operations is not guaranteed, and thus time-dependent reads are possible. For further information on access rights and locks, see also the Section "Additional Observations on Locks".

Record Lock

In many situations file lock is not desired or possible and the user wants the processes to be able concurrently to access the same file and simply to lock (and thus exchange) the records of a file. It is then necessary to provide tools that allow record updating to take place in a

protected way, yet allowing other processes to access records that are not involved in the updating. To solve this kind of problem File System Management allows the process to lock a record in a file by means of the primitives **LockRecordPos and LockRecordKeyed.**

Two levels of record locking are available, namely none (n_) and modify (m_). The none lock is equivalent to absence of lock and it is used to release the lock acquired on the record. The modify lock prevents any other process from modifying the record while allowing it to be read. Alternatively, the process can release all the acquired locks by means of the primitive **UnlockAllRecords.**

The table below summarizes user operations on files, depending on the type of lock used.

FILE LOCK	RECORD LOCK	USER OPERATIONS
no	no	Any open-id: any operation is possible.
no	yes	Locking open-id: the locked record may only be read and updated. Other open-id: the locked record may only be read.
yes	/	Locking open-id: the file may only be read and updated. Other open-id: the file cannot be opened at all.

Additional Observations on Locks

Record locking is meaningful only if the file can be effectively shared, i.e. no process must have the exclusive lock of the file. If, however, one process has exclusive lock of a file, there is no point in locking at the record level, as no other process can have the file opened and consequently no other process can request a record lock. Thus, the two styles of locking cannot be adopted at the same time for the same file.

Since record locking applies to shared files, some care must be taken when using it to update a record. A record lock allows the user to establish a critical region with respect to the file operations on that record. The correct way to use it is to include the reading of the value of the record (which might be necessary to decide its updating) in the critical region as well. Otherwise, a competing process could modify the record while the process is computing the new record value in terms of the old (and out-of-date) one.

Concurrent file access is possible using both Direct and Sequential Access Methods. The only difference is that although the Sequential Access Method provides the concept of record, **no record-locking facility is provided for Sequential files.** Record locking is, of course, of no use when no direct access to records is provided. Consequently, to avoid competition in access at record level, it is necessary to adopt the file lock (as already explained in the description of the Tapelo primitive).

The access right defines - and limits - the operations that the process can make on the opened file at all times. The file lock defines - and limits - the accesses that the process can make **concurrently** with other processes provided that the file is opened. It is perfectly reasonable to require an r access right and exclusive file lock, if the user wants to be sure of examining a file that is not changed by someone else in the meanwhile.

File and record locking interferes with and limits all operations that modify the locked object. Deletion, key attaching and detaching, and file shrinking are very important object modifications and cannot be performed if a competing lock is in force.

PRIMITIVES FOR I/O OPTIONS AND BUFFERS

This section describes the buffering modes available to the user and the related primitives: **SetIO Options, SetBufferOptions.**

Disk Writing Modes

The user process requesting a write operation can decide which of the following modes to use for each file to be written in a connect or a use session:

- Synchronous
- Asynchronous

When the **synchronous 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.

When the **asynchronous mode** is used, the requesting user process does not wait for completion of the writing operation; it receives control immediately after having made the writing request.

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 the maximum guarantee of the integrity of the data contained in the system tables.

Thus, in addition to the updating of volumes, directories and their support files, the following operations are always synchronous:

- changing **allocationUnit** via a WriteInfo primitive
- changing bounds via a SetBounds primitive
- updating the last access time and last modify time when a file is closed.

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).

The user files (byte-stream, positional, keyed) are handled by default in asynchronous mode. Changes to file bounds caused by write operations i.e. when data is appended, are therefore also done asynchronously.

Synchronous 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 so often for quicker recovery from a possible system crash.

When a user file has been Connected or connected and opened, the **I/O mode** can be changed from the default (asynchronous) to synchronous mode or one of the direct modes using the **SetIOOptions** primitive.

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 last set prevails.

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.

Disk Reading Modes

When requiring to read FS, synchronous mode is adopted by default (as the user is in any case obliged to await arrival of the data) unless the user has not effected a "sequential link", as described in the Section "I/O Primitives: Sequential Access". This allows the FSM to pre-fetch I/O blocks, thus making possible a considerable increase in the efficiency of the system. The latter may, however, be adversely affected if the user, although having requested a "sequential link" (using the SetIOOptions) then accesses the file in non-sequential mode.

Buffering Modes

The FSM makes it possible to use both system and private buffers for temporarily storing data to be transferred to disk. It also offers the possibility of operating without using these buffers at all (direct mode).

If the user requires neither the use of private buffers nor direct mode (both via the SetBufferOptions primitive), the asynchronous and synchronous modes both use buffers in the **system buffer pool.** The number and size of these buffer is defined at configuration time. By associating a private buffer to an output file, the user avoids concurrency with other programs that use the system buffers, and exactly matches I/O size to buffer size. This results in an increase of the processing speed. The set of private buffers is defined at configuration time.

The direct mode, however, allows I/O to be performed directly from user space. **Direct I/O** should however, be used sparingly; it is recommended only for large data transfers, e.g. program loading, sorting a file, transferring volumes to and from streaming cartridge tape. File System Management ensures that no memory compaction is carried out by the PMM during direct I/O. Note that direct I/O requests should line up on disk sector boundaries, otherwise the use of buffers cannot be completely avoided.

When a user file is Connected, the buffer selected by default is the system buffer pool. The use of **private buffers** to avoid contention for buffers can, however, subsequently be requested using the **SetBufferOptions** primitive, which dedicates a specified number of buffers of a specified size to the file. These buffers are still **system buffers** because they are in system area and thus not subject to memory compaction or swapping, but they are reserved for a single file and can be dimensioned appropriately. Note that if this file is in fact a volume, then all files in the volume may share the buffers in the volume's **private buffer pool.**

SECURITY PRIMITIVES

The description of the security primitives is preceded by a presentation of the MOS data protection mechanism.

Data Protection Mechanism

To provide security guarantees, MOS effects the following:

- it only allows authorized users to access the system
- it allows authorized users to access only the resources that are made available to them, and in the authorized manner.

The first of the above operations is effected by the login program when the user logs in.

The second necessitates the co-operation of various system components, mainly the FSM and PMM, which handle respectively the data stored in the system and the software constructs used for accessing it (for example, the processes).

The data protection mechanism operates as follows:

- It assigns a system identifier to each user allowed access to the system.

- It associates to each FS resource a list specifying which users may access resources, and in what manner.

- It associates to each PMM resource the identity of the user who effected the activation. This user is then the owner of that resource.

- It checks that the process requiring access to a file, and requiring to operate on it, is owned by a user having the right to effect the operation requested.

Assigning an Identity to a User

Every user enabled to use the system has an identity made up of a double system identifier:

- the primary identification (the "primary-id"), which uniquely identifies the user

- the secondary identification (the "secondary-id"), which identifies the group of users to which the primary-id belongs.

This double identification necessitates the subdivision of system users into subsets, each of which comprises those users having the same secondary-id, thus making the list of file access rights (described in the next section) more brief, without detracting from the efficiency and flexibility of the MOS data protection mechanism.

List of Access Rights

When it creates a file, FSM associates to it a list of access rights, made up of four parts.

The first part specifies the access rights of the user who creates the file (the primary owner); the second those of the secondary owner/s of the file, that is, all the users belonging to the primary owner group. The third specifies the access rights of all the "other users" of the file.

The fourth part comprises the primary and secondary bits. If set to "on" they allow the procedure effecting the load module to change its identity. For further details, see the Section "Assigning Owners to Procedures and Families".

The following access rights for the file system objects can be supplied:

read:	(for files)	Files can only be read.
	(for directories)	Directory contents can only be listed.
append:	(for files)	Files may be written, provided that writes are performed outside the current bounds of the file (this implies that it is not possible to re-write or delete data).
	(for directories)	Files may be created in the directory.
write:	(for files)	Files may be modified without limitations.
	(for directories)	Files may be created and removed from the directory.
execute	(for files)	Files may be executed.
	(for directories)	Files in the directory may be connected to.

Assigning Owners to Processes and Families

Assignment of an owner to processes and families is effected by the PMM which handles these objects. It is mentioned here in order to give a general idea of data security in MOS.

The PMM objects are assigned the identity of the user who has requested their activation, and who is thus their sole owner. PMM allows the superuser, however, to permanently change the identity of a process (using the SetIdentity) primitive, or allows a process to temporarily change its own identity (that is, for the time in which particular programs are executed).

More precisely, depending on whether the primary or secondary bit is on, the process temporarily acquires the primary or secondary id of the file. Only in these cases does the process identity differ from that of the family.

File Operations Control

A list is given below of the main operations controlled by the FSM, and the access rights (r_, x_,...) required by the process in order to effect them. These are the usual ones, i.e.:

x_ = execute	w_ = write
r_ = read	a_ = append

OPERATION	ACCESS RIGHTS
Connect	x_ on father directory, and access rights provided for this user on this file
Create	w_ or a_ on father directory
Open	appropriate openMode for this file
Read	r_
Write	w_ or a_ depending on writing
Attach	w_ or a_ on father directory and must be primary owner

The superuser may effect any operation whatever on any file.

Control operations are carried out as shown in the table below.

CONDITION	ACTION
Owner of the process = primary owner of the file	The access rights necessary for effecting the operation are to be found in the primary user's list; if they do not exist here the operation is rejected.
Owner of the process = secondary owner of the file	The access rights necessary for effecting the operation are to be found in the list of secondary users; if they do not exist here the operation is rejected.
Owner of the process different from primary or secondary owner of the file	The access rights necessary for effecting the operation are to be found in the list of "other users"; if they do not exist here the operation is rejected.

Description of FS Security Primitives

The **GetAttributes** and **ChangeAttributes** primitives make it possible respectively to read / modify the access rights of the file whose name has been specified. They effect analogous functions relative to primary and secondary bits (meaningful only for files contained in l-modules).

ReadAttributes effects the same function as GetAttributes, but requires in input the system identifier of the file instead of the name.

The **GetOwner** and **ChangeOwner** primitives make it possible respectively to read/modify the identity of the user owner of the file (primary owner-id and/or secondary owner-id).

The **ReadOwner** primitive returns the identity of the user owner of the file (as GetOwner), but requires in input the connect-id of the file.

The **ReadAccess** primitive makes it possible to read the access rights specified by the user at connect time.

FS UTILITY PRIMITIVES

A number of additional features of the File System are now described. This section may be ignored if the reader only requires a general understanding of the system.

File Saving and Management

The user may be interested in many levels of file saving, and different file management facilities are available. **Copy** and **Rename** allow him to copy a file or directory (recursively) onto a different volume. When used in conjunction with removable volumes, these facilities provide a simple logical dump service, in the sense that they selectively copy files from volume to volume. However, the single file is copied "physically", i.e. the auxiliary data structures that implement record deletion and access facilities are copied as they are. If the user wants to save disk space, the **Compact** primitive should be used, not necessarily in conjunction with file saving.

Note that **Copy** can be used to copy logical volumes onto logical volumes, thus providing a physical dump service. The reader should note that general use utilities are available to DUMP files interactively and the Streaming Tape Cartridge device can easily be handled by such utilities (although it is also accessible via File System Management as a sequential device).

To help in file management, the **Pry** primitive allows the user to retrieve a set of information about any file (including volumes and directories). This information includes the file **type** and **subtype.**

A file can be converted to a different type using the **Convert** primitive. Only a limited subset of all possible conversions is permitted, but with repeated use of **Convert** the move from one type to another can be made one step at a time.

As **Pry** shows, volumes and directories are not separate file types, but subtypes of a byte file. **Pry** allows the user to see what use is actually being made of a byte file. The subtype of a byte file can be changed by means of the **ChangeType** primitive, but care must be taken to make the necessary changes to the structure of the data in the file, since ChangeType only causes the subtype value to be changed.

Notes on File Conversion

As the internal format of positional files with delete option and keyed files has been changed in Release 6.0, it is necessary to convert positional with delete option and keyed files existing prior to Release 6.0, to the new format of Release 6.0. This conversion must be effected using the FORMAT utility before operating on files existing prior to release 6.0. The opposite conversion requires use of the TAMROF utility.

As positional packed file handling has been removed from Release 6.0, positional packed files existing prior to release 6.0, must be converted (using the CONPACK Utility and a MOS system equipped with Release 5.2 software) into positional files with delete option. These files must then be converted (using the FORMAT utility, available in Release 6.0) from the internal format existing prior to Release 6.0 into the internal format of Release 6.0.

Directory Management

ClearDir removes the files contained in a specified directory. The **clearSubDir** parameter specifies whether the contents of the subdirectories must be recursively removed. **ListDir** lists the file names of a specified directory to an **outputFile.** A recursive name subtree listing can be obtained. The **GetPathName and GetName** primitives obtain the complete pathname and the local name of a file from its system id.

The **GetPathName** primitive returns the global pathname if possible, otherwise the local pathname.

Disk Management

All disks used for MOS file storage must first be **formatted** to contain an Olivetti Standard 24 environment. Hard disks are formatted at system installation time, but new floppy disks must be formatted before use by means of the **MakeStd24** primitive.

When a disk is formatted, an initial volume can be created on it using **MakeVolume.** Then, if the disk is to be used as a system disk, the initial step of copying the operating system's switching procedure and bootstrapper onto the disk is carried out using the **MakeBoot** primitive.

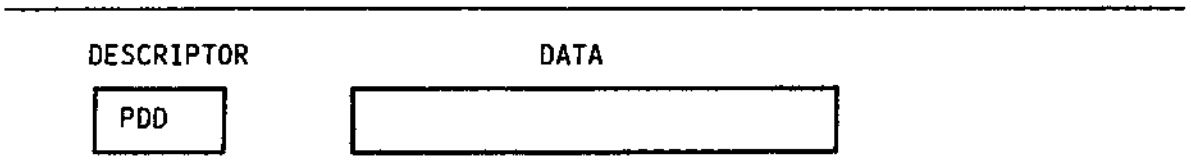
FS STRUCTURES ON DISK

This section, and a number of those following, contains general information on the physical organization of files, directories and volumes in order to ensure complete understanding of FS commands and primitives, and to enable memory occupation by the permanent files (disk files) to be evaluated.

FSM stores data on disk in files, directories and volumes. Note that even if, from the point of view of the user, they are treated uniformly, they differ considerably as objects; from the FSM point of view, however, directories and volumes are merely byte stream files containing special information.

Also, what is seen by the user as a single file is, in reality, in the case of positional and keyed files, made up of a number of elementary bytestream files: positional and keyed files are therefore multiple files. From this point on the term "elementary file" should be understood as a file seen by the system; "file" means a file seen by the user.

An elementary file on disk or in central memory is made up of a descriptor called a PDD (Permanent Data Descriptor) and by a part containing information.



The PDD contains:

- The file characteristics; for example type, sub-type, etc. Files of the following types are allowed by File System Management: byte stream, positional and keyed files (for files on disk or in central memory); files on blocked devices (floppies, mini floppies, ...) or on unblocked devices (terminals, printers, free-running connections to another system).

- The identity of the file's owner and the list of access rights to the file.

- The allocation unit.

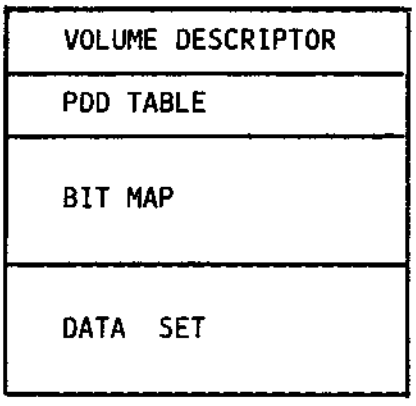
- The physical address of the first block of the "data" part.

Byte stream files allow as sub-types data files, directories, volumes etc. For further information on file sub-types, refer to the Section "Byte Stream Type Files".

The information contained in the PDD, is made available to the user not only in the form of Primitives (Pry, GetAttribute, GetOwner, ...), but also in the form of Shell commands (that is, PRY). The information contained in the PDD is placed in memory at the time of connection and remains there until the system is shut down.

VOLUMES

The MakeVolume primitive or the MKVOL command makes it possible to create a volume structured as follows:



The volume descriptor occupies 1 disk block (512 bytes). It contains relevant information on the volume such as:

- its name and size
- information on the PDD (start address of the PDD Table, maximum number of PDDs that can be allocated to it, and number of extents).

The PDD table contains file and directory PDDs, while the DATA parts of them are allocated to the Data Sets area.

Each entry of the PDD table occupies 92 bytes. The number of bytes occupied by the PDD table is equal to the number of elementary files (assigned by the user to the volume) multiplied by 92. To this volume it is necessary to add the space occupied by the four entries created automatically by FSM at the time of creating the volume, which describe the files making up the bit map, the PDD Table, the root directory of the volume and the volume log files (not used).

The Bit Map of the volume keeps track of the free or occupied blocks of the volume. It is a file whose data part contains 1 bit for each block 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:

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

The **Data Set** area contains the "DATA" part of the files.

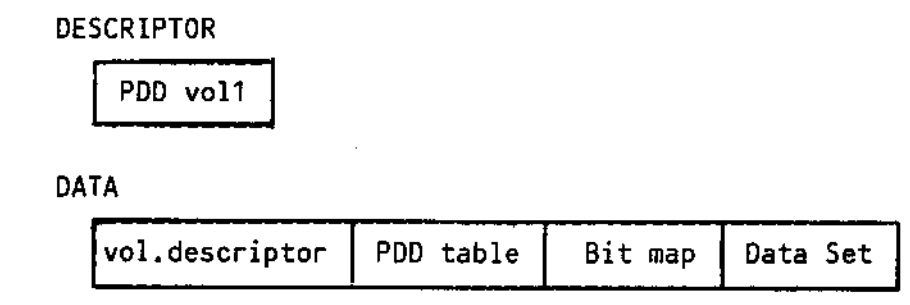
Notes:

- When the volume is created, FSM structures the volume as shown above, and creates the label, the bit map and the file relative to the Root Directory (".").
- When a volume is created, the maximum number of files it will contain should be carefully estimated, so that the PDD table is not filled when there is still free space in the volume.

Volume Description

A volume (or sub-volume) is a byte stream file of volume sub-type (vol_), organized as shown in the figure below.

The PDD of the volume is a structure allocated exclusively in memory and existing only when the volume has been mounted. The DATA part contains the information described in the Section "Volume Organization".



DIRECTORIES

A directory is a byte stream file of directory sub-type (dir_) whose data part is made up of as many pairs of elements as there are files/directories of an immediately lower level, belonging to the directory itself: the first element contains the name of a file/directory; the second is its PDD number. In this way the FS may access the structures of a lower level until reaching the leaves of the tree.

DESCRIPTOR DATA

PDD		file 1 name	PDD number of file 1	file 2 name	PDD number of file 2	....

When the directory is created (CreateDir primitive or MKDIR command) each directory is assigned a space of 512 bytes. Each new name subsequently added causes 18 bytes to be occupied.

BYTE STREAM FILES

The DATA part of a byte stream file created by the user by means of the CreateByte primitive or the MKBST Shell command only contains data. Thus the number of bytes occupied by a byte-stream file is equal to the value obtained as follows:

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

The sub-type of this byte stream file is "data_".

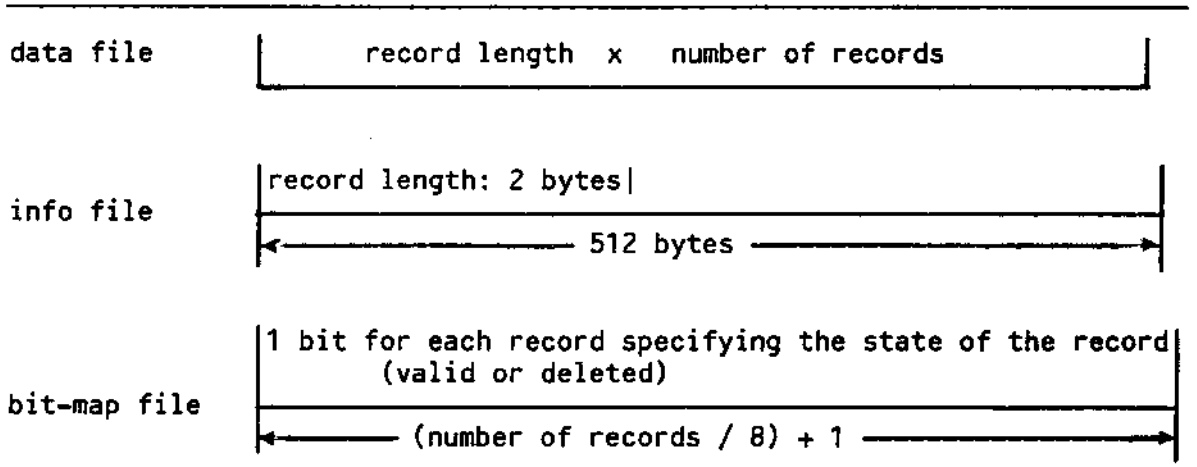
POSITIONAL FILES

If created with the 'no-deletion' option, a positional file physically consists of two elementary byte stream 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 'deletion' option, a positional file consists of three files; as well as the two mentioned above, there is a third, the "bit-map" file, which is used by File System Management to check the validity of each record in the file.

The name of the first of these files is the same as that assigned by the user to the positional file (e.g. "a"). The name of the second file is formed from that of the first, by adding the suffix "$" (e.g. "a$"). The name of the third file is formed from that of the first by adding the suffix "$" followed by the blank character (e.g. "a$ ").

The number of bytes occupied by a positional file is therefore equal to the number of bytes occupied by all the files that make it up. Thus, excluding the space occupied by the PDDs, its occupation is as follows:



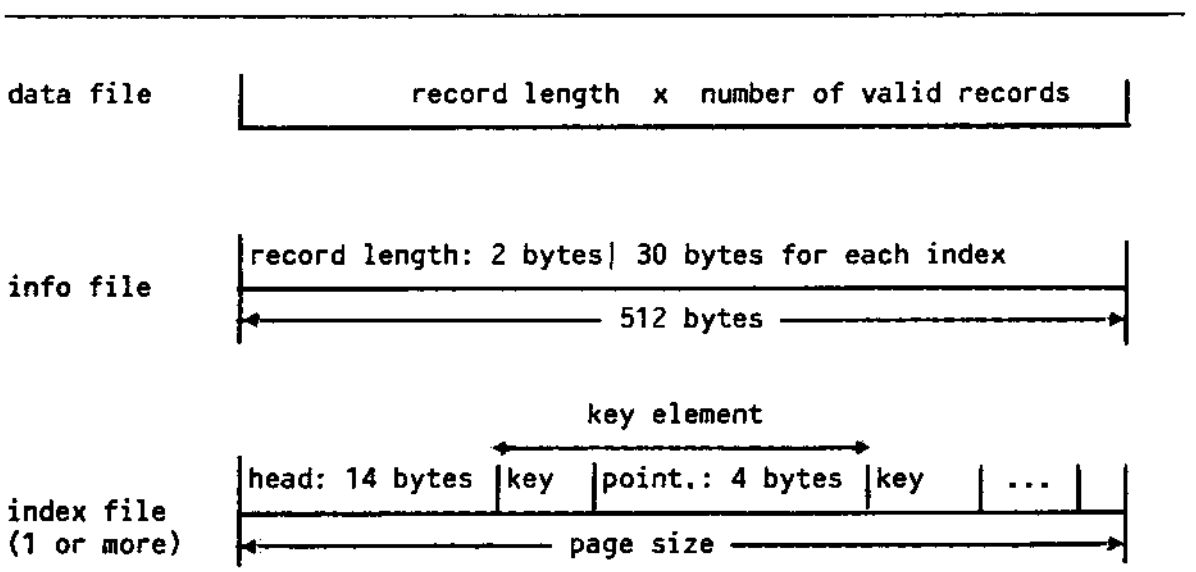
Note: The number of records means the number from the first to the last, inclusive, of the recorded records.

KEYED FILES

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 (one or more depending on whether use is made only of the primary index or also secondary indices).

These three files occupy the following bytes:



The information on the index file is provided in the following Section.

SUPPLEMENTARY NOTES ON THE KEY INDICES

All the indices relating to a keyed file are organized in a "tree" structure (see Figure 2-1). This tree is implemented with a byte file, the "index " file, which is logically divided into pages by File System Management.

Page Contents

Pages are organized according to the tree structure.

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, and 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 carried over 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 criterion defined by the user, and which can be 50-50, 90-10 or 100-0.

This criterion 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 criterion 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 criterion 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 criterion to limit the amount 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 criterion. 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.

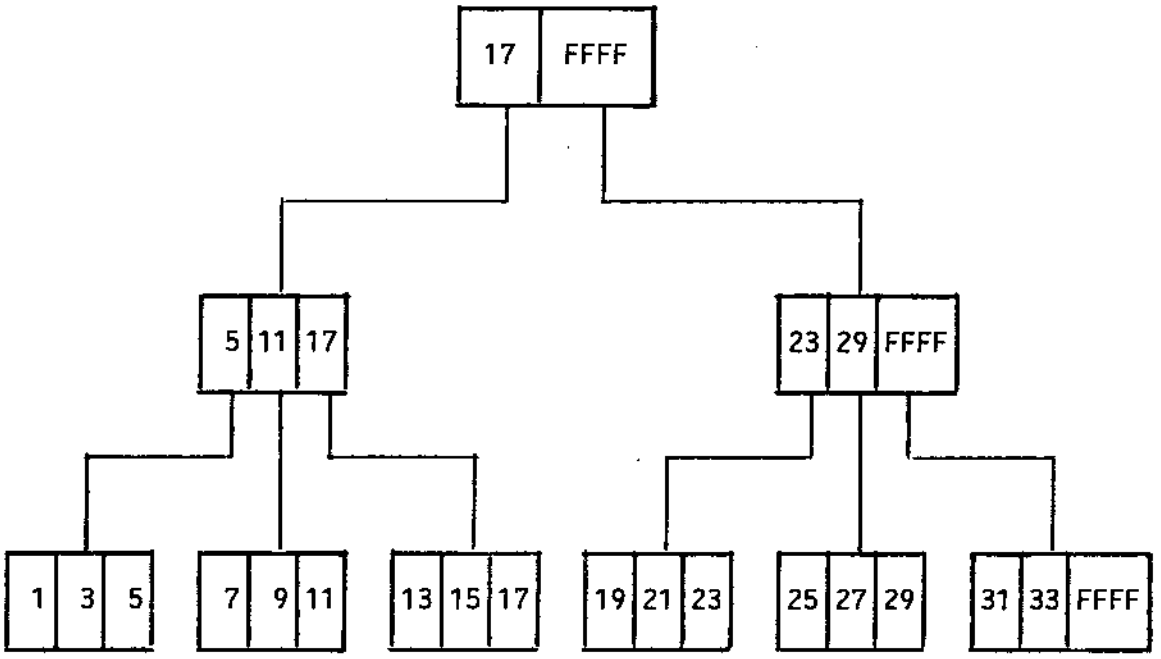


Fig. 2-1 Indices Stored in a Tree Structure

Sizing the Index file

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

The page size is 1024. The number of pages used by the index file depends on the split strategy.

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:

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

If the keys have been inserted in rising order, the number of pages is:

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

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

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

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.

If the keys have not been inserted in rising order, more memory will be required.

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 of which it consists (data, info, index).

USER VISIBILITY OF FILE OCCUPATION

Information relating to the size of some file system objects can be displayed, using the PRY command in a Shell environment, or the primitive having the same name.

With regard to the syntax to be used when calling the command, and the meaning of the information displayed, refer to the 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 concerned.

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 (valid or deleted) from the first to the last, inclusive. BUSY SPACE, however, indicates the number of valid keys between the first and the last, inclusive.

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 sections describing Byte Stream, Positional and Keyed Files.

SUMMARY OF SHELL COMMANDS RELATIVE TO FS OBJECTS

There are MOS commands, or Shell "built-in ", for the majority of name handling and auxiliary primitives.

A list is given below of these commands/built-in grouped according to the functions they carry out. The name of the command is followed by that of any FSM primitive and a brief description of the command.

DIRECTORY HANDLING

COMMAND	PRIMITIVE	DESCRIPTION
CHTYPE	ChangeType	Converts a normal directory into a program directory and vice-versa.
CLEARDIR	ClearDir	Deletes the contents of a directory.
MKDIR	CreateDir	Creates a new directory.
SHDIR	ListDir	Displays the contents of a directory.

DISK AND VOLUME HANDLING

COMMAND	PRIMITIVE	DESCRIPTION
MKENV	MakeStd24	Initializes a floppy disk.
MKVOL	MakeVolume	Creates a volume within another volume already existing, or initializes a disk (HD or FD) and makes it the main volume.
MNT	Mount	Logically connects a volume to the system.
SHLABEL	Pry	Displays the label of a disk (FD or HD) not logically connected.
UNMOUNT	Unmount	Logically disconnects a volume formerly connected from the system.
VCOMPACT		Optimizes occupation of disk space.

FILE HANDLING

COMMAND	PRIMITIVE	DESCRIPTION
COMPACT	Compact	Optimizes memory occupation of a file.
CMPK	Compack	Compacts a keyed file.
CONVERT	Convert	Converts a file from one type to another. Allowed types are: byte-stream, positional or with key.
COPY/CPTREE	Copy	Copies files, directories or volumes.
DELINDEX	Detach	Deletes an index from a keyed file.
DIFF		Compares, byte by byte, two byte-stream files or two physical devices.
EXIST		Checks the existence of a file.
LIST		Displays the partial or complete contents of a file according to ASCII or hexadecimal coding.
LS		Displays information relative to any file, directory or volume.
MKALIAS	CreateAlias	Creates an alias file.
MKBST	CreateByte	Creates a byte-stream file.
MKINDEX	Attach	Creates a secondary index in a keyed file.
MKKEYED	CreateKeyed	Creates a keyed file.
MKPOS	CreatePos	Creates a positional file.
MORE		Displays the entire contents of a byte-stream file according to ASCII coding.
PRY	Pry	Displays the characteristics of files, directories and volumes.
RCODE		Displays the release level of a loadable system module.

>>

COMMAND	PRIMITIVE	DESCRIPTION
REMOVE	Remove	Deletes files, empty directories or volumes.
RENAME	Rename	Changes the name of a file, directory or volume.
RP		Displays the physical description of a file.
SHALIAS		Displays the contents of one or more alias files.
SETBUF	SetBufferOptions	Reserves a private buffer pool for I/O operations on a certain file.
SETINFO	WriteInfoByte, ...	Defines or modifies file attributes.
* WHEREIS		Searches for files, directories and volumes.
WSTAT		Displays the physical description and logical connections of a file.
FORMAT		Causes a positional or keyed file to assume the format required by Release 6.0.
TAMROF		Carries out the reverse action to FORMAT for keyed and positional with deletion files.

FILE PRINTING

COMMAND	PRIMITIVE	DESCRIPTION
LPR		Prints or spools the contents of one or more byte-stream files (in ASCII characters).

SAVING AND RESTORING OBJECTS

COMMAND	PRIMITIVE	DESCRIPTION
FILETAR		Stores files, directories or volumes in magnetic tape archives and vice-versa.
FLD		Copies a file, directory or volume from hard disk to one or more floppy disks or vice-versa.
FLDUMP		Physically copies a volume from hard disk to one or more floppy disks or vice-versa.
VOLSR		Copies a volume from hard disk to SCT and vice-versa.

CONNECTION/DISCONNECTION OF FILES/DIRECTORIES/VOLUMES

COMMAND	PRIMITIVE	DESCRIPTION
CONN	Connect	Connection of an object.
DISCON	Disconnect	Disconnection of an object.
SHCON		Display connected objects.
CONSP		Connection between a local name and a spool class.

HANDLING WORKING DIRECTORIES

COMMAND	PRIMITIVE	DESCRIPTION
SETWDIR	*	Definition of working directory.
SHWDIR		Display working directory.

* The system-id of the working directory is stored in the program context.

SECURITY

COMMAND	PRIMITIVE	DESCRIPTION
RIGHTS	GetDefAttributes SetDefAttributes	Read or modify the access rights of an FS object.
SETMODE	ChangeAttribute	Modify the access rights of an FS object.
SETIDEN	ChangeOwner	Modify the owner of an FS object.

FS PRIMITIVES: LOGICAL USAGE OUTLINES

The following examples only show the logical arrangement of the primitives and their essential parameters. They cannot, therefore, be utilized directly in programs.

CREATING AND READING A BYTE STREAM FILE

The /IPL/AAA/BBB file should be created and then read.

```
Connect(Context,localroot,x_,Idroot,...)
   (* connects to the local root and returns the connect-id        *)
   (* of that connection to Idroot. The local root system-id is    *)
   (* given in the context and is accessed by using the Context    *)
   (* primitive.                                                   *)

Connect(Idroot,IPL/AAA,x_, Iddir ,...
   (* connects to the AAA directory whose connect-id is returned   *)
   (* to Iddir.                                                    *)
```

```
CreateByte(Iddir,BBB,...,...,...)
   (* creates the empty BBB file under the AAA directory.          *)

Connect(Iddir,'BBB',w_,Idfile,..)
   (* connects to the BBB file with the  aim of writing to it      *)
   (* and returns the connect-id to Idfile.                        *)

OpenByte(Idfile,w_,...,...)
    (* opens a writing session on the file whose connect-id is     *)
    (* specified in Idfile and returns the open-id to Idfile. It   *)
    (* is thus necessary to save Idfile before calling OpenByte.   *)

WriteByte(Idfile,1,buff,...,...,newwrite_)
    (* writes, starting from position 1 of the file, the data in   *)
    (* buff.                                                       *)

CloseByte(Idfile)
    (* closes the file writing session.                            *)

Disconnect(Idfile)
    (* closes the file connect session.                            *)

Connect(Iddir,BBB,r_,Idfile,..)
    (* connects to file BBB with the aim of reading it and returns *)
    (* the connect-id of that connection to Idfile.                *)

OpenByte(Idfile,r_,...,...)
    (* opens a reading session on the file with the connect-id     *)
    (* specified in Idfile and returns the open-id to Idfile.It is *)
    (* thus necessary to save Idfile before calling OpenByte.      *)

ReadByte(Idfile,1,buff,...,...,...)
    (* reads data in buff, starting from position 1 of the file.   *)

CloseByte(Idfile)
    (* closes the file reading session.                            *)

Disconnect(Idfile)
    (* closes the file connection session.                         *)

Disconnect(Iddir)
    (* closes the connection to the AAA directory.                 *)

Disconnect(Idroot)
    (* closes the connection to the local root.                    *)
```

CONNECTING TO A FILE OF A REMOVABLE VOLUME

It is presumed that the file /a/b/c is stored in the volume x and that this volume has been mounted under directory y. In order to connect file c it is necessary to connect pathname x/a/b/c relative to y using the sequence:

```
Connect (volumeContextid,x,volid)
    (*  connects to the volume x under directory y,                *)
    (*   as volumeContextid contains the system-id of directory y. *)
    (*  The connect-id of volume x is returned to volid.           *)

Connect (volid,a/b/c,fileid )
    (*  connects to file a/b/c of volume x.                        *)
    (*  The connect-id of the file is returned to fileid.          *)
```

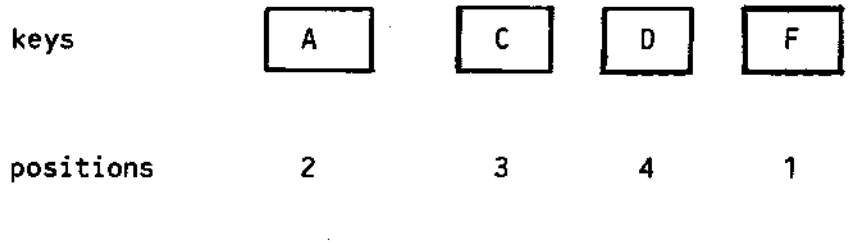
The following can also be used (or any other combination):

```
Connect (volumeContextid,x/a/b/c,fileid)
```

EXAMPLES OF USING BEGIN/COMPUTE/TRANSFORM

Sequentially Reading a Keyed File Using Compute

A file is assumed in which the keys and the positions of the relative records are as shown in the figure below:



A logically sequential reading of the records can be achieved using one of the following sequences. The Idfile parameter contains the open-id of the file. See the "Creating and Reading a Byte Stream File" Section.

The notation "@A" means "address of the record with key A".

Sequence 1

```
posRecord:=BeginKeyed(ldfile,...,Address,Key)
  (* returns the three values posRecord=2, Address=@A, and Key=A     *)

ReadPos(ldopen,posRecord,buff,...)
   (* returns  the first record  to buff; i.e the record with key A. *)

nextRecord:=ComputeKeyed(ldopen,Address,nrKeys,nextAddress,...)
  (* calculates the address  of  the  record  immediately  following *)
  (* that having Address=@A when nrKeys=1 is specified. Then returns *)
  (* nextAddress=@C and nextRecord=3. It then positionally reads     *)
  (* that record and continues in this way ...                       *)
```

Sequence 2

```
posRecord:=BeginKeyed(ldfile,...,Address,Key)
  (* returns the three values posRecord=2, Address=@A, and Key=A.    *)

ReadKeyed(ldfile,...,key,...,buff,...)
   (* returns the first record of the file to buff; that having the  *)
   (* key A                                                          *)

nextRecord:=ComputeKeyed(ldfile,Address,nrKeys,nextAddress,...)
  (* calculates the address of the record - immediately following    *)
  (* nrKeys=1 - having Address=@A. It then returns nextAddress=@C    *)
  (* and nextRecord=3. It then reads this record using Readkey, and  *)
  (* so on.                                                          *)
```

Reducing the Bounds of a File

If requiring to truncate the file shown in the above figure after key C, it is possible to obtain @C using ComputeKeyed (or TransformKeyed) and then pass this value, with the directions for truncation, to SetBounds.

In order to completely empty a file the user may place the start of the file following the end of file, or the end of file before the start, and then use SetBounds, specifying the request for emptying parameter bound=eof.

Sequence 1

```
nextRecord:=ComputeKeyed(ldfile,Address,nrKeys,nextAddress,...)
  (* calculates the address of the record immediately following *)
  (* nrKeys=1, having Address=@E. A value beyond the end of file *)
  (* value is then returned to nextAddress.                      *)

SetBoundsKeyed(openId,...,nextAddress,eof_)
   (* causes the file to be emptied.                             *)
```

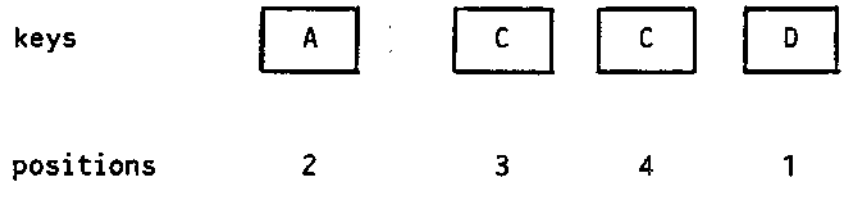
Sequence 2

```
nextRecord:=ComputeKeyed(ldopen,Address,nrKeys,nextAddress,...)
  (* calculates the address of the record immediately following  *)
  (* (nrKeys=-1), having Address=@A. It then returns a value     *)
  (* preceding the start of file to nextAddress.                 *)

SetBoundsKeyed(openId,...,nextAddress,eof_)
  (* causes the file to be emptied.                              *)
```

Use of Transform for Calculating Valid Positions

A file is assumed in which the keys and the positions of the relative records are as shown in the figure:



By means of

```
posRec:=TransformKeyed(openId,searchKey,...,...,...,exactness,address,
                       ...,...)
```

it is possible to transform approximate positions of records into valid positions as shown in the examples below, in which is shown the value of the address (returned by the primitive) according to the value of the searchkey and exactness parameters, provided by the user.

```
TransformKeyed  with searchkey=B and exactness=ge- returns address=@C3,
                founKey=C and posRec=3.

TransformKeyed  with searchkey=C and exactness=eq- returns address=@C3,
                founKey=C and posRec=3.

TransformKeyed  with searchkey=C and exactness=gt- returns address=@E,
                founKey=E and posRec=1.

TransformKeyed  with searchkey=B and exactness=next- returns address=@C4,
                founKey=C and posRec=4.

TransformKeyed  with searchkey=E and exactness=eq- returns address=@E,
                founKey=E and posRec=1.

TransformKeyed  with searchkey=B and exactness=eq- returns invalid
                address and posRec.

TransformKeyed  with searchkey=E and exactness=gt- returns invalid
                address and posRec.
```

Sequential Reading of a File Using Transform

Using the sequence shown below it is possible to read the file described in the preceding example.

```
posRecord:=BeginKeyed(Idopen,...,Address,Key)
  (* returns the three values posRecord=2, Address=@A, and Key=A.    *)

ReadPos(Idopen,posRecord,buff,...)
   (* returns the first record to buff; i.e the record with key A.   *)

posRec:=TransformKeyed(openId,searchKey,...,...,...,exactness,address,
                       founKey,...)
   (* specifying searchKey=A and exactness=next returns the address  *)
   (* position and key of the next record; i.e. the record with      *)
   (* address=@C3, founKey=C and posRec=3.                           *)
```

By means of a series of TransformKeyed and ReadPos it is possible to read the file sequentially up to end of file (signalled by ReadPos).

READING AND WRITING A MAGNETIC TAPE SEQUENTIALLY

Both reading and writing can be effected using Tapelo; in the former case ReadOpt parameter must be set to true, and in the latter, false. If the DataSize parameter has the value 0, physical reading/writing of a block from/to the buffer is effected, otherwise logical reading/writing is effected (that is, transfer of data from the buffer to an area defined by the user with the Data parameter, or vice-versa). The number of characters to be transferred is specified by the Length parameter.

The format of Tapelo is as follows:

```
Tapelo (openId,Data,...DataSize,Length,ReadOpt)
```

Reading a Tape Sequentially

```
Connect (Context,localroot,Idroot)
  (* connection to the local root.                                    *)

Connect (Idroot,DEV/MT1 Id)

SetBuffer Options (Id,1,Buffer)

OpenSeq (Id,r_,e_)

Tapelo (Id, data,0,Length,true)
  (* External physical reading loop, ending with EndOfFile.          *)

Tapelo (Id,data,...,datasize,Length,true)
  (* Internal physical reading loop ending when the buffer is empty.*)

CloseSeq (Id)

Disconnect (Id)

Disconnect (Idroot)
```

```
Connect (Context,localroot,Idroot)

Connect (Idroot,DEVMT1,Id)

SetBufferOptions (Id,1,Buffer)

OpenSeq (Id,r_,e_)

TapeIo (Id,data,...,0,Length,false)
   (* Tape at Beginning of tape: writing to a physical block        *)
   (* containing a File Mark.                                       *)

TapeIo (Id,data,...,datasize, Length,false)
   (* Loop  of TapeIo for logically writing to the buffer, ending   *)
   (* when the buffer is full.                                      *)

TapeIo (Id,data,...,0,Length,false)
   (* Loop of TapeIo for physically  writing,  ending at            *)
   (* End of Tape, if end of data has not already been reached.     *)

CloseSeq (Id)

Disconnect (Idroot)
```

3. PROCESS AND MEMORY MANAGEMENT INTERFACES

There are four sections in this chapter on the PMM. In the first, the main features of the PMM and the objects they support are described. A description of the PMM primitives is then given, dividing them into groups according to their intended use, and MOS commands related to PMM. A summary is then given of the primitives, together with a number of logical usage outlines for them.

The reader shoul note that the Section "PMM Primitives: Logical Usage Outlines" may be ignored by system administrators, as it contains information of interest mainly to programmers.

A description of the Shell commands relative to FS objects can be found in the Manual SHELL Commands - Reference Manual; a description of primitives can be found in the PMM and Driver Primitives Reference Manual. The programming environment in which they are used is described in the Manual PASCAL+ Program Preparation. A complete description of the relevant hardware I/O features can be found in M30/M40 Hardware - Architecture and Functioning.

INTRODUCTION

The main purpose of the PMM is to set up environments where user programs can be executed. This entails:

- Allocating a logical address space where the program and its data will reside.
- Loading the code to be executed and its data into physical memory, and tying the logical address space to the physical memory.
- Creating active agents (processes and process families) capable of performing the computation.
- Setting up the proper connections with the neighbouring environments and the system environment.

It also allows the processes to communicate. The latter possibility is very important. The convenience to the programmer of using more than one process to structure his application depends very much on the facilities offered for inter-process communication (IPC). The PMM offers special "tools" to let processes communicate. In this chapter, when describing IPC, this does not refer to the possibility of the processes communicating via files; the Section "Primitives for Simultaneous File Access" in Chapter 2 is dedicated to this type of IPC, which is sometimes regarded as competition among processes rather than as a way of cooperating.

In order to provide all the above functions, the PMM supports a number of abstractions: segments, programs, processes, pro cess families and channels.

PROGRAMS

In the PMM context, the term "program" indicates a program in loadable format (l-module), the output of a linker. **Programs** are collections of data and code belonging to a logical address space of segmented type (see the Section "Segments"). Every program has one main entry point.

SEGMENTS

Segments are the basic units of which a logical address space is made up. Each address relative to code and data within a program is expressed in the form of the number of segments and the displacement of the code/data within the segment. The PMM **binds** logical segments to physical memory at program load time and fills them with the code and data of the programs to be executed.

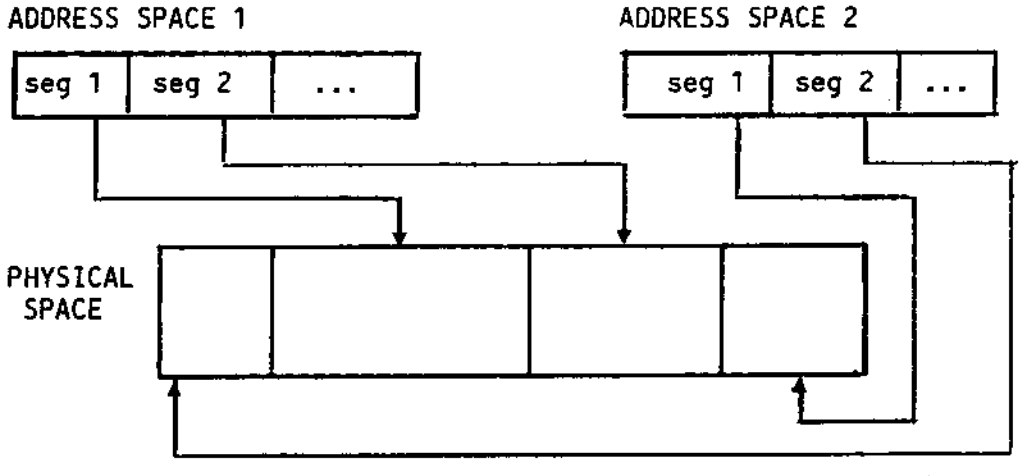


Fig. 3-1 PMM Mapping Functions: Logical and Physical Areas

Calculation of the physical address, starting from the logical one, is effected by the system when the statement containing the logical address is run, using binds established by the PMM.

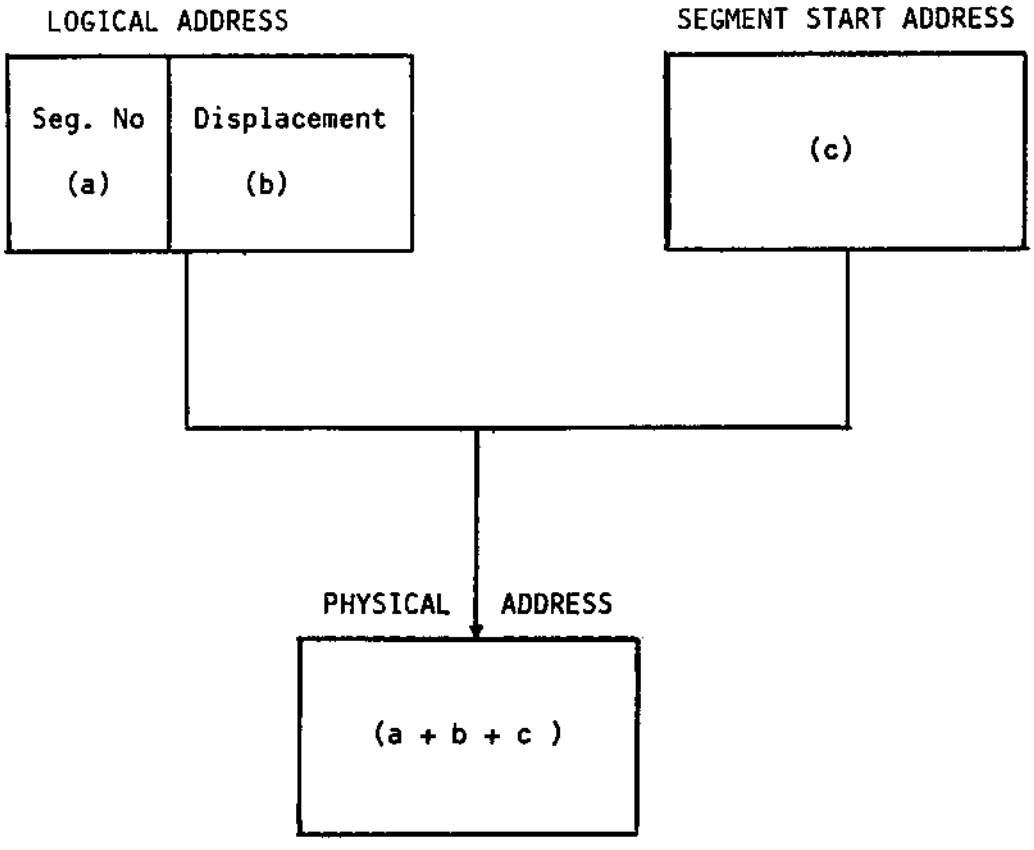


Fig. 3-2 Logical Address Mapping

Some segments are reserved for the operating system, the others are available to the user segments.

Each segment is a continuous area of memory not greater than 64 Kbytes.

FAMILIES

Families are entities where a number of resources are made available: logical segments, specified amounts of physical memory and processes (see next subsection).

A hierarchy exists among families. There is one family that is the common ancestor of all the others. This is the system family. All other families are created by the system family or by one of its descendents. The family abstraction provides the tools for building specialized application-oriented software environments(MTS, BEAM,..), the construction of which requires possession by the father family of tools for controlling the son family. The father family belongs to the software environment; the son family belongs to the user.

PROCESSES

Processes are the primary agents of computation in the system. Each active family will have at least one process that carries out the activity of the family, although as will later be seen a number of PMM primitives must be invoked before the operating environment for a family is set in such a way that a process can be started. A process is a separate stream of instruction executions that is potentially independent of other processes.

Co-resident processes belong to the same family and share the family's address space (with the obvious exception of their stacks); they cooperate by means of mutual synchronization, by executing programs loaded into the family environment (Pascal+ monitors). All co-resident processes have the same priority and no time slicing is applied across them, i.e., process switches within the same family are only performed when the running process executes a suspensive primitive (such as a monitor **wait**). Co-resident processes are meant to support tightly coupled activities. Thus they are meant closely to share the family's resources (address space, CPU time, etc.).

A processes that belongs to different families is called a **disjoint.** They are meant to support the family activities which are loosely coupled and therefore cooperate (synchronize) by means of system monitors (except when address spaces are nested, as described in the Section "Address Space Sharing"). The scheduling of such processes depends on the priorities of the families they belong to.

CHANNELS

Channels are inter-family communication media which a family uses when it wishes the **controlling** family to know that its activity is terminated and that it wants to send back some data. On creation, a family is given the identifier of the channel that it will automatically use on termination of the current activity.

The allocation (and destruction) of channels is explicit; that is, it takes place using special primitives different from those used for effecting the creation/destruction of families.

A controlling family thus has the possibility of either passing the same identifier to all the families it creates or selectively passing distinct identifiers. This gives the controller the choice of waiting for the logical disjunction ("logical or") of the completion of all its children's activities, or for the disjunction of the completion of a subset of its children or for the completion of a single child.

Also, the **channelId** can be passed around among families (within the same system). Thus the constraint that the families are organized according to a hierarchy does not affect the communication structure between a family and its children.

FAMILY ADDRESS SPACE STRUCTURE

The structure and extension of the address space of a family depends on the number of MMUs (Memory Management Units), with which the system to which the family belongs is equipped. The MMU handles a series of registers in which the information concerning the family processes is stored.

Systems Having One MMU

The total address space family spans 64 logical segments.

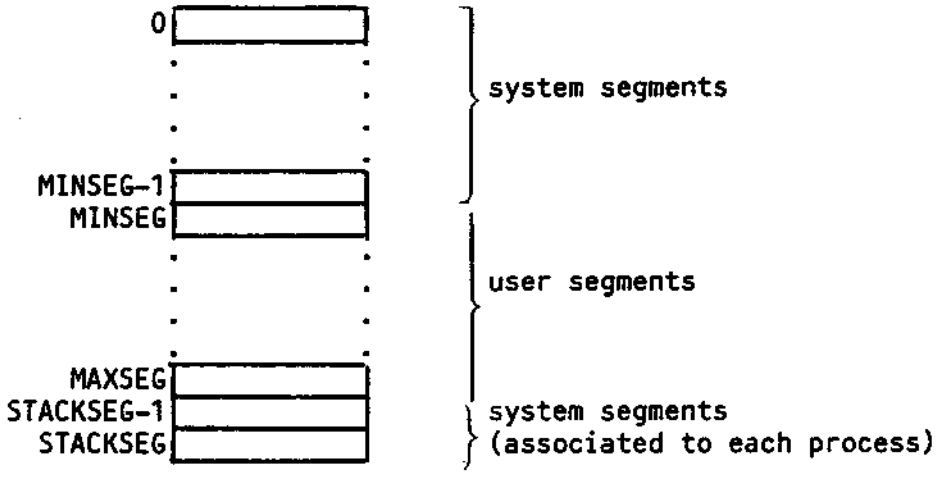


Fig. 3-3 The Segments (1 MMU only)

Segments between 0 and MINSEG - 1 are allocated to the operating system. They contain the code and data areas for the system. These segments are shared among all families, and in this way they are available to each family when it is created.

Segments between MINSEG and MAXSEG are known as user segments and are allocated on a family basis. They are shared among the co-resident processes of the family and contain:

- the code for the user programs (for example, the Shell command interpreter, the BASIC interpreter, an application program, etc)
- the data that is protected by monitors
- the data that would normally be allocated statically (e.g. common blocks).

A program and its data may occupy many segments. ZLOC and OLINK linkers assign the various code and data sections to logical segments in this range, according to the user's requirements (for the "sections" of an I-module, see Appendix A).

The rules regarding assignment of user segments are given in Part 2 of the manual.

Segments number STACKSEG - 1 and STACKSEG are the exclusive property of each process, i.e. there is a private copy of this pair of segments for each co-resident process in the family. Use of segment STACKSEG - 1 is reserved for the system, whereas segment STACKSEG is used as the process's stack.

MINSEG, MAXSEG and STACKSEG are system-dependent constants whose values are defined within a file that can be included within Pascal+ programs which use the PMM. Normal values for these constants are respectively 32, 61 and 63.

Since each family potentially has a logical address space from MINSEG to MAXSEG inclusive, it is clear that the PMM will have to map all these logical address spaces onto the available physical address space.

Since processes exist only within a family environment, they do not have a private logical address space (besides segments STACKSEG and STACKSEG - 1) and "live" in the family's address space. Clearly such close memory sharing is possible and convenient only because the Pascal+ monitors are available to synchronize and share data smoothly. Memory sharing via monitors is so convenient that a form of it is also available for the address spaces of distinct families.

Systems Having 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.

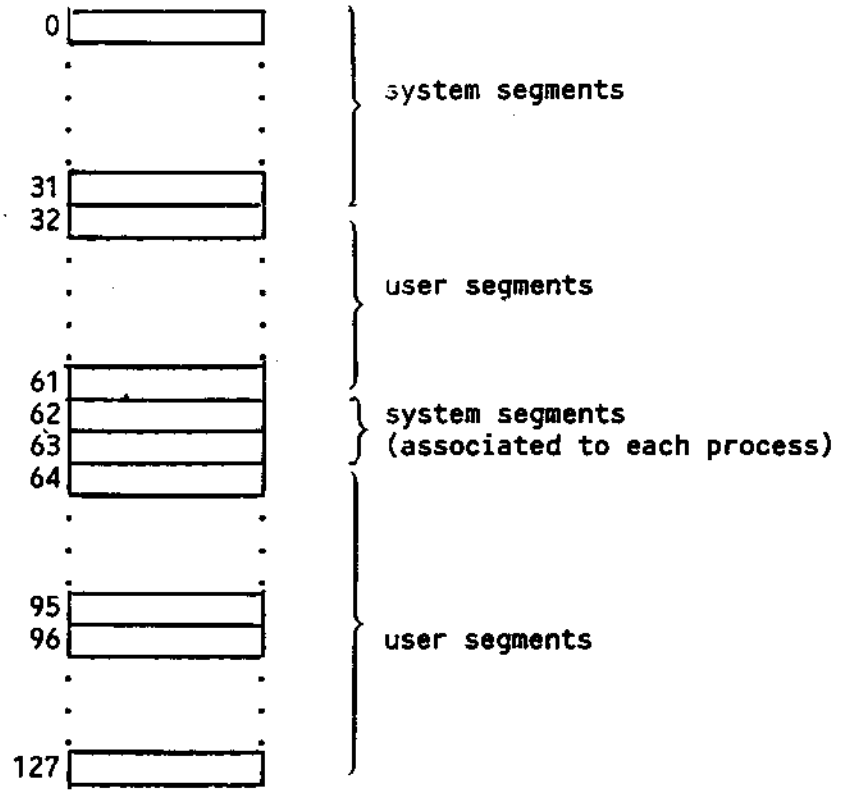


Fig. 3-4 The Segments (M60 with two MMUs)

The second MMU adds a set of logical segments (64 - 127) in which it is possible to load the user packages. See the Chapter "Assigning User Segments", Section "2-MMU Systems: Description of User Segments".

PMM SEGMENT TABLE

The information contained in this table allows the PMM to handle multi-programming, and describes such items as the size, physical location and attributes of the segments used by each family; it also describes the two segments that characterize each process.

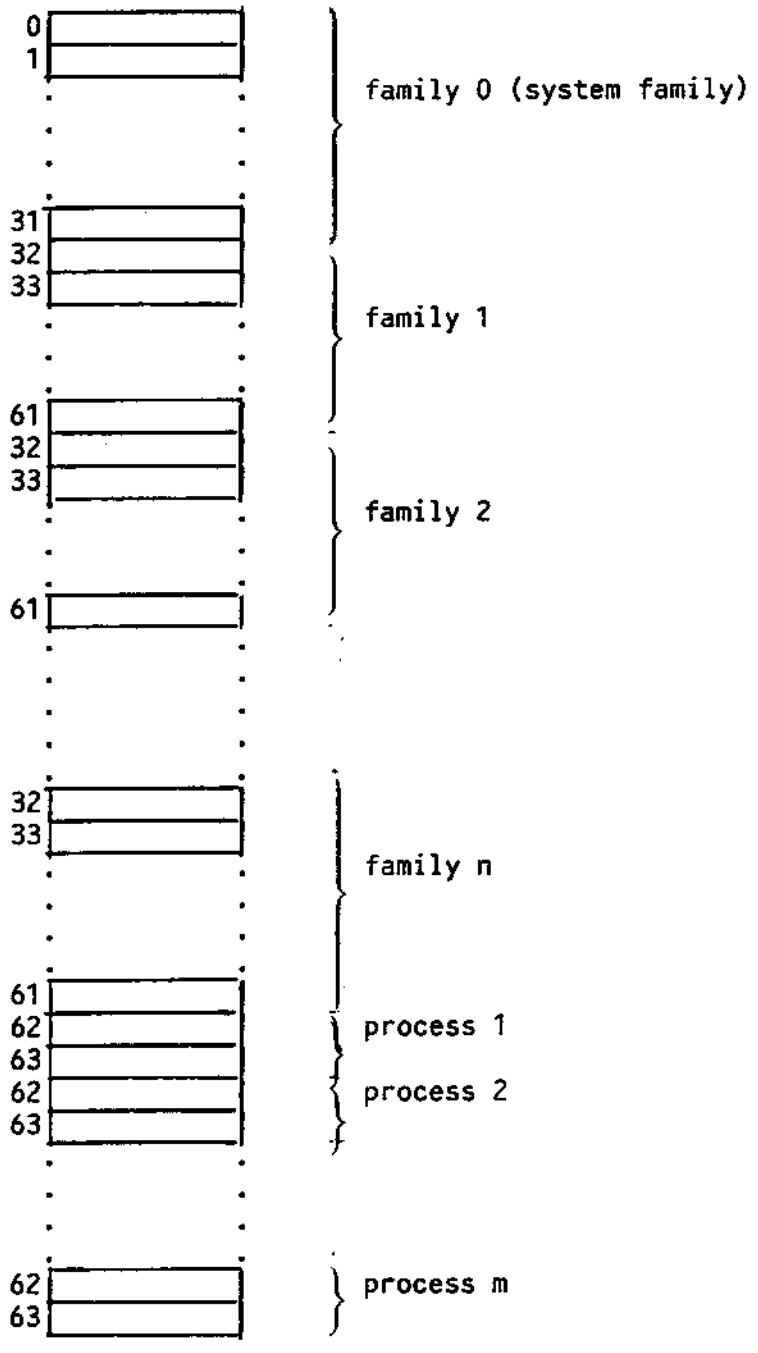


Fig. 3-5 PMM Segment Table (1 MMU only)

This information is stored by PMM using the MMU register contents at the switching time context. One of the following operations is effected when control of the processor is taken from one process and passed to another:

- if the new process belongs to the same family, the contents of the two MMU registers relating to the suspended process are saved in this table; in this way the process can correctly restart execution later (Partial Context Switch)

- if the new process belongs to another family, the contents of the MMU registers relating to the suspended process are saved, as well as the information on the user segments of the family to which the suspended process belongs (Total Context Switch).

PMM PRIMITIVES

This section provides a detailed description of the PMM primitives; they are grouped according to their purposes. Information is also provided on Shell commands related to PMM objects.

INTER-FAMILY COMMUNICATION

The primitives **MakeChan and DestrChan** support the dynamic creation and deletion of inter-family channels. Since (as already explained) manipulation of the channels is explicit, processes have the flexibility needed to pass channel identifiers to created families according to their needs. Creation of the family is effected by the Spawn primitive (described in the Section "Family Creation and Destruction").

Closely related to the above is the primitive **WaitResult** (described later in detail) which allows the issuing family to wait for termination of any family that received at **Spawn** time the channel identifier specified as an input parameter to **WaitResult. WaitResult** returns a string value which conveys the results of the terminating family activity. The communication channel is established at **Spawn** time (by the spawning family).

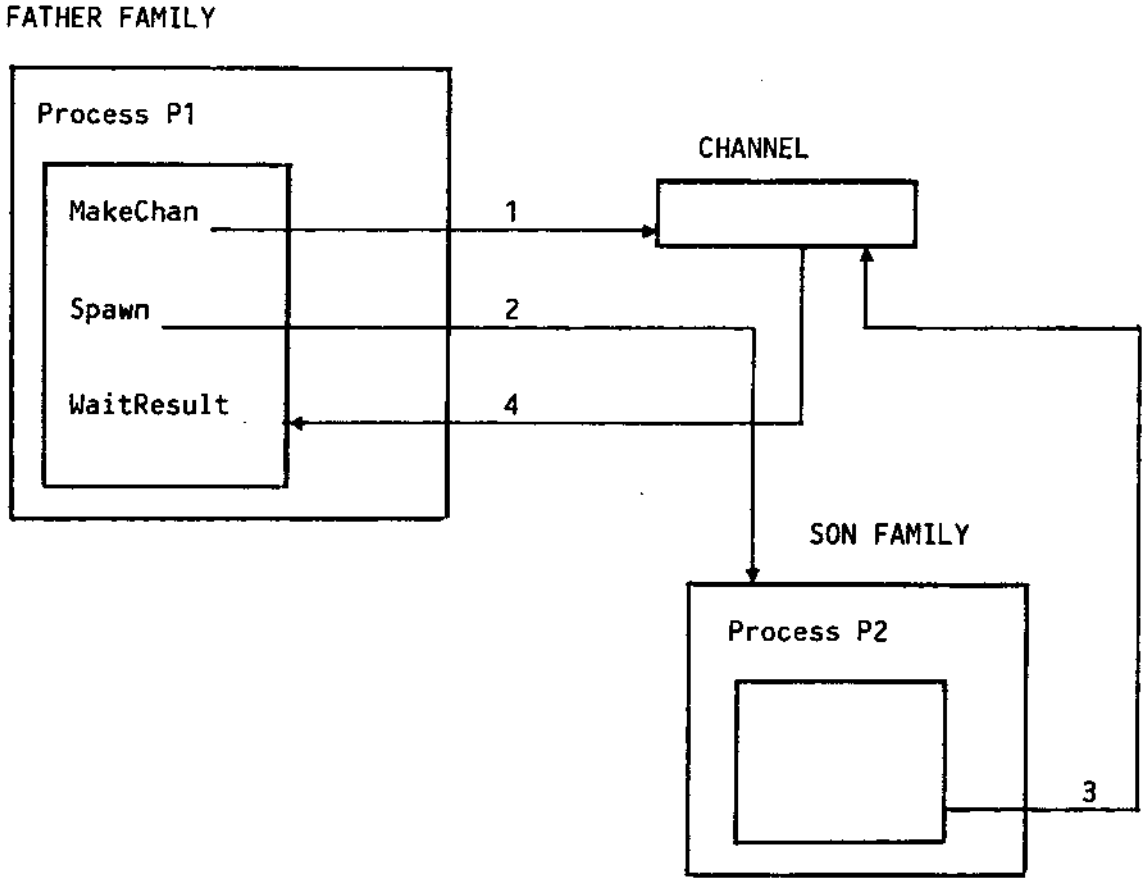


Fig. 3-6 Communications between Families Using a Channel

1. MakeChan returns the system-id of the channel created.

2. Spawn creates the son family and informs it of the system-id of the communication channel.

3. When the son family terminates, it communicates a termination result to the father family, via the channel.

4. WaitResult receives the above result.

Note that the creation and activation primitives for Process P2 are omitted from the figure.

That one specific process is causing termination of the family, and another is awaiting results on behalf of the controlling family, should not be confused with the fact that communication is taking place between families rather than processes. The two processes are in fact the active agents for the two families.

Termination is decided by a specific process acting for the whole family it belongs to, so that all family processes are terminated. Conversely, the computed result string is sent (via the channel) to the controlling family without having to specify which of the controlling family processes should receive the result. The controlling family is autonomous in deciding **at run time** which one of its processes will actually receive the controlled family's results.

Since channel identifiers can be passed around among families **within the same system** , the family which will actually receive the results of a given activity need not necessarily be the family that spawned the activity that is terminating. This decoupling of the issuing activity from the receiving activity is one of the advantages obtained by introducing two levels of control: processes and families.

Finally, note that families may communicate via system monitors and user-defined monitors since they can share parts of the address space (q.v.).

FAMILY CREATION AND DESTRUCTION

•This Section describes the primitives for the creation and destruction of a family, and provides information necessary for handling families and for correctly evaluating a number of parameters of the primitives described.

Family Creation

A new family can be created by invoking the **Spawn** primitive. The caller is allowed to specify a number of parameters:

1. A channel identifier that the family's processes will use for sending back results on completion.

2. The identifier of a context, i.e. one object that the new family is given access to; for example, the identifier of a directory might be located in the context, thus allowing the families to communicate via files (see the Context primitive described in the Section "Passing Parameters to Programs").

3. A user code assigned to the family as an alias (this code will be received by the controlling family process waiting for completion of the task carried out within the family, along with the family identifier).

4. A set of attributes that describe the operating environment for the new family.

Family Attributes

The following attributes must be specified at Spawn time by the father family for the new family:

procPrior The priority of all processes that belong to the family.

timeSlice The time slice for all the processes of the family.

budget The amount of memory (expressed in 256-byte memory page units) that is to be preallocated to the new family.

privateSegs The number of segments private to the environment, not including the STACKSEG and STACKSEG-1 segments for the processes belonging to the new family (the latter are always private).

Detailed information can be obtained on the segments, families and processes by means of the NOSE utility program, which can be activated in the Shell environment.

Process Scheduling

The policy adopted for multiplexing the real processor among the families and processes is determined by the **timeSlice** attribute. First of all, the processor is never switched by the system from one process to another within the family (no process pre-emption within the same family). Co-resident process switches are performed only when the running process executes a suspensive primitive (such as a monitor **wait** or a system primitive that involves the suspension of the calling process). Thus, **timeSlice** determines the scheduling of the processor among the various families. The non-preemptive process scheduling within the same family is consistent with the reasons for introducing co-resident processes. Since these processes are tightly cooperating, there is no point in considering them as competitors with respect to CPU time.

When **timeSlice** is non-negative, it expresses the number of 100 ms ticks a family is allowed to run even if there are other families with the same (or greater) **procPrior** priority and with a ready process. After such a time slice has elapsed, the processor is taken off the running family process.

When **timeSlice** is set to a negative value, pre-emption is disabled for the family. This means that a ready (process of the) family of this kind gets the processor according to its **procPrior** priority and does not give it up until it executes a suspensive primitive. This facility allows families with real-time requirements (for which preemptive scheduling would not be tolerable) to co-exist with non-real-time ones (to which pre-emptive scheduling is appropriate) which will actually run during the idle periods of the real-time ones.

Family Priority and Process Dispatching

When a family is created it is assigned a priority, which is used by all the processes belonging to that family. Activation of ready processes (processes able to run) takes place on the basis of priorities.

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 called.

The value supplied by the user is added to the priority values of the families 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 modifications to the priorities requested by the user by means of PRIOR, the latter can simply call the PRIOR command with a value of 0 as its parameter.

Shared and Private Segments

The address space of the new family is initially empty, except for the segments between 0 and MINSEG - 1, which are shared by all families and contain the system monitors, and for any segments shared with the father family. In fact, the portion of the new logical address space that will be completely private to the new family includes the segments whose number is in the range (MAXSEG - privateSegs + 1)..MAXSEG.

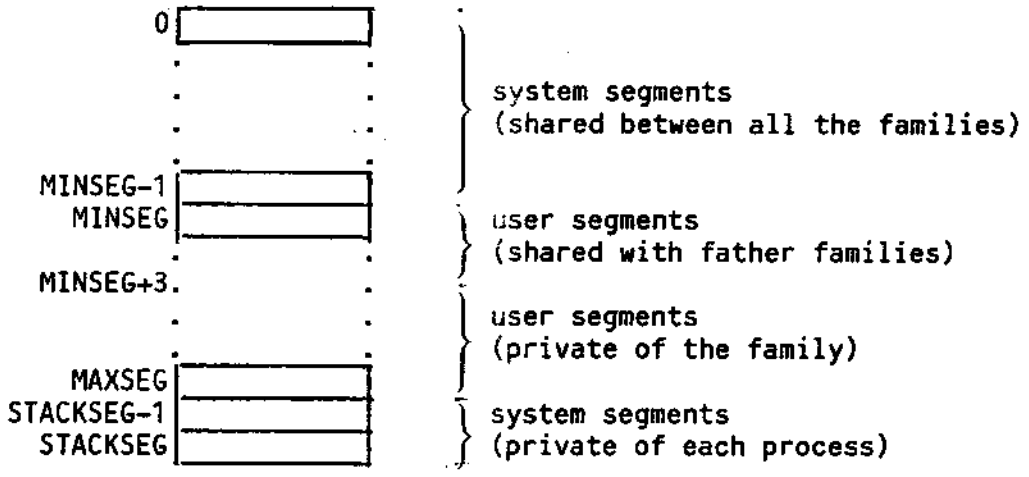


Fig. 3-7 The Private and Shared Segments (1 MMU only)

All the segments in the range MINSEG..(MAXSEG - privateSegs) are not the property of the new family; they are shared with its parent and of course their contents are not controlled by the new family (in the sense that programs using such segment numbers cannot be loaded or unloaded by the new family). In this way it is possible to create **nested address spaces** by giving a child family a limited vision of its parent's address space. Since in turn the parent's address space can be (in part) inherited from and shared with its ancestor(s), a complex nesting of address space can be obtained. It is important to note that only one family is in control of any given logical segment and this controlling family is the only family for which such a segment is private.

Physical Memory Management

Since the private address space is empty at Spawn time, no physical memory is needed to contain it. When physical space is needed (e.g.to load a program or to enlarge a data or stack segment) the physical memory is taken either from the **budget** that was allocated at **Spawn** time (if a positive **budget** was specified), or from the preallocated memory of one of the family ancestors (if one of them has specified a positive budget).

If no budget has been allocated by the ancestors, the necessary memory is first taken from the free physical memory, and then released when the requesting family does not need any more (e.g., when a program is unloaded). In fact, since the whole system can be thought of as being generated by the original system family which owns as budget the whole physical memory present in the system, there always exists one ancestor which owns a memory budget.

If not enough pre-allocated memory can be found when needed, then an error is reported. Note that if memory is pre-allocated for the family or one of its ancestors, the family is guaranteed that memory requests will always be satisfied until the allocated memory is exhausted, and conversely that such a limit will never be exceeded.

Family Destruction

The primitive **Destroy** is invoked by the controlling process to destroy the specified family. A family can be destroyed even when there are some processes active within its address space.

ADDRESS SPACE MANAGEMENT

This Section describes the primitives for loading/unloading programs to/from the address space of a family, and gives information on family management which is necessary for correctly evaluating certain parameters of the primitives described.

Program Loading and Unloading

The two primitives **Load and Unload** are provided for the purpose of loading programs into a family's address space and unloading them from it.

The **Load** function causes a program to be loaded into the address space of the specified family from a file that contains the l-module information generated by a linker.

The figure below shows segment occupation in the address space of the son family, brought about by the following primitives:

Spawn (...,privateSegs)	which creates the son family B and assigns it a number of private segments of the value "privateSegs"
Load (...,progX)	which loads program X into the son family address space
Load (...progY)	which loads program Y into the address space of the family itself.

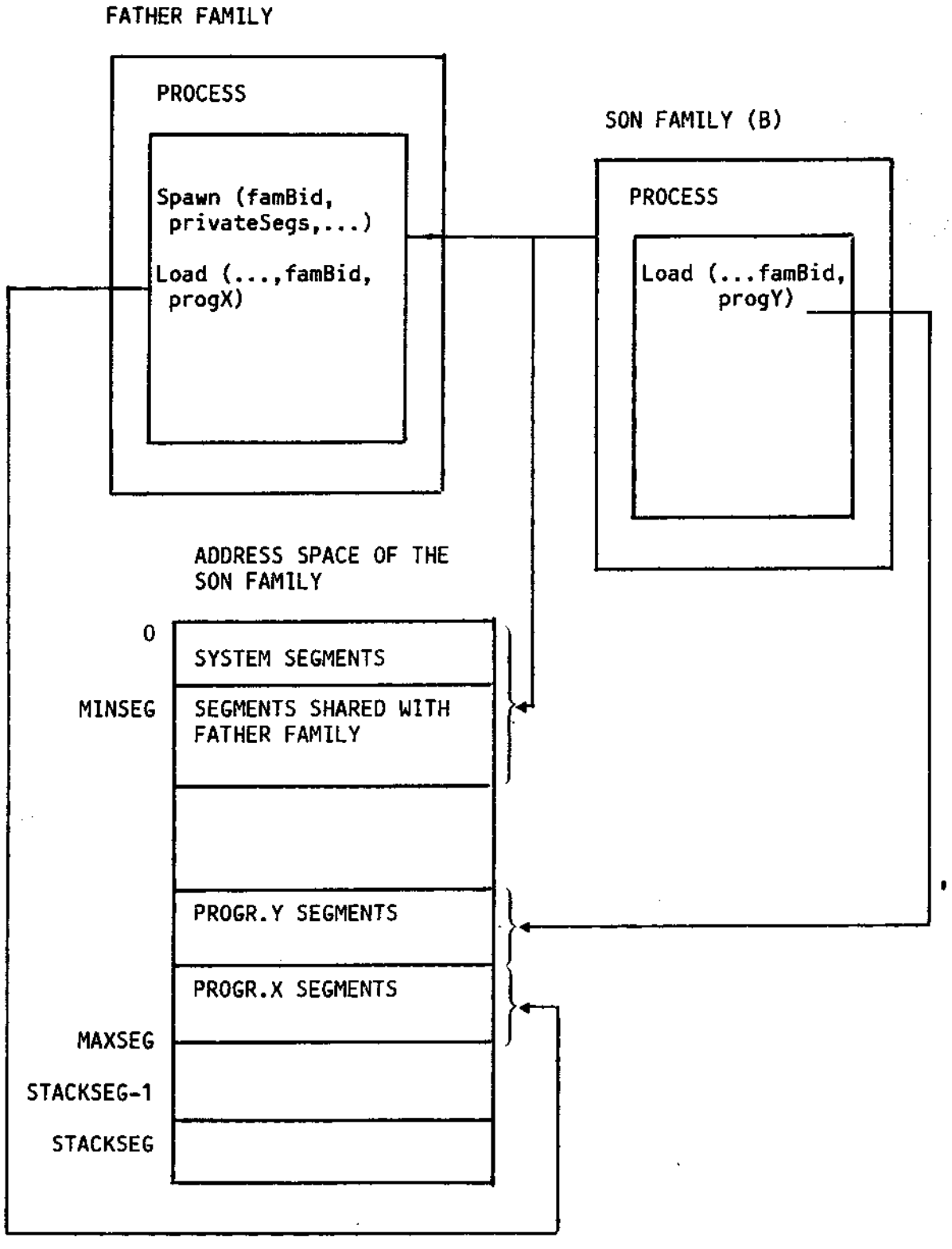


Fig. 3-8 Address Space Management

Among other information of relevance to the operating system, an l-module assigns "attributes" and "types" to each segment that belongs to the program. Such values are under user control.

The **type** information lets the operating system know what the contents of the segment are and how the segment should be handled. The **attribute** information allows the PMM to set the hardware protection mode of the physical segment which will be used to store the logical one. Only some values of the "type" are compatible with each value of the "attributes". The PMM does not allow l-modules with specified types or attributes that are outside the specified range or that are not compatible with each other.

In order for an l-module to be loaded into an address space, the identifier obtained by a file system Connect operation on the executable file is passed to **Load** with the identifier of the family where the program is to be loaded. An l-module is a (static) FS object of the bst_ type (bytestream), of the sub-type "loadable" (l-module). The input file identifier may also identify a File System object of type "bst_" (byte-stream) and sub-type "program_" (program directory). In this case, the object loaded into the input address space is a file called "MAIN" (which is the main program) that has to be present in the directory specified by the input file identifier. The Load primitive takes care of connecting to this "MAIN" file and the Unload primitive handles its disconnection. If the family identifier is set to "nil" then the program will be loaded into the caller's address space.

The **Load** primitive makes available to the program, loaded in a certain family, the family context; when loading a program directory it stores the system-id of the program directory in the context and effects the connection to the program directory.

Load returns a program identifier for the loaded program. Note that such an identifier is unique for the address space it refers to; i.e. loading the same program into two different families always yields different program identifiers and each identifier can only be used with reference to the associated address space.

Note also the following points:

1. Code and read-only data segments are automatically shared among families when multiple copies of the program are needed. Of course, private copies of read/write data segments exist within each family.

2. **Load** can only be performed within the portion of the logical address space that is private to the family (i.e., the segments that make up the program must be in the range (MAXSEG - privateSegs + 1)..MAXSEG).

3. Once a program is loaded into a family's address space, all its children that have some form of visibility of their ancestor's address space also have visibility of the segments that make up the program, within the constraints imposed by their "privateSegs" attribute.

Because the processor architecture requires, effectively, segment numbers of instructions and data to be placed within instructions, most language

processors are not capable of generating position-independent code. Programs are therefore not generally position-independent **with regard to the logical address space** (i.e., the numbers of the segments they use are generally set once and for all at link time). However, note that programs are position-independent in the real memory of the processor. Thus, most programs must always appear in the same part of the logical address space every time they are loaded. If those segments are not empty at the time of the **Load** system call, an error is reported. The loaded program stays in the address space until it is explicitly unloaded.

The **Unload** function removes the specified program from the address space (and again removes the visibility of the program segments, if any, from the address space of children and grandchildren).

Relocation of Segments

The PMM supports the notion of relocatable segments that can be bound to any part of the logical address space they are loaded into. This is to be used for interpreted programs only. The compiled programs are in fact relocatable as regards the logical, but not the physical, memory.

Whenever an interpreted program has relocatable segments, their nature should be declared by specifying the appropriate type/attribute combination in the l-module containing the program (types in the range RLRDCODE..RLRWDATA). The segment number information for relocatable segments is not significant and is only used to specify the entry point for a program with reference to the dummy segment number where the entry point is to be located.

For example, if a program has a relocatable read-only code segment (type = RLRDCODE) whose dummy segment number is 32 and the entry point is at offset 330 within that segment, then the entry point for the l-module is to be specified at segment 32, offset 330.

At **Load** time, when the PMM detects the presence of a relocatable segment, it searches the logical address space starting with segment number MAXSEG down to the first private segment for the family, until a free segment is located; if none is available, an error is returned. Otherwise, the first free segment found is loaded with the contents of the relocatable segment.

Note that the order in which the relocatable segments will appear in the logical address space is reversed with respect to the order they have within the header of the l-module to which belong. An example will make things clearer. Assume a program is to be loaded into an address space where segments 61, 59 and 58 are in use. Also assume the l-module only contains three relocatable segments: a code segment whose type is RLRDCODE, whose dummy segment number is 32, where the entry point is located at offset 0; a read-only data segment whose type is RLRDDATA and whose dummy segment number is 33; a read/write data segment whose type is RLRWDATA and whose dummy segment number is 34. Then, since the segments are listed within the l-module by ascending segment number (in this case: code segment first, read-only data segment second and read/write data segment last), the PMM will load the code segment into logical segment number 60, the read-only data segment into logical segment 57 and the

read/write data segment into logical segment 56. The actual entry point will be placed at segment number 60, offset 0.

On return from the **Load** primitive, the Status primitive (see section on debugging primitives) can be invoked with the new program-id to find out what the actual position was, within the logical address space.

Data Reset

Some languages (e.g., FORTRAN, COBOL and others) have data initialized at compile time. In the course of a computation such data may be changed, so that at the end of it the program cannot be executed as it is, since the initialized data areas have lost their initial contents. Normally this could require the unloading and successive reloading of the program. However, this has the unpleasant effect of also unloading and reloading the read-only segments, which were not modified. To allow for this situation, the **Reset** primitive is provided. It is called with the identifier of a program as a parameter (the one returned by **Load)** and takes care of initializing the read/write data sections only if they have actually been modified since the initial load (the underlying hardware is capable of detecting this).

If **Load, Unload and Reset** are not acting on the caller's address space, no processes must be active within the target address space, otherwise an error is reported.

Information on Segment Types

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.

Fig. 3-9 Segment Types

Types 1, 2, 4, 7 and 9 identify the segments which can be shared among several families. That is, if the same 1-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

Fig. 3-10 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. **StatSegment** returns information regarding the size and type of a given segment.

Modifying the Size of the Segments

A process may change segment sizes by using the **SetSegment** primitive.

This primitive can only be applied to segments in use, either belonging to a loaded program or to a process's stack. The user cannot change the size of segments in the range 0..(MINSEG - 1) nor that of segment STACKSEG - 1 since all such segments are the exclusive property of the system.

SetSegment changes the size of a segment by a specified value, expressed in 256-byte units. The increment can be positive (the segment size is increased) or negative (the segment size is decreased). In no case is a segment allowed to be dynamically created or destroyed.

In order to locate the desired segment, both **SetSegment** and **StatSegment** (described below) need two input parameters: a segment number and the identifier of an address space.

The first parameter is a number in the range MINSEG..MAXSEG or the number STACKSEG for a stack segment. The second parameter can be a family identifier (returned by Spawn), a process identifier (returned by GetMyId) or the Pascal nil pointer. Nil refers to the address space of the current process. Any one of the three types of identifier is accepted when reference is made to a segment in the range MINSEG..MAXSEG.

However, when reference is to the stack segment, only a process identifier or nil are valid, since a family identifier cannot unambiguously identify a specific stack segment.

Only segments declared to be of the types listed below can be expanded or shrunk:

RWCODE (read/write code segment)

RWDATA (read/write data segment)

STACK (stack segment)

RLRWCODE (read/write relocatable code segment)

RLRWDATA (read-write relocatable data segment)

As already described, **StatSegment** returns information regarding the size and type of a given segment.

FAMILY AND PROGRAM ACTIVATION AND TERMINATION

Once one or more **Load** operations have been performed, there must be a means of activating the newly loaded program. The operations of activation differ depending on whether the new program belongs to the address space of the family in which the activating program is loaded, or to another family. In the latter case activation of a new family can still be effected.

Activation by a Program Belonging to the Same Family

If the new program is loaded in the address space of the family that performed the **Load** operation and is prelinked to the program that carried out the **Load**, a simple procedure call will be sufficient. This will also allow control to be transferred to various entry points in the new program (not necessarily the main one). In fact, the linkers are capable of outputting symbol tables for a program that is being linked, and has the complementary facility of accepting external symbol tables in input: thus two separate l-modules can actually be linked to each other, with no overhead at run time.

In addition, any program being executed can transfer control to a new one that is not prelinked to it (and yet is loaded within the same address space) by means of the **Call** primitive that sets the program counter of the invoking process to the main entry point of the program whose identifier is passed as a parameter. See the figure for this primitive in Fig. 3.10.

A new stack frame is built on top of the previous one in the process's stack. When execution of the called program terminates, control is transferred back to the caller. Notice that a mechanism is provided to pass parameters in input to the called program, encoded as a string of characters. In fact a system-wide convention exists for this form of

encoding that allows both positional and keyword parameters to be specified; also some system utilities are provided that allow both the encoding of standard types into strings and the retrieval of standard types from strings.

The following parameters are given in input to Call:

1. A packed array of characters that contains the input parameters ("parms").

2. The index of the first character in the array where the string of input parameters begins ("startIndex").

3. A count with the length of the input string ("count").

4. An integer variable ("resType") by means of which the terminating program can qualify the computed result.

5. The identifier of the called program (obtained by an invocation of **Load**).

6. A variable where the system stores the diagnostic result code of the Call invocation ("outcome").

When the called program terminates (see the Section "Program Termination") the calling program resumes its execution at the statement following the Call invocation. The "parms" array that was used to pass parameters to the called program is also used to retrieve a string of characters. In general, its contents will be overwritten, starting at position "startIndex"; on return, "count" will contain the number of characters in the returned string and "resType" will be set according to existing conventions between the caller and the called program.

Activation by a Program Not Belonging to the Same Family

Start is used to cause the execution of a program in a family different from that of the loading process. Hence, it has ties to the world of programs as well as to the world of processes. It provides the means of activating a family, since family creation and address space loading do not initiate any processing activity. In other operating systems the actions of creating, loading and activating a process are inextricably tied together. Here, these actions are separated in order to provide the programmer with more flexibility, especially as far as address space management is concerned.

The **Start** primitive creates a process in the address space of a controlled family and forces the new process to execute a program whose identifier it receives as a parameter. As in the case of **Call** , the program is passed a string of characters that encodes all its input information. The input parameters to **Start** assume a meaning similar to that of parameters to **Call** , except that on termination of the started program, the input array will not be overwritten with a result string (because they belong to different address spaces).

Family and Program Termination

Termination of a program can be achieved by means of the **Halt** primitive. This is true for both started and called programs, thus providing complete independence of a program from the way it is activated; behaviour of the system is, however, different in the two cases.

If the program was activated by a **Call** primitive, then the calling program (in the same address space) will continue from the instruction following the **Call** invocation (see the **Call** primitive above).

If the program was activated by a **Start** primitive, then all the activities of the family where the **Halt** is issued are terminated.

Note It is safe to stop all the other co-resident processes of the family because they are either waiting on a monitor or ready to run after the condition they were waiting on was signalled (remember that there is no pre-emption among co-resident processes). The co-resident process that issues the **Halt** primitive does not need to be the process that was created by execution of the **Start** execution. Any of the co-resident processes that have been subsequently forked autonomously by the created family may decide to terminate the activity of the whole family. Thus, the program where the **Halt** call is found will not necessarily be the program that was loaded by the family that issued the **Start** call.

Returning Results

The results of an executed program or a family activity are returned to the calling program or to the controlling family, respectively. A string of parameters can be passed to **Halt** with the array index of the first character to be passed and the total count of characters in the result string. Also a result type (whose value is chosen according to conventions set up between the caller and the called program) is returned.

The PMM is capable of detecting whether the program was activated via **Call** or via **Start.** In the former case, the result string is copied to the array passed at **Call** time and the count and result type are returned as well. After copying the parameters, control is passed back to the instruction that follows the **Call** invocation in the calling program.

If the program was activated via **Start** , the family that controls the started program can synchronize on termination of the latter by means of the **WaitResult** primitive.

WaitResult is invoked by the controlling family by specifying a channel identifier that was assigned at **Spawn** time to one of the families with which the controlling family wishes to synchronize.

This primitive returns a result string, a count of characters and a result type and may receive an error code if the receiving array is not large enough for the returned string (in the same way as for **Call**). Moreover, since several families may have been given the same channel identifier at **Spawn** time, a family identifier and the user code associated with the family are also returned, so that the controlling family is able to know which family terminated its execution.

Besides using the **Halt** primitive, there is another way user programs written in Pascal can terminate their execution: they can simply reach the end of the code within the Pascal main program. If the process reaching the end of the Pascal main is the last one in its family, this is interpreted by the PMM as equivalent to a **Halt** with a result type equal to 0 and a null result string. Otherwise, the process silently dies and the rest of the family activity performed by the remaining processes continues.

Note that the functional value of **WaitResult** allows the caller to distinguish among various types of termination for the controlled family. Special codes are in fact returned when the process in the controlled family is destroyed (presumably because of a segment fault), or is suspended or hits a breakpoint.

After the **Halt** , a new **Start** may be invoked by the controlling process either to have the same program re-executed or to send a new one for execution. In the second case, various combinations of **Load, Unload and Reset** have to be invoked.

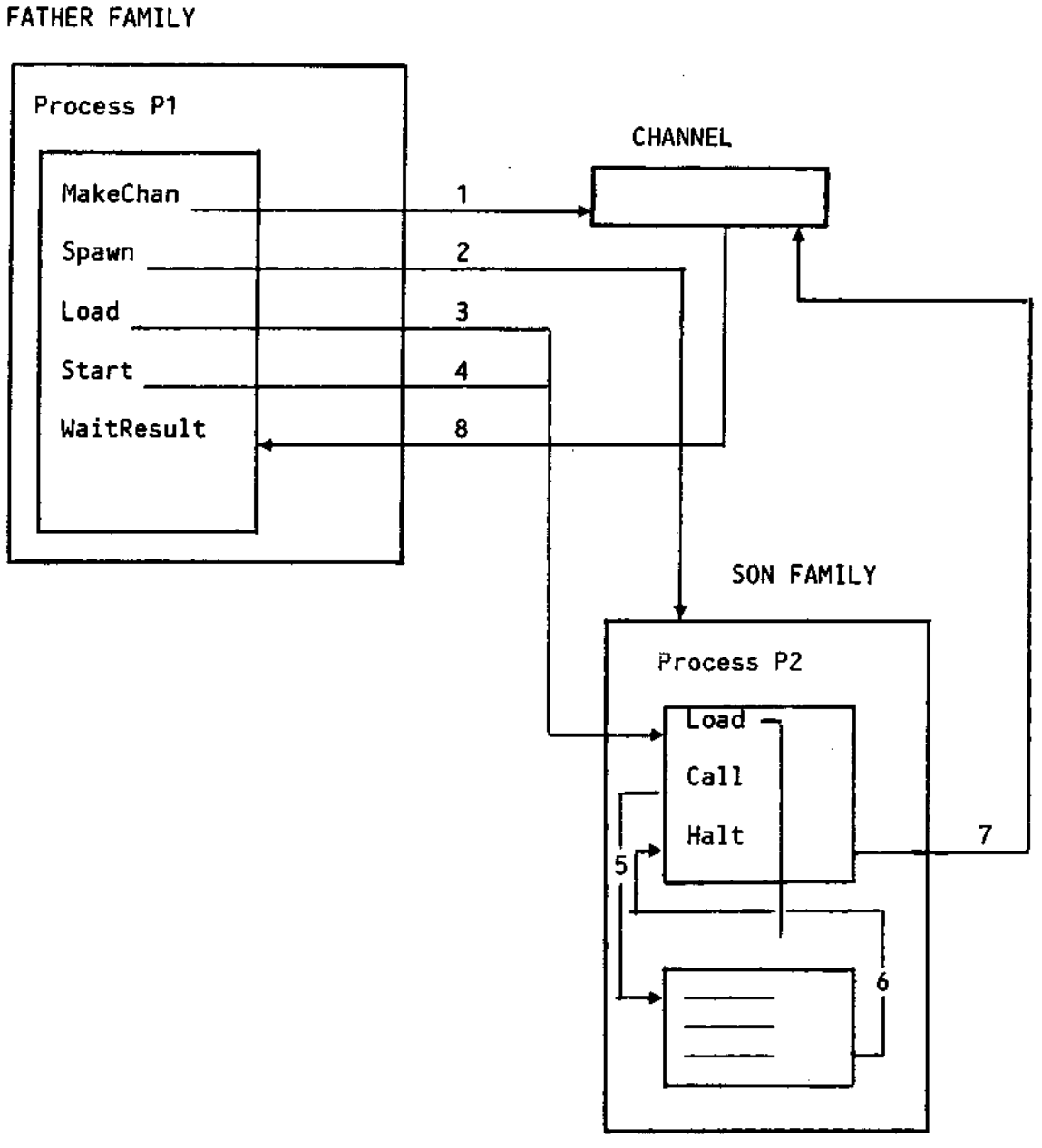


Fig. 3-11 Communications between Families Using a Channel

MANAGEMENT OF CO-RESIDENT PROCESSES

The primitive **Fork** creates a new process that is co-resident with (in the same family as) the creating process. In the program, two execution streams emerge from a **Fork** operation: one is the process that performed the **Fork** ; the other is the new process. The new process shares all non-stack data with the original process, as well as all its private monitors. A **Fork** operation causes a new STACKSEG stack segment and a new STACKSEG - 1 segment to be made, and a new process to be started at the statement following the Fork call. The value returned from the **Fork** is different for the parent and child. The parent gets the identifier of the new process whereas the child gets the Pascal null pointer nil. The processes can thus find out whether they are the parent or the child and can decide accordingly which statements to execute.

The primitive **Die** is invoked by a process that wishes to die. If the process invoking this primitive is the only (or the last) one in the family, **Die** will have the same effect as a **Halt** with result type set to zero and a null result string. The same effect is achieved when a process terminates the execution of its program by reaching the end of its outermost block.

Note that any of the co-resident processes could invoke the **Halt** primitive that would return a (possibly null) result to the controlling process and terminate all the processes in the family.

The **Join** primitive suspends the invoking process until the specified process dies. This accomplishes the synchronization of two co-resident processes on termination of one of them.

The following figure illustrates the use of primitives for handling co-resident processes, including the use of all those primitives relative to handling processes and families.

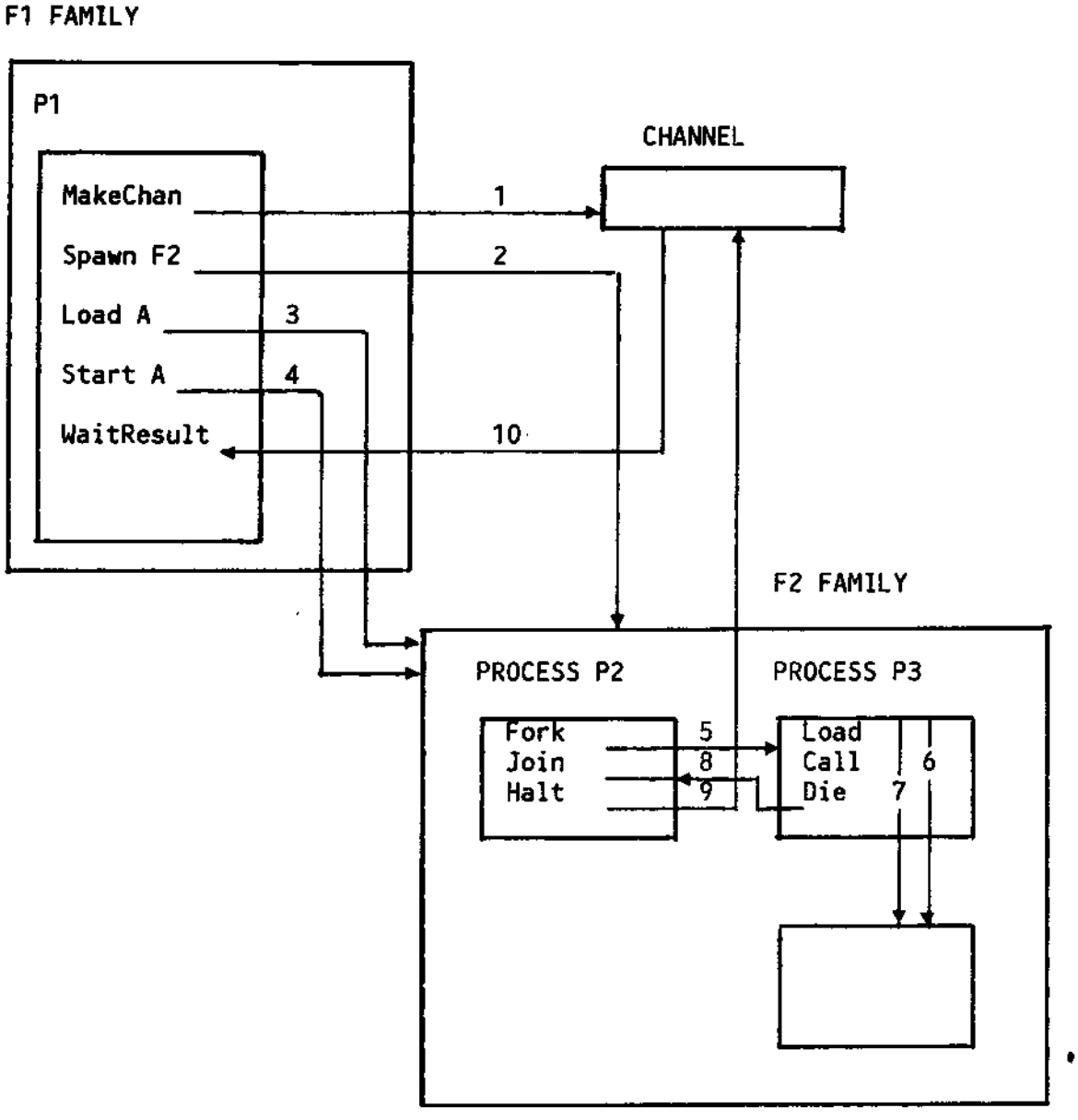


Fig. 3-12 Creation of Families and Processes

PARAMETER PASSING

Parameters may be passed to a program either explicitly or implicitly.

Passing Parameters Explicitly

As already described in the Section "Family and Program Activation and Termination", a program is allowed to receive in input positional or keyed parameters, in form of character strings, long integers and system object identifiers.

A called or started program is able to receive its imput parameters by accessing the packed array of characters **lineParam** that is predefined in the language. Note that from the point of view of the new program, everything appears as though the program itself were a subroutine of the operating system (or of the calling program) and were called with a predefined packed array of characters as its input parameter. The implicit definition of the predefined array is as follows:

```
var lineParam: packed array
   [lineParamLb..lineParamUb: integer] of char;
```

Formatting of primitives for parameter passing makes it possible to:

- initialize the array (Init primitive)
- write parameters to the array
- read parameters from the array
- check the filling state of the array (ReadCount primitive).

There exists a particular writing primitive relative both to the type (positional, Keyed) and the format (character string, long integer, system object identifier) of the parameter to be passed. For example, PutKeyLOng writes a keyed parameter of long integer format. The same facility is also available for the reading primitives.

Passing Parameters Implicitly

Another way of passing parameters to a family is indirectly, via the contextId parameter of the Spawn primitive. In fact, the **Context** primitive is used by a process to retrieve the contextId value its family was given at **Spawn** time. Such an object identifier allows the family to receive information from the spawning family or from any other family that knows the same identifier. The precise kind of interaction depends on the type of identifier that is shared, on the kind of operations it offers, and finally on the conventions the sharing families agreed upon.

The information stored by Grandpa in the context of the families that it creates is described in the Chapter "Activating User Activities". As already mentioned in the Chapter "File System Management Interfaces", the context is a file belonging to the CONTEXT$$ directory.

The identifier returned by **Context** is then used to perform file system Connect operations to such files as standard input, standard output, etc. In some cases, when a new family is spawned, the parent wants the child to inherit its I/O context; then **Spawn** can be passed the identifier returned by **Context** directly.

SECURITY

The overall MOS Security system has been described in general in the Chapter "File System Management Interfaces". The following sections therefore only provide additional information regarding PMM security features.

The PMM provides protection of the objects that it handles itself (families, processes, channels, active programs). As it is with these objects that the user executes his processing in the system, it is thus necessary to enable only authorized users to access them.

The PMM assigns to these objects the identity of the user who has requested their creation, and who becomes their owner. PMM then inhibits operations on objects not owned by the user who is attempting to use them.

It operates as follows:

- It assigns as owner of a new family/channel/process/active program the owner of the calling process. This assignment is made at the time of creation of the object (by Spawn, MakeChan, Load, Start and Call).

- It only allows an object to be destroyed by a process owned by the user that created it.

- It activates a program only if the user who requested its activation has the right to execute the file system object containing the load module or the program directory to be activated.

- It provides the SetIdentity primitive, which enables a process to permanently change its identity. For further details refer to the Section "Security Primitives" in the Chapter "File System Management Interfaces".

- It recognizes a privileged identity: that of the super-user allowed to carry out any operation on any PMM object.

As regards assignment of identity to processes and families, note that the first process created in a family receives the identity of the owner of the family; subsequent ones receive the identity of the owner of the creating process.

Bearing in mind that in its turn the family receives the identity of the creating process it is evident that the identity of the family, and that of the processes belonging to it, may differ only in the event of there having been a permanent or temporary change in identity within the family.

The identity of the family is known as the "real identity"; that of the process the "effective identity".

The PMM security primitives are briefly described below:

SetIdentity is available only to the super-user. It assigns permanently the identity specified by the primitive to the current process.

GetIdentity returns the identity of the current process.

SetDefAttributes specifies to the PMM the default access rights that the family assigns to the files created by it.

DYNAMIC LINKING

This mechanism enables a load module to call - at run time - procedures that are not physically included in it at link time, and are shared by all the families using the system. This is the type of link used in MOS for calling basic and environment (e.g. MTS) services.

In the case of calling a service belonging to the calling l-module, both linkage editors effect a jump to an address expressed in the form:

(segment number, offset in the segment context)

For a service linked dynamically, however, the linkage editors effect a jump to an address expressed in the form:

(l-module number, procedure number)

---

where

l-module number — (called m_code) is an integer that uniquely identifies within the system the l-module exporting the procedure. Each system module (File System Management, PMM, ...) has its own m_code.

procedure number — (called p_code) is an integer that uniquely identifies within the system the called procedure in the context of the exporting l-module

The pair of values (m_code, p_code) is passed to the linkage editor by the interface files (or libraries); more precisely, if a program calls the "AAA" procedure, the interface file contains the statement **jump (n1,n2)**, where n1 and n2 are respectively the m_code and p_code of procedure "AAA". Knowledge of these values allows the system to locate, at run time of the application, the pair of values (segment number, offset), using the following tables:

1. **p_table,** (one for each load module that exports global procedures) containing the address of the procedures concerned in the load module. It is the task of the load module to construct this table and communicate it to the system. For further details, refer to the Chapter "User Package".

2. **m_table**, which defines the correspondence between the m_code of an l-module and the address of the p_table associated to the module. Each table entry contains the address of a p_table.

The figure below illustrates the addressing mechanism that the system uses at run-time for a dynamic link.

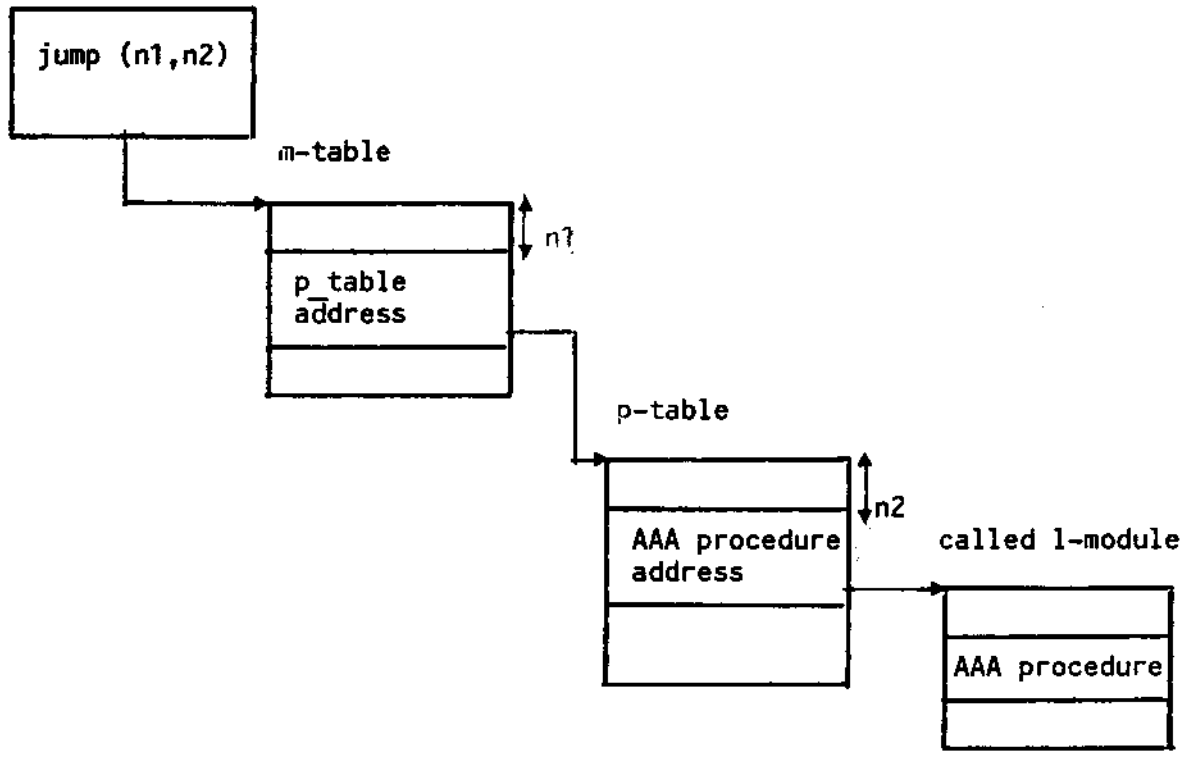


Fig. 3-13 Example of the Dynamic Link of Procedure AAA

Only one m_table exists in the system, created during the system generation phase; at run-time this may, however, be substituted by another m_table (which may specify new p_tables). At any moment the system knows only the current m_table.

The PMM provides a number of primitives for manipulating m_tables and p_tables.

MakeMTable duplicates the current m_table.

SwitchMTable exchanges the current m_table with that provided by the program.

SwitchPTable switches a p_table by modifying an m_table entry.

MergePTable manipulates a p_table.

FAMILY AND PROCESS IDENTIFICATION

The **GetMyId** primitive is used to retrieve the identifier of the calling process. Notice that this identifier is distinct from the one received at **Spawn** time since the latter is the identifier of a family whereas the one returned by this primitive is the identifier of a process created **via Start or via Fork.**

There is no primitive for obtaining the id of the running family. A similar facility is, however, provided. When a family is created, it is assigned a default "name": its type is T_systemId, related to the identifier of the channel the family uses on completion, to the "user code" specified at **Spawn** time, and finally to the system where the family runs (it is relevant only for distributed systems). Note that two family names can be identical only if the families received the same channel identifier and user code and run on the same system.

This name can be retrieved by a process that invokes the **GetMyName** primitive.

The default name assigned to the family at Spawn time can be changed by calling **SetMyName.**

TIMING

There are three timer-related primitives provided by the PMM.

SetTime allows the caller to initialize the system clock to the correct time. It takes the current time expressed in seconds since 00:00:00, January 1, 1982. Normally this primitive would be invoked only once at system initialization time.

Time returns the current time in seconds since 00:00:00, January 1,1982. Note that system utilities are available to map a number of seconds into either a printable string or into a Pascal record that breaks down the number of seconds into a more manageable date format. Also the inverse mappings are provided (Calendar primitives).

Sleep is a primitive that allows the caller to specify a number of seconds as its input parameter and suspends execution of the calling process for the specified amount of time.

CALENDAR

These refer to dates and times. The current time is returned in the form of seconds, or by strings made up of hours, minutes and seconds. The date is given in both the Italian and English format.

EXCEPTIONS AND SOFTWARE INTERRUPTS

This subsection discusses the occurrence of unexpected events and the way they should be handled. The use of such primitives should be avoided as they conflict with the use of a high-level language. No guarantee is given that such primitives will be maintained across different releases of the PMM.

Two types of unexpected event may alter the sequential execution of a program, as follows:

1. **Exceptions.** Such events occur synchronously with the execution of a program. They may be caused by faults (segment access violations, attempts to execute privileged instructions while in non-privileged hardware mode and so on) or by explicit invocations of the PMM **Exception** primitive.

2. **Software Interrupts.** These are asynchronous to program execution and are caused by the invocation of the PMM **Interrupt** primitive, specifying the identifier of a family distinct from that of the caller.

Note that the MOS exception handling mechanism is not intended as a substitute for language-based exception handling, since some of the relevant issues for such a scheme can only be dealt with in the context of a language implementation.

When the run-time support of the PASCAL+ language executes a "program", it accepts responsibility for setting an exception handler of its own which handles exceptions without activating any handlers defined by the program. When effecting a "module" this does not happen and thus on occurrence of an exception the handler defined by the user, if present, is activated.

In what follows, software interrupts will often be referred to as "interrupts"; the user should, however, bear in mind that such interrupts are not related to hardware interrupts.

When a new family is spawned, no exception or interrupt handlers are associated with it. The user program itself (or its run-time support) is responsible for defining its own handlers. The objective is that each program should be able to perform its own clean-up before terminating so that the permanent user context altered by the program execution may be left in a consistent state. If no handlers are defined, exceptions and software interrupts will cause the process to be destroyed.

Both exception and interrupt handlers are executed only when the process is out of the system state and is executing user code.

Exceptions always trigger the execution of the exception handler as soon as the process reaches the user state. They can never be disabled since they are really synchronous with the processing and should operate within the context in existence at the time they occur.

The exception handler is passed a code that specifies the reason for the exception. This is either a system-generated code (for segment faults and

similar) or it is the code specified to the **Exception** primitive as an input parameter.

The Interrupt primitive makes it possible to send two software interrupt types. The interrupts of both types bring about the suspension of the family that is the object of the interrupt, and passage of control to the interrupt handler. When the latter has terminated execution two cases are possible:

- for type 1 interrupts, termination of the suspended family

- for type 2 interrupts, resumption of activity of the suspended family, starting from the point of suspension.

Handling of type 1 interrupts should therefore terminate with execution of the Halt primitive. In the event of this not occurring, or if the user has not defined an interrupt handler, the system itself will force execution of the termination function in the context of the eldest son of the interrupted family.

A type 2 interrupt has no effect on a family that has no interrupt handler; otherwise it will be executed by the process specified in the context of the primitive, or the eldest son.

The **Interrupt** primitive takes two input parameters:

1. The family-id of the family to be interrupted or, only for type 2 interrupts, the process-id of the process - belonging to the family to which the interrupt was sent - which is required to execute the interrupt procedure.

2. An interrupt code to be passed to the interrupt handler and the interrupt type, class and code.

Interrupts may be regarded as a way of communicating elementary information across family boundaries. Therefore, the code they pass requires conventions to be established between the "interrupter" and the "interruptee", in order for the latter adequately to respond to a given request.

Typically this mechanism is used as follows: a process that "spawned" and "started" a son process causes a "soft" termination of its son by sending an interrupt to it; the son may then close its own operations as best it can by providing the appropriate "interrupt handling procedure" when its program is initialized.

Another situation where the user might want to use interrupts is when it is necessary to inform a family that some "significant event" has occurred without devoting a process in the family to monitoring the "events". In fact this is saving is an illusion, because the PMM has to support the asynchronous triggering of the interrupt handler activity, which in this case must be general purpose. Moreover, the event which is communicated to the controlled family and gathered by the interrupt handler must in turn be communicated to the ordinary family processes, and monitors must be introduced for this purpose. Therefore, there is no

actual saving for the programmer, and it is much safer to use the monitors right away rather than have the "interrupter" communicate with the "interruptee". The PMM therefore forbids the use of type 1 interrupts in situations like the one above. The interrupt handler activation must terminate by issuing a **Halt** call.

A single interrupt handler may be associated to each family and all the mechanisms thus operate on the family as a whole; no process in the family is different from the other processes in the same family, as far as the interrupts are concerned.

Interrupts are enabled when a new process is started in a family. However, the primitives **Disable and Enable** are available to disable type 1 interrupts selectively within critical code sections. Such primitives handle a counter for the family; this means that if N consecutive invocations of **Disable** are performed in the family (either by the same process or by different ones), only after the n-th invocation of **Enable** are interrupts effectively enabled again.

No form of disablement is provided for type 2 interrupts. Execution of an interrupt handler cannot be interrupted by arrival of an interrupt of the same type. A type 1 interrupt interrupts handling of a type 2 interrupt, but not vice-versa.

It is important to realize that while interrupts are disabled, only the last of a series of similar interrupts is saved, since it overwrites all the previous ones.

As soon as interrupts are effectively enabled again the pending interrupt, if any, immediately triggers execution of the current interrupt handler.

The user makes available an exception and/or an interrupt handler through the **SetHandler and ResetHandler** primitives.

SetHandler takes two input parameters:

1. A Boolean that specifies whether an exception handler (TRUE) or an interrupt handler (FALSE) is being handled.

2. A procedure parameter that is the name of the handler (this procedure must be global to its enclosing Pascal+ compilation unit).

The primitive returns a pointer to either an exception handler (first parameter set to TRUE) or an interrupt handler (first parameter set to FALSE). The pointer is null (nil) when no previous handler of the same type was available for the family; otherwise, it points to the previous handler.

ResetHandler also takes two input parameters:

1. The first parameter is the same as for **SetHandler.**

2. The second parameter is a pointer to an exception or software interrupt handler, as returned by **SetHandler.**

The value returned by **SetHandler** , in combination with the **ResetHandler** primitive, allows users to define arbitrarily complex recovery policies even where there is a great deal of nesting among separate programs.

To clarify the previous paragraph, let us think of a started program ("prog1") that calls another program ("prog2") that in turn calls a third one ("prog3"). Since each of the three programs is compiled and linked separately from the others, each program may (and should) ignore the implementation details of the others and only know the kind of activity the called program is supposed to carry out. As a consequence, the exception and interrupt handlers for "prog1" may not be appropriate for the context of the others.

The solution is fairly straightforward: "prog1" defines its own exception and interrupt handlers; when "prog2" gains control, it first calls **SetHandler** to define its own handlers and stores away the pointers returned by this primitive; "prog3" does exactly the same. When "prog3" is about to **Halt** , thus passing control back to its caller, it calls **ResetHandler** twice, with the exception handler and interrupt handler pointers it had saved. This resets the handlers to those provided by the outer context ("prog2"). Finally, "prog2" does the same with respect to the handlers of "prog1". Thus the initial context is entirely restored.

Note : It is good programming style to save the pointers to the previous handlers when defining new ones and to restore the previous handlers on program termination. This makes the program totally transparent with regard to the way it is activated **(via Call or via Start).** In fact, although no handlers currently exist for started programs, called programs run in the context set up by the caller and this generally implies the existence of previously defined handlers.

In writing exception handlers it must be clear that some situations immediately require the death of the process and possibly of all the co-resident processes as well. For instance, in cases of segment violations and the like, the only sensible way out is that of trying to restore a consistent context and then performing a **Halt** invocation. In other cases, other forms of recovery might be possible. For instance, if the Pascal run-time causes an exception, specifying the code of a heap overflow, the exception handler might try to solve the situation by invoking **SetSegment** to extend the heap area and then returning to the caller. This could allow the run-time package to retry the failed heap allocation.

OTHER PRIMITIVES

Both **Suspend** and **Resume** primitives take a family identifier as their parameter.

Suspend suspends all the processes executing within the family. The suspension may not be immediate since for processes that are executing system primitives, the suspension will be effective only when they reach the user state again. (Note that the system primitive aborted by the **Suspend** primitive will return an error code denoting that it received a software interrupt).

Resume immediately activates all the processes at the instruction pointed to by their program counter.

GetCollection returns statistical information regarding:

- PMM objects
- PMM operations
- physical memory

SUMMARY OF SHELL COMMANDS FOR PMM OBJECTS

Certain Shell commands concern families, others the system clock.

A list is given below of these commands, grouped on the basis of their functions. The command name is followed by that of any PMM primitive carrying out analogous functions, and a brief description of the command.

COMMANDS FOR HANDLING FAMILIES

KILLFAM	Disactivates one or more families (including those in background).
SHFAM	Displays information regarding families active in the system (including those in background).

COMMANDS FOR HANDLING THE SYSTEM CLOCK

SETDATE	Modifies the system date and time.
SHDATE	Displays the system date and time.

MISCELLANEOUS COMMANDS

RESUME	Resume	Resumes a suspended user program.
SUSPEND	Suspend	Suspends a family in execution.

PMM PRIMITIVES: LOGICAL USAGE OUTLINES

The following examples only show the logical arrangement of the primitives and their essential parameters. They cannot, therefore, be utilized directly in programs.

LOADING AND ACTIVATING A PROGRAM OF THE SAME FAMILY

The program contained in the /IPL/AAA/PROG file should be loaded and activated by a process belonging to the same family.

```
Connect(Context,localroot,x_,...,Idroot,...)
   (* effects the connection  to the local root whose systemId        *)
   (* can be found in the context of the process. The context is      *)
   (* accessed using the Context primitive.                          *)

Connect(Idroot,IPL/AAA,x_,...,Iddir ,...)
   (* effects connection to the directory AAA whose systemId is       *)
   (* returned to Iddir.                                             *)

Connect(Iddir,PROG,x_,...,connId,...)
   (* effects connection to the program PROG and returns             *)
   (* the systemId of the connection to connId.                      *)

Load(progId,familyId,connId)
   (* loads the program with the connection number connId into       *)
   (* the address space of the calling family as familyId=NIL.       *)
   (* Returns the program_id to progId.                              *)

Call(...,...,...,...,progId,...)
   (* activates the program that has just been loaded. For simplicity, *)
   (* the  mechanism for  passing parameters between the calling and  *)
   (* the called program is not shown.                               *)

Disconnect(connId)
    (* closes the connection to the file containing the program.     *)

Disconnect(Iddir)
    (* closes the connection to the directory.                       *)

Disconnect(Idroot)
    (* closes the connection to the local root.                      *)
```

ACTIVATION OF A PROGRAM OF A NEW FAMILY

The program contained in the /IPL/AAA/PROG file should be loaded and activated by a process belonging to another family.

```
Connect(Context,localroot,x_,Idroot,...)
   (* effects the connection  to the local root whose systemId        *)
   (* can be found in the context of the process. The context is      *)
   (* accessed using the Context primitive.                          *)

Connect(Idroot,IPL/AAA,x_,Iddir)
   (* effects connection to the directory AAA whose systemId is       *)
   (* returned to Iddir.                                             *)

Connect(Iddir,PROG,x_,idprogcon,...)
   (* effects connection to the PROG program and returns the          *)
   (* systemId of the connection to idprogcon.                       *)

MakeChan(chanId)
   (* obtains the identifier of a new channel.                       *)

Spawn(famId,chanId,Context,...,attribute)
    (* creates a new family having the attributes specified,         *)
    (* returns its system identifier to famId, and establishes the   *)
    (* communication channel chanId with it.                         *)

 Load(progId,familyId,connId)
   (* loads into it the program with the connId connection identifier *)
   (* and returns progId.                                            *)

 Start(progId,...,...,...)
   (* activates the program that has just been loaded. For simplicity, *)
   (* the mechanism  for passing parameters between the  calling and  *)
   (* the called program is not shown.                               *)
 .
 .

 Disconnect(IdProgCon)
    (* closes the connection to the file containing the program      *)
    (* whose systemId is in IdProgCon.                               *)

 Disconnect(Iddir)
    (* closes the connection to the AAA directory.                   *)

 Disconnect(Idroot)
    (* closes the connection to the local root.                      *)
```

CREATION OF A CO-RESIDENT PROCESS

```
procedure procfork;
var P:T_id;

Fork(P)                              (* creates co-resident process.       *)
if P<>NIL then  begin
                .                    (* this part of the code is executed  *)
                .                    (* only by the Forking process.       *)
                Join (P)
                end
           else begin
                .                    (*  this part of the code is executed *)
                .                    (*  only by the Forked process.       *)
                Die
                end
```

4. WORK STATION MANAGEMENT INTERFACES

After briefly summarizing the Work Station Management (WSM) characteristics, this chapter describes the primitives for accessing the various peripherals forming a work station: terminal, printer, badge reader/writer, cheque reader/writer, PIN Pad and cash adapter.

This chapter also includes a list of Shell work station commands; these commands are described in full in the SHELL Commands, Reference Manual.

For complete details on the WSM primitives, the reader should consult the PMM and Driver Primitives Reference Manual. This Manual also contains a number of examples of WSM primitives usage.

INTRODUCTION

The WSM makes it possible to connect, locally or remotely, the following types of WS:

- Olivetti WS (KDC, WSL1)
- Personal Computers (M24), ...)
- VT100-like terminals (WS 584, ...).

The services that WSM provides for these WS are as follows:

- Logical access to the individual WS peripherals.
- Global virtualization of WS.

ACCESSING PERIPHERALS

As the access method provided is of the logical type, it hides the various modes of connecting work stations to the system and the different physical characteristics of the WS peripherals, although in most cases retaining the possibility of using the peripheral commands. This happens, for example, in the case of terminals.

Device independence is assured in exploiting the various features of any class of components other than terminals; this independence is lost in the case of terminals, because of the various connecting modes and of the various characteristics of the terminals themselves.

VIRTUAL WORK STATIONS

These have been implemented to enable a number of applications to share the same physical work station; each one still has complete visibility of the WS.

Note that the term "application" means the set of programs belonging to an application environment (e.g. the Shell environment).

Virtualization of WS is available only for Olivetti WS (KDC and WSL1), and is based:

- on the virtualization of the terminal
- on implementation of the mutual exclusion mechanisms for regulating access to other WS peripherals.

Virtualization of a terminal is based on assignment of the entire screen and the keyboard of the physical terminal to the application associated to the virtual WS required by the operator (using the appropriate function keys).

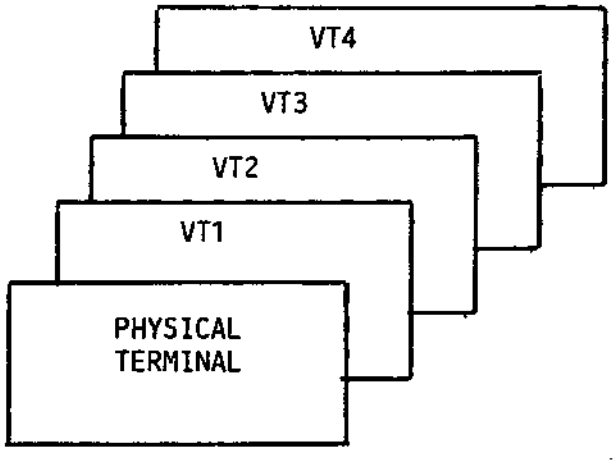
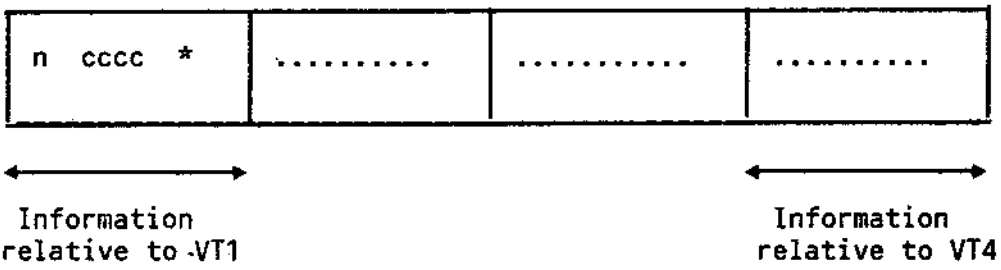


Fig. 4-1 Associating Virtual Terminals (VT) to a Physical Terminal

Note also that the term "application associated to object x" (for example WS x) means an application whose interactive programs find the system identifier (wsid) of object x (e.g the WS x) stored within its own context.

The WSM keeps track of the status information of virtual terminals and displays it to the operator at his request (using appropriate function keys). The following information on the status line of the physical terminal of a virtualized WS are displayed by WSM:



where:

n is the identifier of the virtual terminal (character 1 ... 4)

cccc is the name of the current activity associated to the virtual terminal addressed (4 characters)

* specifies that the activity associated to the terminal is awaiting input.

If a message has been sent to a VT (that is, there is a message attention condition) the WSM causes the corresponding VT field in the status line to blink.

As regards sharing WS peripherals other than terminals, note that:

1. The badge reader/writer, the PIN pad and the cheque reader/writer cannot be shared between several physical WS, but can be shared between the virtual WS based on the same physical WS. Thus applications associated to such virtual work stations share the above-mentioned devices.

   Given the physical characteristics of these devices, sharing is of the mutually exclusive type. More precisely, when a virtual WS requests to use a device already being used by another virtual work station based on the same physical WS, the WSM queues the request until it can be handled.

   Any request from an application associated to another physical WS is rejected, as well as any request coming from processes belonging to the same application using the device.

2. Unlike the badge reader/writer, the PIN pad and the cheque reader/writer, the printer and cash adapter may be shared between a number of physical WS, and thus between any virtual WS based on them. Also for these peripherals, what has been stated above regarding the mutually exclusive use of peripherals by applications applies.

WINDOWS

Besides the WS virtualization mechanism described above, for terminals there exists a further possibility for virtualization, which enables two programs belonging to the same environment to share the same terminal, regardless of whether or not it belongs to a virtual WS.

This is achieved by dividing the screen into two subsets of contiguous lines, and associating the keyboard to one or the other of the two subsets. This virtual terminal is referred to as a "window" throughout the manual; the term "virtual terminal" being intended for terminals belonging to virtual WS. The features concerning windows must be requested at run time using the appropriate primitive (Split); those features that concern virtual terminals must be requested at the time of configuring the WS.

IDENTIFICATION OF WORK STATIONS

The WSM uses a system identifier (called wsid) for identifying active WS (whether virtual or not), and active windows.

Every program belonging to an environment finds within its context the WsId(s) of the available window(s). If the splitting of a terminal has not been requested either by a program or by its antecedents, WSM considers the terminal to be equipped with just one window, having the same size as the screen.

If virtual WS features are not requested at the WS configuration time, the WSM considers the physical WS as equipped with just one virtual terminal, which coincides with the physical one.

If both the functions relative to the virtual WS and windows have been requested for a physical WS, up to 8 WsIds associated to these WS can exist.

All the WS configured in a system, whether active or not, are identified using a pre-defined symbolic name, as follows:

VTxx :which identifies the local virtual WS xx.

TTYx :which identifies the local non-virtual WS x.

RTxx :which identifies the remote WS xx.

By using the above symbolic names an application can ask the system the corresponding system identifier (WsId), a knowledge of which makes it possible to access any work station of the system (not only those associated to the particular environments concerned).

WSM PRIMITIVES

The Work Station Manager (WSM) primitives allow the programmer to access all the components of a work station and to exploit the different characteristics of each of them. One group of primitives is provided for each component class. There thus exist Terminal Access primitives, Printer Access primitives, ...

In addition to specific components primitives, general management primitives are provided which apply to the whole work station. The primitives of the first type are provided by the WSM component which controls the specific work station device (Terminal Driver, Printer Driver, ...); that of the second type are provided by the Terminal Driver.

The WSM primitives are used by the FSM when of requiring access to a terminal or printer.

General management primitives are discussed first, followed by specific-component access primitives. For details on the WSM primitives, the reader should consult the PMM and Driver Primitives Reference Manual Parts 2, 3 and 4. More specifically, reference should be made to:

- Part 2, for primitives supplied by the Terminal Driver
- Part 3, for primitives supplied by the Printer Driver
- Part 4, for primitives supplied by the banking peripheral Drivers (Cash Adapter Driver, ...).

WSM GENERAL PRIMITIVES

The **ReadInfoWs, WriteInfoWs** and **ReadInfoUnit** primitives allow the programmer to read/write information on the physical characteristics of a work station (whether physical or virtual) identified by a WsId. The first two are related to the terminal, whereas the third one is related to the other work station devices. These primitives have the usual form of MOS system primitives. Their parameters might be considered somewhat more complicated than usual because the configuration of the work station must be specified and the characteristics of each component are more complex than those of other MOS objects. In particular, the view of the terminal is extensive.

The Split primitive allows the creation of two windows on the original terminal (whether virtual or not), or allows the opposite to be performed depending on the **splitRow** parameter. This parameter allows the programmer to specify where the terminal screen should be split. The original terminal will then have a screen which has shrunk, and the remaining area will belong to the newly created window. The Split primitive returns in this case the WsId of the newly created window.

The **splitRow** parameter also allows the programmer to "kill" a window and to return its screen to the window whose screen is "above" the one that has been killed.

If the Split primitive is not used, then the WSM assumes the existence of only one window with a full screen address space and the WsId of the work station.

The **GetWSName** primitive returns the work station name and the lower window identifier.

The **GetTTYId** primitive returns the system identifier of the upper window of the work station whose symbolic name is provided.

The dynamic re-configuration of a number of work station devices (printers and cash adapters) is made possible by the **ReconfWS** primitive. It is possible in this way to modify the configuration of the WS established at the time of configuring the system.

The **WriteInfoVT** makes it possible both to set the message attention condition of a VT, and to write the name of the current activity associated to a VT, in the status line (see the Section "Virtual Work Stations"). When WriteInfoVt is not used, the name of the environment selected by the operator in the context of the GrandPa menu is assumed by WSM as the current activity name. For details on the Grandpa menu, see the Chapter "User Activity Activation Modes", Section "Activation/Disactivation of Application Environments".

For the **Logoff** primitive, see the Chapter "MOS Application Services".

TERMINAL ACCESS PRIMITIVES

The terminal access primitives provide the first and most important tool for directing the terminal. This access method can be classified as sequential, since the information is written or read to/from a position on the terminal which is maintained by the terminal itself. The records that are written and read are called **strings** , and their structure has a particular meaning to the terminal. The reader should be wary of applying to such strings concepts derived from the EDP sequential file concept, as they are not entirely analogous.

The similarity between the WSM terminal access method and the FS sequential files is very strong in the case of reading, as it does not make sense to try and read previously input strings nor to skip ahead of the string being input by the operator. However, the similarity is weak in the case of writing, because the string is output starting from the position of the **cursor** , and this position can be changed explicitly by

the user. Cursor control is one of the facilities that can be exploited by "writing" output strings formed in a specific way onto the terminal. In other words, the output string is seen by the WSM as a **control string** for the terminal, which may contain, as a special case, only data to be written on the screen from the cursor position onwards.

The terminal access primitives provide a transparent interface both for the keyboard type and the video characteristics. They are used by the software environments, which generally enable the user to make full use of all the WS that can be connected to MOS. Further information on this subject can be found in Appendix C.

The access method for the terminal provides primitives for opening, closing a session on a window (**OpenTm, CloseTm**), and reading from and writing to a window (**ReadTm** and **WriteTm** primitives).

It is also possible to obtain the description of the window (**ReadInfoTm** primitive), to change the dimension of the type-ahead buffer (see the "Reply String" Section), to define a timeout for the ReadTm function (**WriteInfoTm** primitive), and to set particular reading conditions (see "Raw" status in the "Reply String" Section).

If a terminal (whether virtual or not) is split into two windows, concurrent output activities will take place on the physical screen, since the processes may write on different windows concurrently, each with its own cursor. Input activity, on the other hand, is of a sequential nature, since there is only one physical keyboard. The operator will thus have a specific key (Change_Window) which allows him to bind the keyboard to a particular window and to input a string that the process receives via the **ReadTm** primitive.

Concurrent access to the window by more than one process can be controlled by the programmers, by opening the terminal either with an **exclusive** or with a **none** lock.

The TTY interface provided by the terminal access primitives is now described.

The TTY Interface

This interface is provided by the Terminal Access Primitives; it is oriented towards string input/output (one string at a time) and includes facilities for scrolling screen areas. The screen of a TTY work station coincides with the associated window. The Terminal Driver supports the notion of **active position,** which coincides with the cursor position. Most of the terminal's commands make implicit reference to the active position. The concept of **scrolling region** is provided, and specific commands are available to define it. Once this region has been defined, the program can enable (or disable) scrolling. When enabled, scrolling is performed when a character is written in the last position of the scrolling region, or a line feed character is sent to the terminal with the active position on the last line of the scrolling region. In addition, the program can explicitly scroll up one specified area of the screen, and control whether the scrolling involves the attributes or the attributes remain unchanged. In this way, either areas that are shown in

reverse remain in reverse (no attribute scrolling), or strings that are in reverse remain in reverse after scrolling (attribute scrolling).

The Control String

The TTY terminal is controlled by means of the commands in a string sent by the **WriteTm** primitive (control string).

The terminal identifies the control strings by syntax, i.e. all commands begin with an **ESCAPE** character. Following the **ESCAPE** , the various commands have different string syntax which in many cases enables variable-format commands.

The **Control String** is scanned character by character looking for ESCAPEs. Every character that does not belong to a command is displayed on the screen in a position and with attributes that depend on the state of the terminal. When an ESCAPE is found, the following control string is scanned to find one of the terminal commands. When a command is successfully recognized, it is immediately executed, and the search for the next command starts. Commands are available to display a given string transparently (as it contains ESCAPE characters).

The characters which do not belong to the command strings are displayed on the screen. Some of the characters have such a visual effect that they might be considered as commands. Examples are line feed, carriage return, tabs, form feed and bell.

The Control String handling described above makes it possible to use the terminal in both "sophisticated" mode (using control strings containing commands) and in "simple" mode, using control strings containing only data. In the latter case the functions provided by the terminal driver (scrolling, ...) enable the terminal to be treated as a sequential-type support.

Control String Commands

The WSM allows use of two command sets, Olivetti and VT-100 like. They are, respectively, a superset and a subset of the industry standard commands. For Olivetti terminals or PCs used as work stations, both sets can be used. The VT-100 like set permits easier conversion of existing application programs, written for industry standard workstations, in programs to be used on Olivetti workstations. Application programs written for industry standard workstations are therefore compatible, from the functional point of view, with Olivetti terminals / PCs.

Both command sets normally allow the following:

- Positioning of control cursor.
- Setting/resetting the visual attributes.
- Turning the LEDs on/off, and controlling tabulation.

- Specifying the graphic set to be used.

The main sophistication of the Olivetti command set concerns the visual attributes.

For VT-100 like terminals, only the set described in the reference manual of these terminals must be used, as the WSM does not provide any interpretation function.

The Reply String

The **Reply String** conveys to the calling process the last characters typed in by the operator. Pressing the keys causes sending ISO characters to the program (normally two in the case of function keys; one in the other cases). The characters are made available to the process in "blocks", i.e. they are grouped in sequence according to some criterion which is under process control.

It is possible, using the **dataSize** parameter of the ReadTm primitive, to specify if a string of characters or a single character is to be received. Different interpretation functions for characters input on the window are provided by the ReadTm primitive. The interpreted (recognized) characters are not passed to the user process.

If the window is in the normal (default) status and dataSize>1, the **Reply String** consists of the characters the operator typed in, terminated by a line feed character. The operator must terminate the input characters with the line feed key and cannot enter more characters than requested by the application program (including the line feed). The interpretation functions are extensive; e.g. characters corresponding to the Hard Copy, End of File keys are not returned to the program. The WSM performs the echo; the line feed key is echoed by a line feed character followed by a carriage return.

If the window is in normal status and dataSize=1, only one character at a time is returned to the program, and there is no echo. The interpretation functions are reduced. Neither is echo performed by the WSM when the window is in "Raw" status. In this case, the Reply String contains the values corresponding to every key which has been typed in by the operator, except for the Change Window key and the scroll control keys, which are already interpreted by the ReadTm. Using "Raw" status, the user can implement Editor-type programs different from those supplied by Olivetti.

Any characters typed in by the operator when no **ReadTm** is pending are stored in the type-ahead buffer and are returned to the program in the same order in which they have been typed in, using the mode specified by the program.

PRINTER ACCESS PRIMITIVES

A printer is accessed within a session that must be opened and closed via the **OpenPr** and **ClosePr** primitives.

A printer can be shared among several physical work stations, and each physical WS can be shared among any virtual WS based on it. Given the physical characteristics of the printer, only one session may be open at a time.

A printer is made up of several subdevices (platen, sprocket, front feed, ...), and one of these is the default subdevice. The default subdevice is set by the firmware and is one of the subdevices allowing printing operations on sheets or continuous forms.

The user is given the possibility at **OpenPr** time (via the access mode parameter) to select the default subdevice (default session), or he can specify to the system the use of a front feed subdevice.

In a **front feed** session, because of the fact that the printer can be shared, a particular interaction with the operator is required in order to actually establish a session. In fact, the use of the front feed requires the appropriate document to be inserted before the **OpenPr** returns. Several programs running for different operators may issue an **OpenPr** at the same time. The printer is equipped with a number of lights which show which operator has issued an **OpenPr** on the front feed. The operators notify to the Work Station Manager which one of them has introduced the document in the front feed by causing the corresponding light to come on. After the OpenPr, the following activities can be effected:

- For **default** sessions, printing operations can be carried out via the default subdevice, or another non-magnetic subdevice can be selected and printing operations carried out. Both operations can be executed via **WritePr**

- For **front feed** sessions, a specific front feed subdevice must be selected, after which reading/writing operations may be carried out on the document stripe, or printing operations may be performed on the document itself. **ReadPr** allows reading a document stripe, whereas **WritePr** may be used both to print characters on documents or to record data on a magnetic stripe.

WritePr enables commands and data to be sent to the printer. Also, it makes it possible, depending on the requirements of the application, to use directly the commands provided by the peripheral ("transparent" use of the peripherals), or to send to the default subdevice only data inter-mixed with appropriate linefeed commands.

By means of command character sequences, the program can access the subdevices, turn the lights on and off, and define the visual attributes of the characters to be printed. The Printer Access Method complies with the Olivetti Standard 12.

It is possible, using the **WriteInfoPr** primitive, to impose timeouts on completion of the printer operations.

BADGE READER/WRITER ACCESS PRIMITIVES

The badge reader is accessed within a session that must be opened and closed via the **OpenBg** and **CloseBg** primitives. Sharing of the peripheral is allowed only among the virtual WS based on the same physical WS. Only exclusive access to the badge reader is allowed and enforced. The specific badge track on which the subsequent read and write operations will occur is selected at the time of opening.

Data on the magnetic stripe of the badge is read and written via the **ReadBg** and **WriteBg** primitives, which completely hide the peripheral commands from the user.

WriteBg may be asked to verify what is written on the stripe.

Notice that the **ReadInfoBg** primitive must be used to check whether the previous **WriteBg** was successful.

CHEQUE READER/WRITER ACCESS PRIMITIVES

The cheque reader/writer is accessed within a session that must be opened and closed via the **OpenCr** and **CloseCr** primitives. The peripheral may be shared only between the virtual WS based on the same physical WS. Given the physical characteristics of the peripheral at the time of the Open, it is necessary to request exclusive use.

Data on the cheque is read and written via the **ReadCr** and **WriteCr** primitives, which completely hide the peripheral commands.

The **WriteInfoCr** primitive performs different actions on the cheque (i.e. returns it, ejects it, moves it backwards, ...), and writes on it.

PIN PAD ACCESS PRIMITIVES

The PIN pad is accessed within a session, which must be opened and closed via the **OpenPp** and **ClosePp** primitives. The peripheral may be shared only between the virtual WS based on the same physical WS. Given the physical characteristics of the peripheral at the Opening time, it is necessary to request exclusive use of it.

Handling of the PIN Pad is supported by the driver, but switching LEDs on/off and enabling/disabling the keyboard is left entirely to the application program.

The **WritePp** primitive allows sending of a sequence of hexadecimal commands to this device.

The **ReadInfoPp** primitive returns information from the PIN pad device.

Data are read from the PIN pad via the **ReadInfoPp** primitive.

CASH ADAPTER ACCESS PRIMITIVES

The cash adapter is accessed within a session that must be opened by the **OpenCa** primitive, and closed by the **CloseCa** primitive. During a session the user has a peripheral dedicated to itself, which can perform all the operations allowed for the type of session requested.

It is possible to work on the cash adapter opening a session in the following ways:

- PRIVATE mode : useful to perform off-line operations.

- TELLER mode : useful to carry out all the transactions involving note-dispensing operations on the Cash Adapter.

WriteCa allows sending note-dispensing commands. **ReadCa** allows reading status information concerning this type of peripheral device. **WriteInfoCa** allows carrying out off-line operations. **WriteCa** also allows sending hexadecimal commands to the device.

When designing an application program that uses a cash adapter, the possibility of sharing this device should be borne in mind. It may be shared among the virtual WS based on the same physical WS; it may also be connected to the Controller Unit (CU) in the following ways:

- One user on a single CU

- Two users on a single CU

- Two users on different CUs

To achieve connection in one of the above ways, the Cash adapter access method hides how the cash adapter is connected to the CU, leaving accounting information handling to the application program, which may do this in any way (e.g. using memory tables as the data area of the program itself in the one user-one CU case, or information stored in one (or more) controlled access file(s) in the second and third cases).

The first case is simple, and it applies to "automatic tellers" or to a work station where a dedicated peripheral is needed; moreover the Cash Adapter has a simple delivery throat.

The second case is more difficult to handle, but it applies to the most used configuration: two work stations where a single device is shared between two operators. In this case, the Cash Adapter has an orientable delivery throat.

The last case is the most complex both from the driver and the application program point of view. The driver must handle connecting cables/wires signals so as to prevent undesired access conflicts on the peripheral; special cables and special adjustment on the line controller of the peripheral are also required. The application program has to handle accounting information (total number of notes for every cassette, hopper status, number of cassettes, type of notes, ...), coming from different CUs.

SUMMARY OF SHELL COMMANDS FOR WS PERIPHERALS

The following terminal handling commands are available:

CLEAR	Deletes all the characters and visual attributes on the video.
ECHO	Displays the result of an expression.
SETTERM	Defines the video format; that is, the number of characters that can be displayed on it.
SHOW	Displays the string specified on video or on another system device (printer, ...)
SHTERM	Displays the video and keyboard characteristics.
DEVSHARE	Enables the printer and cash adapter to be re-configured dynamically.

The commands SHTERM and SETTERM carry out analogous functions to the primitives ReadInfoWS and WriteInfoWS.

The CLEAR command effects analogous functions to those of the WriteTm primitive, whose control string contains the "clear" command.

5. MOS APPLICATION SERVICES

This chapter describes certain facilities offered by MOS for application purposes for the compiled languages Pascal+, BASIC, and COBOL.

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

- building frames on the screen
- logging, in a system file, a trace of certain execution steps
- producing, if necessary, a dump of the user address space contents
- verifying a signature
- performing MCL activities from Shell or Grandpa.
- suspending the modem connection, between a remote work station and a MOS system.
- intercepting warning messages addressed to the Master Work Station.

The services available to Pascal+ programmers return T-reply errors. This error type is used by the majority of MOS components that offer basic services for the return of errors. This provides a method of classifying errors using a class and a code; the code uniquely identifies the error within the class.

The description of the T-reply type is contained in the "systypes.i" system file.

```
type

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

    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);

    T-dbgErrCode  = (RECORDTOOLONG, INVALIDOPERATION,
                     INVALIDID, INVALIDPARAMETERS,
                     ERRORINTHESTRING, SECURITYVIOLATION,
                     NOTYETIMPLEMENTED, BADSCOPE,
                     BADSTATE, ILLEGINDEX,
                     CHANDESTROY, CHANBUSY,
                     CHILDALIVE, NEOBJECT, NOWRITERS);

    T-pmmWarnCode = (MEMCONFLICT, NEFILE,
                     TYPECONFLICT, FORMATERROR,
                     ILLEGSEGLEN, INVALIDSEGNUM,
                     SOFTINTERRUPT, NOHALT,
                     PROCDESTROY, PROCSUSPEND,
                     BREAKPT, NOTEMPTY);

    T_pmmErrCode  = (MEMOVF, CONCERROR, TOOLONG);

    T_fsWarnCode  = (BYTESTREAMOPERATION, POSITIONALOPERATION,
                     INCONSISTENTDATABASE, NOPRIMARYINDEX,
                     DUPLICATEDKEY, INCOMPLETERECORD,FSRES,
                     RECORDCOMPLETED,FS_ENDOFTAPE);

    T_fsErrCode   = (TIMEOUT, RECORDLOCKED, DEVICENOTREADY,
                     INVALIDPATHNAME, NAMEALREADYEXISTS,
                     NAMENOTFOUND, NOTEMPTYDIRECTORY,
                     ENDOFFILE, OUTOFBOUNDS,
                     RECORDNOTFOUND, RECORDALREADYEXISTS,
                     KEYNOTFOUND, KEYALREADYEXISTS,
                     OPERATORINTERVENTION, ALIASEDNOTFOUND);
                                                          >>
```

```
type

    T_wsWarnCode  = (ENDOFPAGE, DOCUMENTTOOFARIN,
                     OPERATORREQUEST, NOCHARS,
                     BREAKOCCURRED, TERMINALON, PARITYWARNING,
                     WEAKKEYWARNING, WARNINGCHECK);

    T_wsErrCode   = (UNITDOWN, UNITNOTAVAILABLE,
                     RESOURCEBUSY,
                     LINEERROR, BOURRAGE,
                     NODOCUMENT, CARTERISOPEN,
                     LOCAL, VERIFYORREADERROR,
                     VIRGINSTRIPE, PRINTHEADBLOCKED,
                     ENDOFINPUT, BADGEINSERTED,
                     INVALIDINSERTION, ENDOFTAPE,
                     OFFOCCURRED, PRTIMEOUT, FUNCTIONDISABLED,
                     ATTEMPTOVNUMBER, PINNOTMATCHED, WRONGPASSWORD);

    T_rtCode      = (RTTODEFINE);

    T_usrCode     = (USRTODEFINE);

    T_lmWarnCode  = (LMENDOFFILE);

    T_lmErrCode   = (CONNFAILED);

    T_thWarnCode  = (THWTODEFINE);

    T_thErrCode   = (THETODEFINE);

    T_frWarnCode  = (FRTIMEOUT, FRTERMCHAR);

    T_frErrCode   = (FRLINEERROR, SYSTEMFRERROR,
                     PERIPHERALNOTOPENED, PERIPHERALOPENED,
                     PERIPHERALDOWN, INVALIDFRID,
                     FRUNITNOTAVAILABLE,
                     INVALIDFRPARAMETERS);

    T_intrptCode = (INTRPTTODEFINE);
                                                                    >>
```

```
type

    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  );
                         INTPERROR  : (intrptCode  : T_intrptCode );
                   end;
```

HOW TO BUILD FRAMES

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

The Frame Characteristics

The frames which can be built consist of:

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

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

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

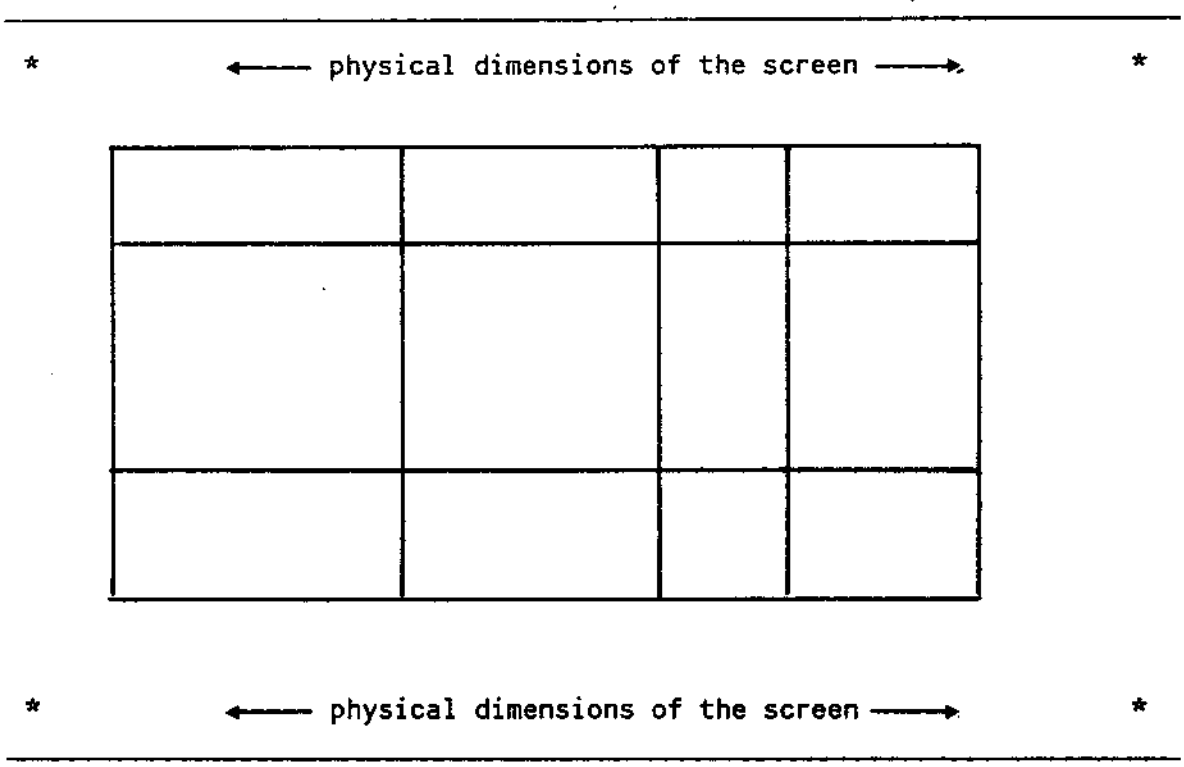


Fig. 5-1 Example of a Frame

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.

How to link the above procedure is described in the Manual COBOL, Program Preparation and Examples.

Parameters

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 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 Note)
OUTPUT		
RET-CODE	01 RET-CODE PIC 9(4) COMP.	Reply code.

Tab. 5-2 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 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 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 )
```

How to link the above procedure is described in the Manual COBOL, Program Preparation and Examples.

Parameters

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

Tab. 5-3 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.

USER ADDRESS SPACE DUMP

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

The part of memory whose contents are to be dumped is the "user address space"; that is, the set of user segments currently in use. (See the Chapter "Segments, Families and Processes" for further details on memory segments.)

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 C.

DUMP DEFINITION

The user address space dump is the process of saving the image of all the user active segments in 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, otherwise its contents are replaced.

Further details on the dump file and its contents are 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 it from the module in which it is 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.

How to link the above procedure is described in the Manual PASCAL+, Program Preparation.

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+
- any special characteristics.

DISABLEDUMP

This procedure disables the dump activity.

Pascal+ Call

```
PM_Dump.DisableDump;
```

Characteristics

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;
```

Characteristics

This procedure substitutes a new exception handler for 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 in a "dump file".

Pascal+ Call

```
PM_Dump.MemoryDump;
```

Characteristics

This "dump file" can be displayed and printed by using the DMPRINT utility described in Appendix C.
The calling program is not aborted.

SIGNATURE VERIFICATION

MOS provides two Pascal+ procedures for verifying signatures. These procedures respectively allow:

- acquisition of the signature, on paper, to be verified by an optical reader in MHC (Modified Huffman Code) (SCANNER)

- its decompression into a bitmap in normal or zoomed format (BITMAP).

The signature can be displayed by means of the PGU (Graphics Management Package) "PUT" procedure.

The Signature Verification procedure is illustrated below.

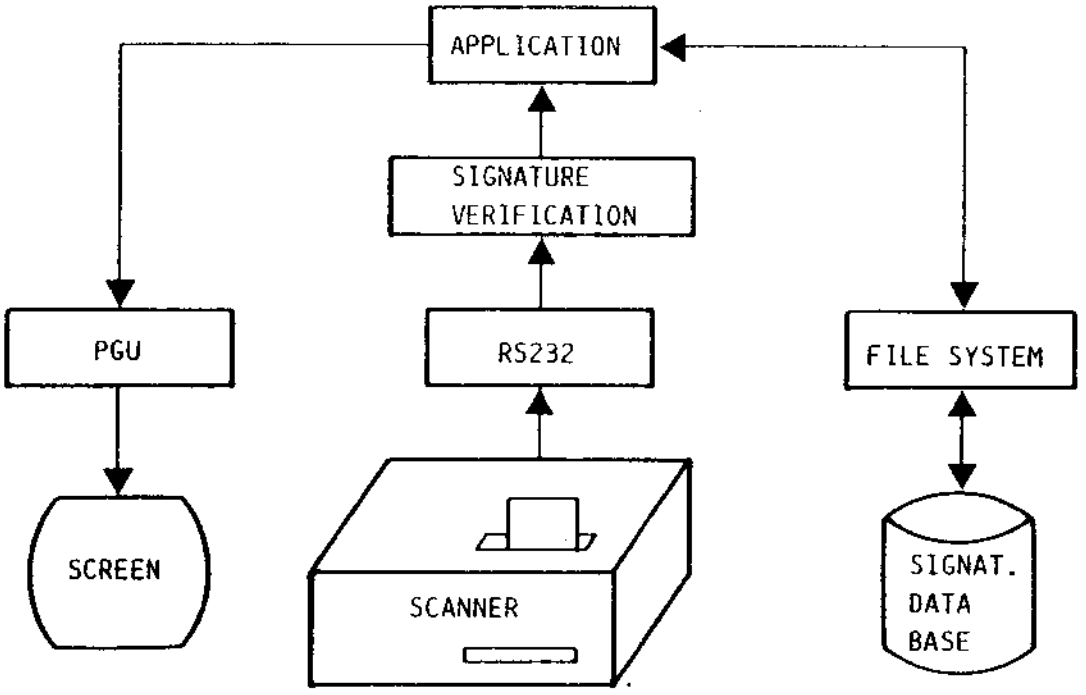


Fig. 5-4 Functional Diagram of the Signature Verification Feature

The application program:

- executes the SCANNER procedure to acquire the signature in MHC code and recieves 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    (size of the driver reception buffer)
INITTXBUF    = 512    (size of the driver transmission buffer)
PGMRXTXBL    = %0B    (transmission speed: 4800 bauds)
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 MOS, System Software Generation and Installation, Manual.

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 a bitmap.

PGU Procedures to be Called

The following PGU procedures must be called by 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 PGU, Graphics Management Package, Programmer Guide.

COBOL INTERFACE FOR PGU CALL

How to link the above procedures is described in the Manual COBOL, Program Preparation and Examples.

Parameter 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. 5-5 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

How to link the above procedures is described in the Manual PASCAL+, Program Preparation.

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.

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. 5-6 Pascal4 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 PGU, Graphics Management Package, Programmer Guide.

BRESET

This procedure disables buffer use, after which 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 execution of the OPENPGU procedure
- unloads the PGU.

COBOL Call

```
CALL "CLOSEPGU".
```

Pascal+ Call

```
ClosePGU;
```

This procedure initializes the PGU variables and the bitmap. The existence of a font file 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 cannot be loaded.
5	PGU cannot 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, 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, 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 is already in this state the procedure call has no effect.

COBOL Call

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

Pascal+ Call

```
SetAlpha;
SetGraph;
```

Characteristics

1. After execution of a graphics procedure the PGU is left in the graphics state. In this 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 this 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, it is unnecessary to use the SETALPHA and SETGRAPH procedures.

COBOL INTERFACE FOR SIGNATURE VERIFICATION PROCEDURES CALL

How to link the signature verification procedures is described in the Manual, COBOL, Program Preparation and Examples.

Parameter Descriptions

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 on normal 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 normal bitmap.
x_pixel_Zoom	01 x_pixel_Zoom PIC S9(4) COMP.	Pixel number per row of the zoomed bitmap.

>>

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 normal 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 on zoomed bitmap.

Tab. 5-7 COBOL Parameters Description

PASCAL+ INTERFACE FOR SIGNATURE VERIFICATION PROCEDURES CALL

How to link the above procedure is described in the Manual PASCAL+, Program Preparation.

Parameter Descriptions

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 on normal bitmap. Passed by VAR.

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

>>

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 normal 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 normal 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 on zoomed bitmap. Passed by VAR.

Tab. 5-8 Pascal+ Parameters Description

SIGNATURE VERIFICATION PROCEDURES

This section describes the two procedures used for signature verification:

- BITMAP
- SCANNER

BITMAP

This procedure decompresses 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 on non-zoomed bitmap. See Characteristic 1.
Zoom_Bitm	Area containing the signature on 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, PC used as work stations and alphanumeric integrated work stations with an MEG 3354 board.

This procedure enables acquisition of 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 "Characteristics".
y_mm	Height in millimeters of the window to be read. The value must be the range from 10 to 100 millimeters. See "Characteristics".
line_id	Identifier of the channel to which the optical reader is connected.

>>

PARAMETER	MEANING
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	Operation not allowed.
16	Line error (linedown, overrun, etc.)
32	Reading or writing break.
64	Time out.
100	Optical reader is off.
101	Sheet not ready.
102	Optical reader off-line
103	Transmission error.
104	Reading window too large
105	Reading window too high.
106	Reading window too big.
255	MHC code unrecognized.
1000+n	Line closing error of code "n". The "n" value is in the range 0 to 64.

Characteristics

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

EXECUTION OF MCL ACTIVITIES FROM SHELL OR GRANDPA

MOS provides the EXEC function so that an MCL activity (executable programs, MCL procedures,..) is called and executed by a program activated via Shell or Grandpa, and control is returned 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
```

How to link the above procedure is described in the Manual PASCAL+, Program Preparation.

EXEC

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 under Characteristics.
result	This array specifies the character-string contained in the Shell variable %STATUS.
start	Ordinal number of the character at which the the calculation of 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 if the MCL activity can be executed, or a different value of T_reply type if errors have occurred in the request for execution of the activity (e.g. if the name of an executable program does not exist). For a description of the T_reply type see the introductory section of this Chapter.

Characteristics

The reply code returned by the MCL activity by means of the "usrtype" parameter 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

In order to save telephone expenses for the use of switched lines, MOS provides a function which suspends the connection between a remote work station and the MOS system to which it is connected.

COBOL INTERFACE

The variable "WAITFORON" must be defined in the application program as follows:

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

WAITFORON specifies the time for which the connection is to be suspended, expressed as units of 100 milliseconds.

PASCAL+ INTERFACE

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

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

How to link the above procedure is described in the Manual PASCAL+, Program Preparation.

LOGOFF

This function disconnects the physical line between the work station and the host computer.

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 Characteristic 2.
waitForOn	Time for which the connection is to be suspended, expressed as units of 100 milliseconds.

Reply Codes

The following Table gives a list of the reply codes returned by the function. These reply codes are of the T_reply type; this type is described in the introductory section to this Chapter.

REPLY CLASS	REPLY	MEANING
DEBUG	INVALIDID	Invalid work station identifier.
	INVALIDOPERATION	This operation is required for a Work Station not connected to a system via Twin or Mux. See Characteristic 3.
WSERROR	UNITNOTAVAILABLE	Work station not configured.
CORRECT	OKAY	Operation correctly executed

Characteristics

1. See the PMM and Driver Primitives, Reference Manual in the Section "Work station Identification" in Chapter 3 for the work station identifier.

2. This function sets the DTR (Data Terminal Ready) and the RTS (Request To Send) control wires to "off" for the work station identified by "wsId"; it then suspends the caller for "waitForOn" time, and sets the DTR and RTS wires "on" again.

3. This function is operative only for work stations connected via Twin and Mux in RS232 mode (both transparent and WS mode). The function is intended to be used in modem-included connections. The Mux/ws11 connection case needs a 1.2.0. (Rev. 2), or later, Mux firmware release.

4. The user should take particular care when using this function, as its behaviour depends on the PHYSICAL CHARACTERISTICS of the connection, and the system cannot know and check all of them.

5. To guarantee the complete restore of work-station state before LOGOFF operation, a physical work-station OFF/ON transition is necessary.

INTERCEPTION OF WARNING MESSAGES ADDRESSED TO MWS

This section describes the message interception function available for the Pascal+ language.

The Master Work Station (MWS) may receive the following message types:

- Warning Messages
- Messages containing miscellaneous information
- Messages requesting operator intervention (requiring a reply)

Warning messages are be associated to messages of the second type.

If no use is made of the interception services described in this section, the warning messages are immediately highlighted in the lower window of the MWS (also known as the warning terminal).

The other messages are queued. The Shell MTA command makes it possible to de-queue them and display them in the upper window (also known as the Master Terminal) of the MWS. After having taken note of the contents of the messages, the operator may send replies where required.

The interception services make it possible to modify the above mechanism; the warning messages are intercepted by the application, which may then handle them appropriately before sending them to the MWS.

The procedures available for the interception of warning messages are as follows:

OPINTERCEPT	which opens a message interception session
TAKEWARN	which de-queues the intercepted messages
SAWARN	which sends the intercepted messages to the MWS
CLINTERCEPT	which closes an interception session.

The above can be used only by those applications owned by the super-user.

PASCAL+ INTERFACE

To use these services the following modules must be included in the program concerned:

```
systypes.i
cem.d
iwarn.d
```

How to link the above procedures is described in the Manual PASCAL+, Program Preparation.

Definition of Types

As well as the T_reply type, described in the section "System Types", these primitives make use of the following types and constants:

```
type
        T_MParNum     = 1 .. MMAXPAR;

        T_MTypelem    = (mstr_, mlong_);

        T_MElem       = record
                        case typelem : T_MTypelem of
                          mstr_      : (len  : byte;
                                        pstr : ^char);
                        mlong_       : (val  : longinteger);
                        end;

        T_MParam      = record
                          parnum : T_MParnum;
                          per    : array [T_MParnum] of T_MElem;
                        end;


        T_Mvarpar     = ^T_MParam;

        T_intBuf      = packed array [1 .. 80] of dataitem;

        T_intName     = packed array [1 .. 80] of dataitem;

        T_TypeWarn    = (ClearMsg,UsrCodeMsg,SysCodeMsg);

        T_Warn        = record
                            case typewarn : T_TypeWarn of

                             clearMsg     : (sender   : T_intName;
                                             warnText : T_intBuf);

                             UsrCodeMsg   : (uClass : integer;
                                             uCode  : integer;
                                             vpar   : T_Mvarpar);

                             SysCodeMsg   : (sClass : integer;
                                             sCode  : T_reply);

                          end;


const
        NMMAXPAR = 4;
```

CLINTERCEPT

This function closes the interception session.

Definition of the Function

```
function ClIntercept    : T_reply;
```

Reply Codes

The function returns the following diagnostics:

REPLY CLASS	REPLY	MEANING
CORRECT	OKAY	Operation correctly executed.
DEBUG	INVALIDOPERATION	No open interception session exists.
SYSTEM	SYSTEMERROR	System error caused by the communications sub-system.

Characteristics

On completion of CLINTERCEPT the warning messages are sent again to the warning terminal.

If the interception session is not closed using this primitive before termination of the program, the warning messages will continue to be queued in the interception queue, but they will not be processed.

This function opens an interception session. After this function has been called, the warning messages will cease to be displayed on the warning terminal and will be placed in a special queue (called the interception queue) to await de-queuing by the application, using the TAKEWARN primitive.

Definition of the Function

```
function OpIntercept (timeout    : longinteger;
                      threshold  : integer)  :  T_reply;
```

PARAMETER	DESCRIPTION
INPUT	
timeout	This parameter specifies the maximum length of time (in milliseconds) for which a process invoking this primitive will be suspended if no message(s) arrive in the interception queue during that time. See Characteristic 1.
threshold	This parameter specifies the maximum size in bytes of the warning message queue at any time. See Characteristic 2.

Reply Codes

The function returns the following diagnostics:

REPLY CLASS	REPLY	MEANING
CORRECT	OKAY	Operation executed correctly.
DEBUG	INVALIDPARAMETERS	One or more parameters incorrect.
	INVALIDOPERATION	The request cannot be accepted as an interception session has already been opened.
SYSTEM	SYSTEMERROR	System error caused by the communications subsystem.

Characteristics

1. Zero is a valid value for the timeout parameter and implies no suspension, i.e. either a message is returned or an immediate reply code (TIMEOUT) is returned.

   An infinite timeout (i.e. no timeout period but the function waits until a message is available) can also be provided by setting the timeout value to -1.

2. The threshold parameter is an important tuning parameter, and its value should be calculated with care. It indicates to the system the maximum size (in bytes) of the warning message queue. The system accepts messages for this queue until the threshold value has been exceeded; subsequent messages are then rejected.

This function makes it possible to de-queue the warning messages sent to the lower window of the MWS. The messages are de-queued one at a time.

Definition of the Function

```
function TakeWarn (var warn : T_warn) : T_reply;
```

PARAMETER	DESCRIPTION
OUTPUT	
warn	Contains the "captured" information. They differ, depending on whether the de-queued messages are of type SysCodeMsg, UsrCodeMsg or ClearMsg (see Characteristic 3). Messages of the first type originate from system components; the second and third types originate from other components (e.g. the Shell command CALLMWS). For SysCodeMsg the parameter includes: - the class of the component that sent the message (from 1 to 8 chars). - the identifying code (of T_reply type) of the message by the component. For UsrCodeMsg the parameter includes: - the class of the component that sent the message (from 1 to 8 chars). - the identifying code of the message (from 1 to 5 chars). - any parameters relative to the message (see Characteristic 2). For ClearMsg the parameter includes: - the name of the component that sent the message. - the text of the message.

Reply Codes

This function returns the following diagnostics of T_reply type (described in the introductory section of this Chapter):

REPLY CLASS	REPLY	MEANING
CORRECT	OKAY	Operation executed correctly.
SYSTEM	RECEIVERROR	A receive error (parity, over-run, ...) has been detected on the line connecting the MWS.
	TIMEOUT	The time established by OPINTERCEPT for the interception of messages has elapsed. See also Characteristic 1.
	SYSTEMERROR	System error caused by the communications sub-systems.

Characteristics

1. When no message is queued and a zero time-out has been specified in the OPINTERCEPT function, the calling program is suspended until a new message arrives. When no message is queued and a non-zero timeout has been specified, the calling program is suspended for the time specified by the time-out.

2. Within the messages associated to Warnings (for example messages requiring operator intervention) it is possible to set parameter fields whose effective values are subsequently passed as parameters to the primitive. In this way messages of the type

   ---

   ERROR IN LINE %1: FILE %2 DOES NOT EXIST

   ---

   can be defined, where %1 and %2 are set with the values passed as first and second parameter of the primitive.

These values may be strings or long integers, in which case they are automatically converted into strings.

A maximum of four parameters is allowed.

3. The above-mentioned types of messages (SysCodeMsg, ...) are those provided in the centralized database for system messages. For construction, updating and nationalization of this database, the Shell MKCEM command should be used; for further information, see the MKCEM command.

SAWARN

This function makes it possible to enqueue to the Master Work Station the warning messages intercepted by the TAKEWARN function.

Definition of the Function

```
function SAwarn (warn : T_warn) : T_reply
```

PARAMETER	DESCRIPTION
INPUT	
warn	Contains the warning information to be sent to the MWS. It differs depending on the message type: SysCodeMsg, UsrCodeMsg, and ClearMsg (see Characteristic 3). Messages of the first type originate from system components; the second and third types originate from other components (e.g. the Shell command CALLMWS). For SysCodeMsg the parameter includes: - the class of the component that sent the message (from 1 to 8 chars). - the identifying code (of T_reply type) assigned to the message by the component. >>

PARAMETER	DESCRIPTION
	For UsrCodeMsg the parameter includes: - the class of the component that sent the message (from 1 to 8 chars). - the identifying code of the message (from 1 to 5 chars). - any parameters relative to the message (see Characteristic 2). For ClearMsg the parameter includes: - the name of the component that sent the message. - the text of the message.

Reply Codes

The function returns the following T_reply type diagnostics (described in the introductory section of this Chapter).

REPLY CLASS	REPLY	MEANING
CORRECT	OKAY	Operation correctly executed.
DEBUG	INVALIDOPERATION	No open interception session exists.
SYSTEM	SYSTEMERROR	System error caused by the communications sub-system.

PART 2 - ENVIRONMENT INTEGRATION

INTRODUCTION TO PART 2

This part is a guide to integrating environments, as it:

- specifies the memory segments available to the applications according to the services used by the applications concerned (Chapter 6)
- specifies the possibilities, existing in multifunctional systems, relative to data exchange between different environments (Chapter 7)
- guides the user in the implementation of shared user services, besides those services supplied by Olivetti (Chapter 8).

Part 2 is of particular interest, therefore, to systems administrators.

6. ASSIGNING USER SEGMENTS

This chapter provides the information that will enable the user to correctly allocate the segments making up an application in the address space of a family.

STRUCTURE OF THE FAMILY ADDRESS SPACE

As already described in the Chapter "Process and Memory Management Interfaces", in the Section "Family Address Space Structure", the operating system occupies the segments between 0 and MINSEG inclusive and the services that it offers are available to all users.

The user packages are allocated by Grandpa immediately after the operating system (Fig. 6-1) in one-MMU systems; in two-MMU systems they are allocated in the second MMU. They are allocated in the Grandpa address space.

After all user packages have been loaded, Grandpa evaluates the largest segment (GpaSeg) currently used in its address space. Private segments of all spawned families must be larger than the GpaSeg. The user must allocate its segments between the value of GpaSeg+1 and the highest allowed value (these limits are inclusive). The highest user-available segment is "61", as "62" is used by the system and "63" is the stack segment.

The user segments that implement the specific functions of an application and any (non-shared) services used by the application itself reside in the family private space.

As regards allocation of the non-shared services used by the application, note that:

- some (e.g, BASIC run time services) must be allocated in prefixed positions of the address space range available to the user (Gpaseg+1, 61)

- others can be located in any segment of this range.

Note also that the former, although linked to an l-module prepared by the user, are loaded into memory transparently to the l-module that uses them. This occurs, for example, in the case of VISA services.

The figure below shows the positioning in the address space of segments belonging to the standard services in the case of systems fitted with both 1 and 2 MMUs.

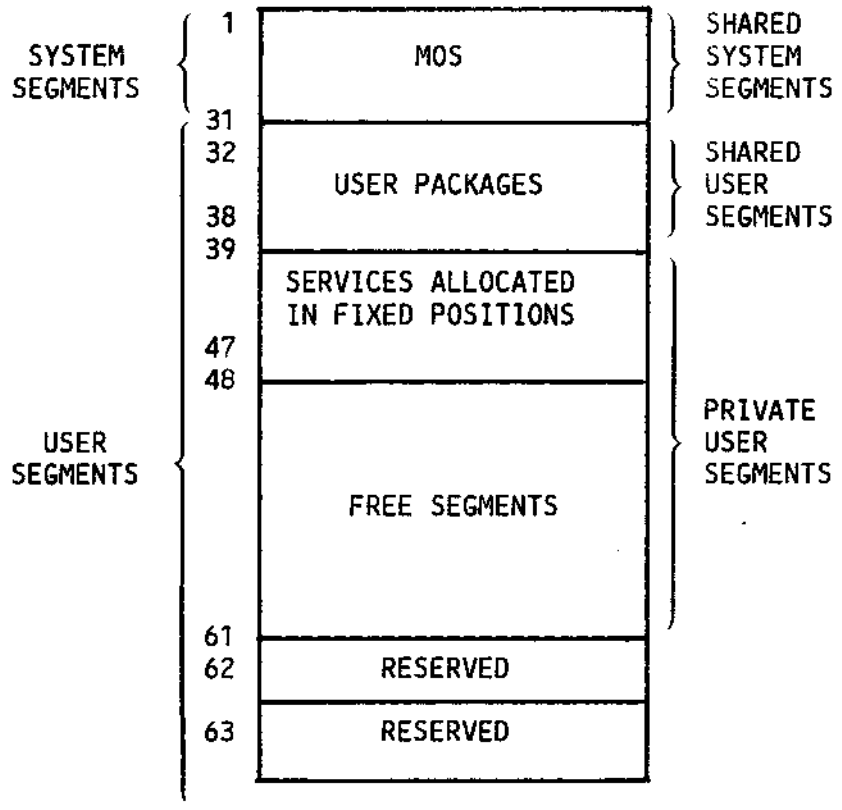


Fig. 6-1 Structuring of Family Address Space (1-MMV Systems)

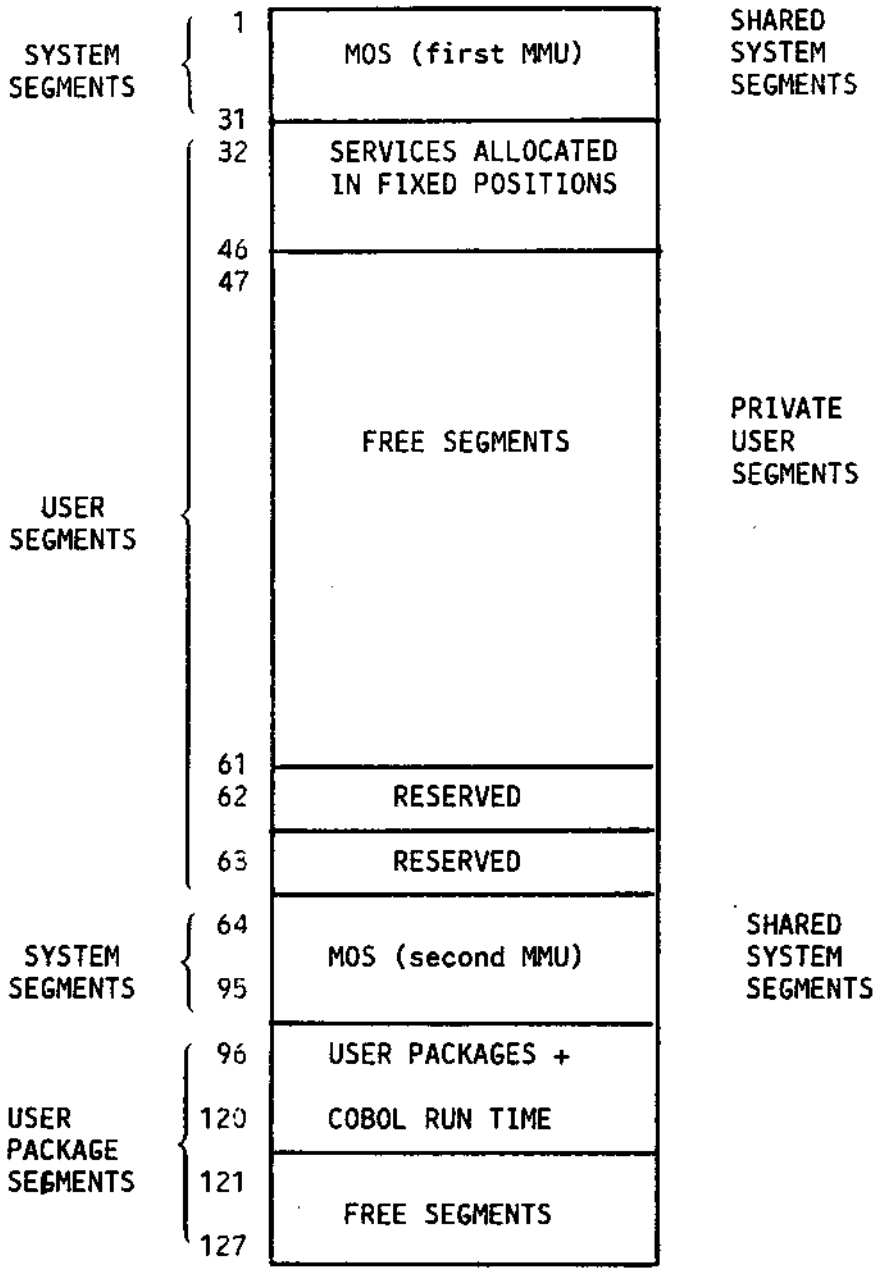


Fig. 6-2 Structuring of Family Address Space (2-MMU Systems)

In order to avoid the risk 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).

1-MMU SYSTEMS: DESCRIPTION OF USER SEGMENTS

The following Table shows, for systems fitted with only one MMU, which segments are occupied by each user package, and by the standard non-shared services requiring allocation in fixed positions of the address space.

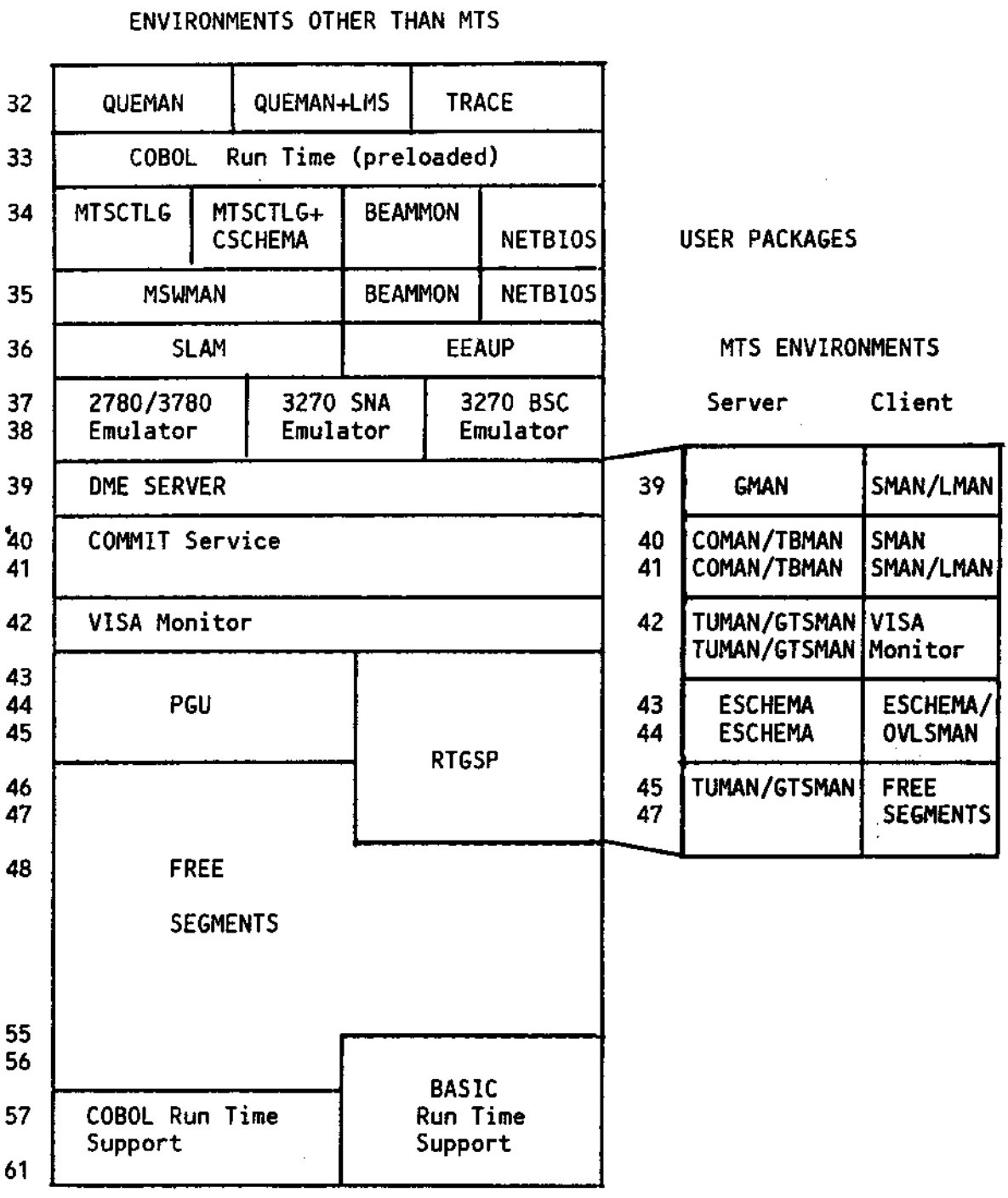


Fig. 6-3 Allocation of User Packages and Non-Shared Services Occupying Fixed Positions

To fully understand the contents of Tab. 6-2 the reader should note that:

QUEMAN is the queue manager (used by the batch environment monitor and the spooling services).

TRACE executes the trace of the calls that the application effects to the system.

BEAMMON is the BEAM environment monitor.

MSWMAN is the manager of the Message Switching function.

NETBIOS makes it possible to insert a MOS system in an OLINET network.

SLAM belongs to the ONE environment.

EEAUP is the SNA network monitor.

MTSCTLG and MTSCTLG+CSCHEMA belong to the MTS environment respectively without and with the chained database.

GMAN, SMAN, ... belong to the MTS environment server or client, as specified in Tab. 6-2.

DME server makes it possible to access, from PB applications, data resident on MOS systems.

PGU is the Graphics Package, Unified

RTGSP is the run-time of the Graphics Subroutine Package.

The table highlights clearly the impossibility of simultaneously using certain components (for example, MTS and BEAM). A complete list of incompatible components is given in the Section "Limitations on Multifunctionality in 1-MMU Systems".

2-MMU SYSTEMS: DESCRIPTION OF USER SEGMENTS

The following tables show, for systems having 2 MMUs, which user segments are occupied by user packages and the standard non-shared services that have to be allocated in fixed positions of the address space.

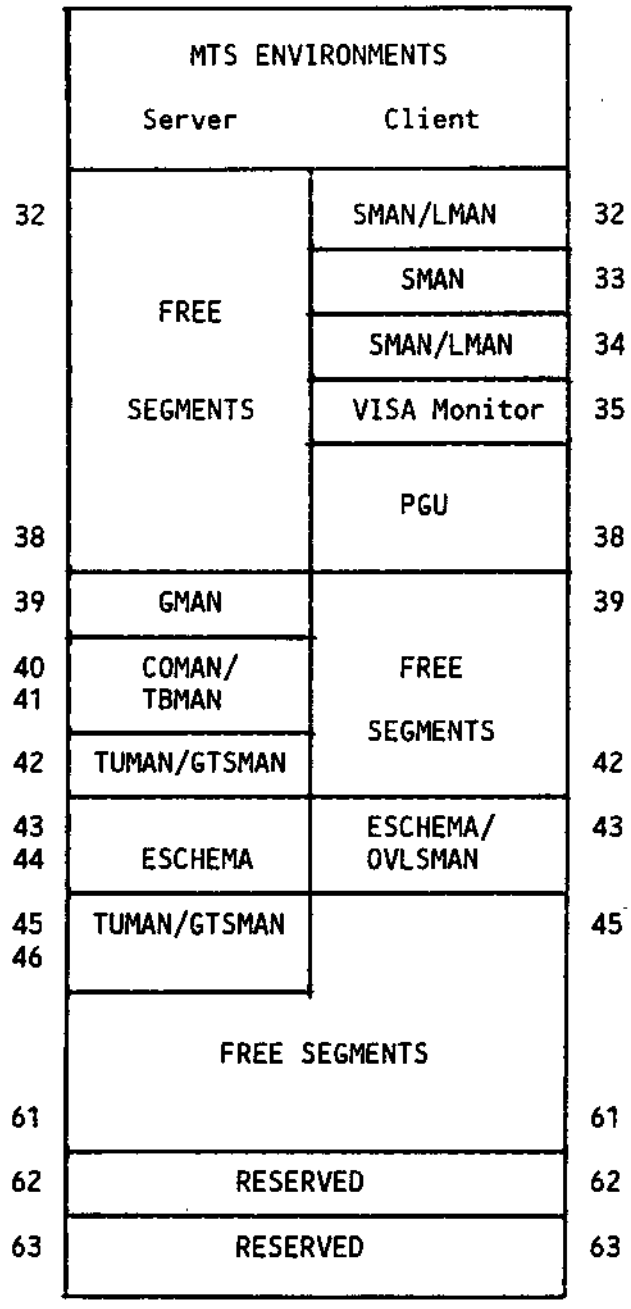


Fig. 6-4 Description of User Segments on the First MMU

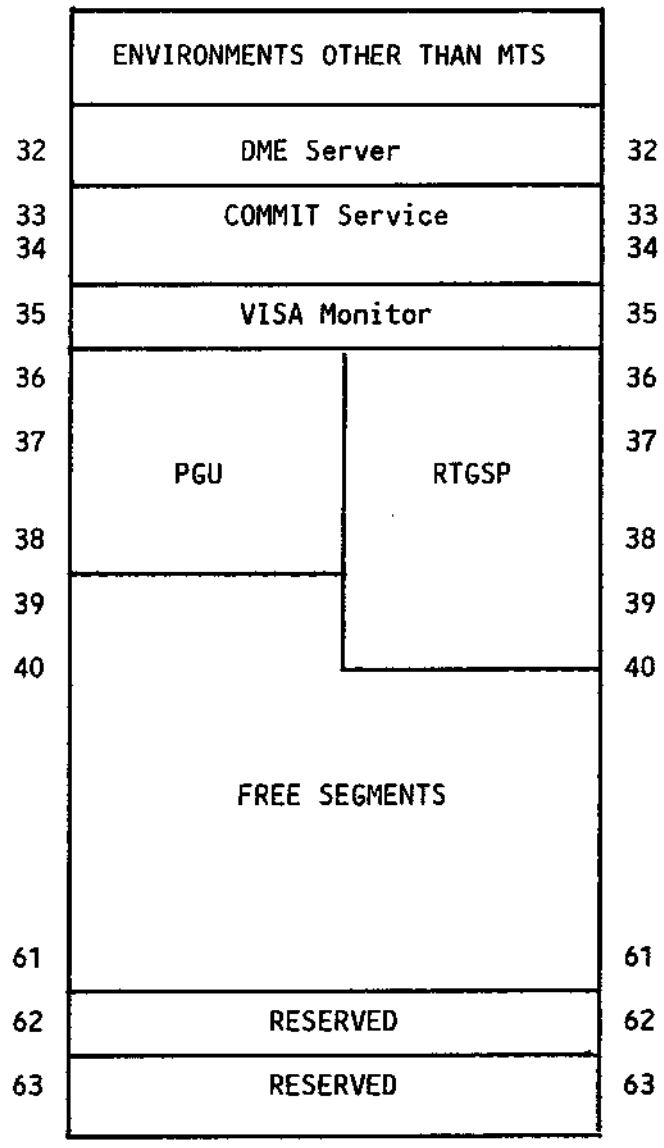


Fig. 6-5 Description of User Segments on the First MMU

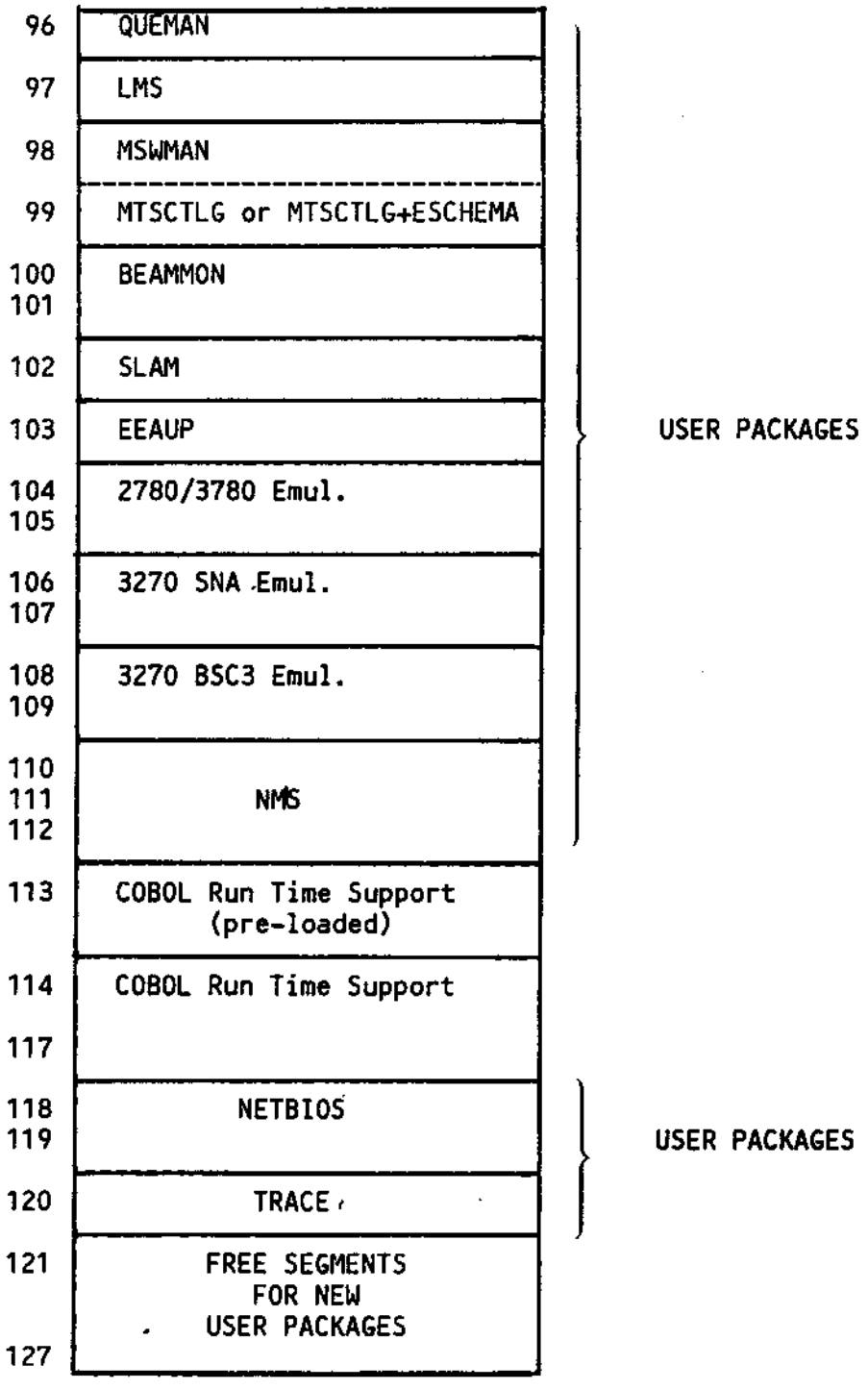


Fig. 6-6 Description of User Segments on the Second MMU

Note

1. By comparing Tables 6-4 and 6-5 with Table 6-3, it will be evident that, in the case of systems having 2 MMUs, the standard non-shared services take the place of the User Packages which, together with the COBOL run-time support, are located in the second MMU.

2. As the above tables show, the following do not belong to the application family that requests them:

   - The services provided by the interpreters (for example VISA, BEAM, SHELL, etc.)

   - The debugging services

   These services belong to another family that communicates, if necessary, with the requesting one by means of a monitor allocated in a space shared between the two families. For example, the BEAM interpreter communicates with an application using BEAM primitives by means of BEAMMON.

3. The system Network Control Centre (NCC) of the Network Monitoring Service (NMS) must be equipped with two MMUs as NMS is not available on one-MMU systems.

CALCULATING FREE USER SEGMENTS

By using Tables 6-3, 6-4 and 6-5 it is possible to obtain the segments in the address space of a MOS system, equipped with 1 or 2 MMUs which are "free"; they can thus be used for allocating the user-written segments of an application.

The following segments are thus made available to an application:

- The segments marked "free user segments" in the preceding Tables.

- The segments of Olivetti user packages (on 1-MMU systems) which have not been configured in the system concerned (unless the corresponding segments are used for the allocation of user packages written by the user).

- The segments of non-shared services not used by the application concerned.

The segments of any Olivetti user packages not configured are the ideal segments for loading any user packages written by the user.

However, it should be noted that in both 1-MMU and 2-MMU systems the "user" segments are in the range 32 - 61.

The segments not occupied by user packages and by the standard services allocated in fixed positions are - ideally - available also for application programs; the user is recommended, however, to consult the relevant manuals for the standard services to ensure that all the functions of these services are guaranteed if the program occupies these segments; e.g. the Commit Service requires the program to be allocated in segment numbers 42 upwards.

An example is given below for the calculation of free segments for a COBOL application, activated by Shell, that uses the VISA and COMMIT services, for a system fitted with 1 MMU and Gpaseg = 38.

Shell creates a family for the application whose private segments available are between 39 and 61 inclusive.

The following segments must be excluded from this range:

- Segments between 57 and 61 (inclusive), as these are used by the COBOL run-time support.

- Segment 42, used by the VISA monitor.

- Segments 40 and 41 used by COMMIT.

The following segments are available for user programs: segments from 43 to 56 (inclusive).

It is also possible to use the segments in the range 32 - 38, and 39, if they are available (as explained above).

USER VISIBILITY OF MEMORY OCCUPATION

The user may obtain the value for the memory occupation of a program by means of the CSIZE command, which can be activated in the Shell environment. This supplies information regarding:

- memory occupation of one or more l-modules specified by the user calling the command

- the segments allocated to these l-modules.

The memory occupation value given by this command refers to that of the program specified, and not to the disk space occupied by the file containing the program.

The memory occupation of a program may be obtained without it being loaded into memory; in other words the CSIZE command may be executed for a program that does not currently reside in memory, but on disk.

7. MOS MULTI-FUNCTIONALITY

As already explained, MOS's multi-functionality means that all the MOS software environments and programming languages can be used to the best advantage in a user-written application. It is thus possible to:

- simultaneously execute applications programs written in different languages, with data exchange between them (if required) using shared files accessed concurrently

- have different software environments active at the same time in which the above applications programs run

- operate from one work station in different environments, exchanging data by means of shared files, accessed non-concurrently.

It therefore follows that use of the MOS multifunctionality is conditioned by the type of data exchanged between programs written in different languages. This is because:

- each language uses organizations for storing the data (files) which are not always used by the other languages

- not all languages have control mechanisms (locks) on concurrent access to shared data.

A further restriction to multifunctionality is due to the technique followed by MOS for the allocation of software environments in memory. This only occurs, however, in systems having only one MMU.

PROBLEMS DUE TO LANGUAGES

In order to clarify the restrictions in multifunctionality due to languages it is necessary to classify the possible situations (levels) of multifunctionality, according to the type of data exchange involved (files shared concurrently or non-concurrently ...).

Note that the file organization and file locking mechanisms allowed by the various languages must also be borne in mind, as well as the relationships existing between the data types of the various languages.

DATA ORGANIZATIONS ALLOWED BY LANGUAGES

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

The table also highlights any possibility of locking in the various languages.

It should be remembered that access to the File System is achieved by Pascal+ routine calls, so that all information relating to files accessible to 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.
ICE COBOL	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)
C	yes	no	-	-	no	-	-	-	-	-	4096 bytes	20
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. 7-1 Accepted File Organizations

MULTI-FUNCTIONALITY LEVELS AND DATA EXCHANGE

From the point of view of the data exchange, four multi-functional 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 data between the application environments.

Level 2 - The function of Level 1, but the programs exchange data using files accessed non-concurrently.

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

Level 4 - Possibility of linking objects obtained by sources written in different languages.

The restrictions of these four multifunction 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. on those environments which can exchange data and the types of file by means of which this operation is performed (with separate access), can be obtained directly from the preceding Table.

For example, the languages

COBOL and Compiled BASIC

can exchange data 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 simply 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 data and the files by means of which this operation is performed (with concurrent access), can be obtained directly from the table by comparing the organization of the files relating to the two languages, and 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 data concurrently by means of the following files:

Relative COBOL - Relative Compiled BASIC
Indexed COBOL - Keyed Compiled BASIC

Level 4

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

Compiled BASIC - Pascal+	when a Compiled BASIC program is used for recalling a Pascal+ procedure. See the manual Compiled BASIC, Program Preparation and Examples.
COBOL - Pascal+	when a COBOL program is used for recalling a Pascal+ procedure or when a Pascal+ program is used for recalling a COBOL procedure (the MAIN must be written in COBOL.) See the Manual COBOL, Program Preparation and Examples.
C Language - Pascal+	when a C program is used for recalling a Pascal+ procedure or when a Pascal+ program is used for recalling a C procedure (the MAIN must be written in Pascal+). See the Manual C, Program Preparation and Execution.

NOTES ON DATA EXCHANGE

For data exchange between the various environments, via shared files, it is necessary to bear in mind the relations that the types of data from one environment have towards the other.

Pascal+ Types ⟶ Other Component Types

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

Fig. 7-2 Data Types that can be Exchanged between Pascal+ and other Components

1. Data can also be exchanged between a Positional with Deletion Pascal+ file, and a Random Interperted BASIC file, remembering that:

   - a Positional No Deletion file must be created by the appropriate MCL utility or by a Pascal+ program

   - an Interpreted BASIC program also accesses deleted or non-existent records, replying not with an error indication, but with data equal to binary 0.

2. Data can also be exchanged between a positional No Deletion Pascal+ file and a Sequential DMS file, remembering that:

   - the file must be created by the appropriate MCL utility or by a Pascal+ program

   - the DMS cannot carry out deletion operations.

3. Data can also be exchanged between a Positional No Deletion Pascal+ file and a Sequential SORT file, with the latter mapped on the Positional No Deletion file, remembering that the Sequential SORT is only an input file.

4. A C string is terminated by a byte with the value zero; in the other languages, a string does not have a standard terminator.

ICE COBOL Types ⟶ Other Component Types

ICE COBOL	PASCAL+	COBOL	Comp. BASIC	Interp. BASIC (1)	FORTRAN (2)	DMS (3)	SORT (4)	C
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings		Alphanum. strings		Alphanum. strings (5)
COMP	integer (2 bytes)	COMP	integer	integer		COMP		int or short
COMP-3	-	COMP-3	-	-		COMP-3		-

Fig. 7-3 Data Types that can be Exchanged between ICE COBOL and Other Components

1. Data can also be exchanged between a Relative ICE COBOL file and a Random Interpreted BASIC file, remembering that:

   - a Positional No Deletion file must be created by the appropriate MCL utility or by a Pascal+ program

   - an Interpreted BASIC program also accesses the deleted or non-existent records, replying not with an error indication but with data equal to binary 0.

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

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

   - a Positional No Deletion file must be created by the appropriate MCL utility or by a Pascal+ program

   - the DMS cannot carry out deletion operations.

4. All types of ICE COBOL data are compatible with SORT. In addition, data can be exchanged between Sequential ICE 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.

5. A C string is terminated by a byte with the value zero; in the other languages, a string does not have a standard terminator.

COBOL	PASCAL+	ICE COBOL	Comp. BASIC	Interpr. BASIC (1)	FORTRAN	DMS (2)	SORT (3)	C
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings			Alphanum. strings (5)
COMP	integer (2 bytes)	COMP	integer	integer	INTEGER*2			int or short
COMP-1	-	-	Single Precision	Single Precision	REAL*4			float (4)
COMP-3	-	COMP-3	-	-	-			
COMP-2	real (8 bytes)	-	Double Precision	Double Precision	REAL*8			double
COMP-4	longint	-	-	-	INTEGER*4			long

Fig. 7-4 Data Types that can be Exchanged between COBOL and other Components

1. Data can also be exchanged between the Relative COBOL file and a Random Interpreted BASIC file, remembering that:

   - a Positional No Deletion file must be created by the appropriate MCL utility or by a COBOL program

   - an Interpreted BASIC program also accesses the deleted or non-existent records, replying not with an error indication but with 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 files, remembering that:

   - a Positional No Deletion file must be created by the appropriate MCL utility or by a COBOL program

   - the DMS cannot carry out deletion 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.

4. When it is passed as a parameter, a float in C is always converted into a double. Thus, a C program cannot pass a float-type parameter to a function in another language; and a program in another language cannot pass a float-type parameter to a C function.

5. A C string is terminated by a byte with the value zero; in the other languages, a string does not have a standard terminator.

Interpreted BASIC Types ⟶ Other Component Types

Interp. BASIC	PASCAL+ (1)	ICE COBOL (1)	COBOL (1)	Comp. BASIC (1)	FORTRAN	DMS (2)	SORT (3)	C
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	yes	Alphanum. strings (5)
integer	integer	COMP	COMP	integer	INTEGER*2	COMP	yes	int or short
Single Precision	-	-	COMP-1	Single Precision	REAL*4	COMP-1	yes	float (4)
Double Precision	real	-	COMP-2	Double Precision	REAL*8	COMP-2	yes	double

Fig. 7-5 Data Types that can be Exchanged between Interpreted BASIC and other Components

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

   - a Positional No Deletion file must be created by the appropriate MCL utility or by a program in the relevant language

   - an Interpreted BASIC program also accesses the deleted or non-existent records, replying not with an error indication but with 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 deletion operations.

   - if the file is created by DMS, the Interpreted BASIC program also accesses deleted or non-existent records, replying not with an error message 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.

4. When it is passed as a parameter, a float in C is always converted into a double. Thus, a C program cannot pass a float-type parameter to a function in another language; and a program in another language cannot pass a float-type parameter to a C function.

5. A C string is terminated by a byte with the value zero; in the other languages, a string does not have a standard terminator.

Compiled BASIC Types ⟶ Other Component Types

Comp. BASIC	PASCAL+	ICE COBOL	COBOL	Interp. BASIC (1)	FORTRAN	DMS (2)	SORT (3)	C
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	yes	Alphanum. strings (5)
integer	integer	COMP	COMP	integer	INTEGER*2	COMP	yes	int or short
Single Precision	-	-	COMP-1	Single Precision	REAL*4	COMP-1	yes	float (4)
Double Precision	real	-	COMP-2	Double Precision	REAL*8	COMP-2	yes	double

Fig. 7-6 Data Types that can be Exchanged between Compiled BASIC and other Components

1. Data can also be exchanged between Relative Compiled BASIC and Random Interpreted BASIC files. It should be remembered that:

   - a Positional No Deletion file must be created by the appropriate MCL utility or by a Compiled BASIC program

   - an Interpreted BASIC program also accesses the deleted or non-existent records, replying not with an error indication but with data equal to binary 0.

2. Data can also be exchanged between Sequential Compiled BASIC and Sequential DMS Files, remembering that:

   - a Positional No Deletion file must be created by the appropriate MCL utility or by a Compiled BASIC program

   - the DMS cannot carry out deletion 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.

4. When it is passed as a parameter, a float in C is always converted into a double. Thus, a C program cannot pass a float-type parameter to a function in another language; and a program in another language cannot pass a float-type parameter to a C function.

5. A C string is terminated by a byte with the value zero; in the other languages, a string does not have a standard terminator.

FORTRAN Types ⟶ Other Component Types

FORTRAN	PASCAL+	ICE COBOL (1)	COBOL	Comp. BASIC	Interp. BASIC	DMS	SORT	C
Alphanum. strings	Alphanum. strings		Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	yes	Alphanum. string (3)
INTEGER*2	integer		COMP	integer	integer	COMP	yes	int or short
REAL*4	-		COMP-1	Single Precision	Single Precision	COMP-1	yes	float (2)
REAL*8	real		COMP-2	Double Precision	Double Precision	COMP-2	yes	double
INTEGER*4	longint		COMP-4	-	-	COMP-4	yes	long

Fig. 7-7 Data Types that can be Exchanged between FORTRAN and other Components

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

2. When it is passed as a parameter, a float in C is always converted into a double. Thus, a C program cannot pass a float-type parameter to a function in another language; and a program in another language cannot pass a float-type parameter to a C function.

3. A C string is terminated by a byte with the value zero; in the other languages, a string does not have a standard terminator.

C Language Types ⟶ Other Component Types

C	PASCAL+	ICE COBOL	COBOL	BASIC COM	BASIC INT	FORTRAN	DMS SORT (3)
short	integer	COMP	COMP	integer	integer	INTEGER*2	COMP
int	integer	COMP	COMP	integer	integer	INTEGER*2	COMP
long	longint.	-	COMP-4	-	-	INTEGER*4	COMP-4
float (1)	-	-	COMP-1	sing-pr	sing-pr	REAL*4	COMP-1
double	real	-	COMP-2	doub-pr	doub-pr	REAL*8	COMP-2
string (2)	string	string	string	string	string	string	string

Fig. 7-8 Data Types that can be Exchanged between C Language+ and other Components

1. When it is passed as a parameter, a float in C is always converted into a double. Thus, a C program cannot pass a float-type parameter to a function in another language; and a program in another language cannot pass a float-type parameter to a C function.

2. A C string is terminated by a byte with the value zero; in the other languages, a string does not have a standard terminator.

3. All types of ICE COBOL are compatible with SORT.

DMS Types ⟶ Other Component Types

DMS	PASCAL+ (2)	ICE COBOL (1) (2)	COBOL (1) (2)	Comp. BASIC (2)	Interp. BASIC (3)	FORTRAN	SORT (4)	C
Alphanum. strings	Alphanum. strings			Alphanum. strings	Alphanum. strings	Alphanum. strings		Alphanum. strings (6)
COMP	integer			integer	integer	INTEGER*2		int or short
COMP-1	real			Single Precision	Single Precision	REAL*4		float (5)
COMP-2	-			Double Precision	Double Precision	REAL*8		double
COMP-4	-			-	-	INTEGER*4		long

Fig. 7-9 Data Types that can be Exchanged between DMS and other Components

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

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

   - a Positional No Deletion file must be created by the appropriate MCL utility or by a program in the relevant language

   - the DMS cannot carry out deletion operations.

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

   - if the file is created by DMS, the Interpreted BASIC program also accesses deleted or non-existent records, replying not with an error message but with 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 deletion operations.

4. The data which can be exchanged between DMS and SORT is the same as that allowed by COBOL.

5. When it is passed as a parameter, a float in C is always converted into a double. Thus, a C program cannot pass a float-type parameter to a function in another language; and a program in another language cannot pass a float-type parameter to a C function.

6. A C string is terminated by a byte with the value zero; in the other languages, a string does not have a standard terminator.

Types Related to SORT ⟶ Other Component Types

SORT	PASCAL+ (2)	ICE COBOL (1) (2)	COBOL (1) (2)	Comp. BASIC (2)	Interp. BASIC (2)	FORTRAN	DMS (3)	C
Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings	Alphanum. strings (5)
integer (2 bytes)	integer	COMP	COMP	integer	integer	INTEGER*2	COMP	int or short
integer (4 bytes)	longint	-	COMP-4	-	-	INTEGER*4	COMP-4	long
real (4 bytes)	-	-	COMP-1	Single Precision	Single Precision	REAL*4	COMP-1	Float (4)
real (8 bytes)	real	-	COMP-2	Double Precision	Double Precision	REAL*8	COMP-2	Double

Fig. 7-10 Data Types that can be Exchanged between SORT and other Components

1. All types of ICE COBOL 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, Positional No Deletion Pascal+, Sequential ICE COBOL, Sequential COBOL, Sequential Compiled BASIC, and Random Interpreted BASIC.

3. The data which can be exchanged between DMS and SORT is the same as that allowed by COBOL.

4. When it is passed as a parameter, a float in C is always converted into a double. Thus, a C program cannot pass a float-type parameter to a function in another language; and a program in another language cannot pass a float-type parameter to a C function.

5. A C string is terminated by a byte with the value zero; in the other languages, a string does not have a standard terminator.

MULTI-FUNCTIONALITY LIMITS FOR SYSTEMS HAVING 1 MMU

On reading the Chapter "Assigning User Segments" it will be evident that for 1-MMU systems there exist incompatibilities with some application environments.

These environments are mutually exclusive and thus cannot co-exist simultaneously within a system; they can be used separately, however, taking care to initialize the system each time using the appropriate Grandpa configuration file so that Grandpa can activate one or other of the environments. For further details concerning Grandpa configuration files refer to the Chapter "User Activity Activation Modes".

For **stand alone systems** or **application/terminal server systems,** the following components are incompatible:

- Terminal Emulators
- BEAM and MTS application environments
- ONE and NEMOS networks (SNA control networks)
- QUEMAN and TRACE
- LMS and TRACE
- QUEMAN and LMS, which are compatible when using the QUE_LMS module
- Message Switching and BEAM
- NETBIOS and BEAM
- NETBIOS and MTS
- NETBIOS and Message Switching
- MTS client environment programs and graphics functions: the former may use the latter if they are not accessing chained DB.

An MTS application program cannot use the graphics functions provided by the GSP nor, if a chained database has been configured, the PGU. Applications belonging to MTS server environments cannot use the VISA.

For **LAN multiserver systems** the following are incompatible:

- BEAM and MTS application environments, if their respective servers reside on the same system.
- The ONE and NEMOS networks (these are mutually exclusive if they reside on the same network.

Note

1. Terminal Emulators of the type 3270 SNA, 3270 BSC3, 3770 and 2780/3780 must reside on the same system on which the Line Manager resides. This system is identified with the logical name 128.

2. The Network Control Centre (NCC) of the Network Monitoring System must reside on a system fitted with 2 MMUs.

3. For systems with 2 MMUs there are no limitations on multi-functionality regarding the co-existence of environments; the user must, however, respect the limitations regarding the simultaneous presence on the system of an Emulator and the Line Manager.

PART 3 - USER ACTIVITY EXECUTION

INTRODUCTION TO PART 3

Part 3 describes user activities activation and execution phases.

User activities derive this name from the fact that they are allocated in user address space. Olivetti-prepared application environments like Shell, Interpreted BASIC, ICE COBOL etc. are user activities, as well as programs (for example, utilities) written by Olivetti or the user - whether inter-active or not - and user packages.

Part 3 is made up of Chapters 9 and 10.

Chapter 9 is of particular interest to users having to plan the activities of a system. It describes the various activation modes of user activities, and particularly the automatic activation mode.

Chapter 10 describes a number of programming facilities relative to the system start-up and shutdown phases.

8. USER PACKAGE PREPARATION

Only those users who wish to implement new user packages (in addition to those distributed by Olivetti) need read this Chapter.

Information regarding activation of the user packages is provided in the Chapter "User Activity Activation Modes", which should be read by all users.

INTRODUCTION

As already stated in Chapter 1, user packages are sets of Pascal+ functions and/or procedures available to all users.

They are loaded and initialized by Grandpa when the system is switched on, in the address space of its family, and the part of the address space that contains them is shared with all the user families subsequently created. In order to use the services they offer, the applications must simply link the interfaces to them.

Besides the user packages with the above characteristics, there exist special user packages called "MTS user packages", which are also partly handled by MTS. For further details refer to the Section "MTS User Packages" at the end of this Chapter.

WRITING A USER PACKAGE

A user package consists of a set of procedures which implement the functions of the package, and two procedures for its initialization and termination.

There are no particular restrictions concerning the package procedures, which are available to all the user programs until 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 in order to execute a "program" ("stdin" and "stdout", which are normally associated to a work station and are part of the program's execution context); a "module" or a "monitor" does not.

Also the initialization and termination procedures must be single procedures 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 a parameter.

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, effect the following functions:

- 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. Any 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 are, however, ignored by Grandpa.

In addition, if it is required to make the dynamic link mechanism available to users of the package's services, both the initialization and termination procedures must make use of the PMM SwitchPtable primitive in order to:

- Advise the PMM, during the initialization phase, of the address of the table of user package procedures (p_table), by means of appropriate modifications to the m_table of the module calling the user package services. For information on modification of the m-table see the Section "Dynamic Link and its Primitives" in the Chapter "Process and Memory Management Interfaces".

- Restore, during the termination phase, the pre-existing situation relative to the above table.

The initialization/termination procedure cannot be called within a monitor, as the monitor's lock would not be released when the PMM "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, for example, the ZLOC linker's ENTRY command).

LINKING A USER PACKAGE

It is possible to choose to make the static or dynamic link mechanism available to the l_module calling the user package services. This choice is effected by the user who implements the package.

DYNAMIC LINK MECHANISM

If it is required to link the user package services dynamically to an l_module at run-time, it is necessary to prepare the user package as follows:

- Compile it.
- Prepare the source p_table relative to the user package, using Editor.
- Obtain the p_table in object format, and the interface file to the package, using ZSTUB.
- Execute the link between the user package in object format and the p_table output by ZSTUB.

The interface file produced by ZSTUB is used at the linking phase by the l_module using the user package services.

STATIC LINK MECHANISM

When executing the link of a user package, the user who prepares the package should request the linker to generate the symbol table for the set of package procedures (using the linker SYMBOL command).

This means that the package's procedures can be called by any program to which the symbol table has been linked (by means of the linker's INSYM command).

If a package has been linked in this way, the link of any program that uses the package services must be repeated each time that internal modifications are made to the package.

9. USER ACTIVITY ACTIVATION MODES

User activities may be activated in the following modes/times:

- On termination of the system initialization phase, automatically and transparently to the operator.

- On termination of the system initialization phase, at the request of the operator, by means of the environment menus displayed on the screens of the multi-functional work stations.

- On termination of an environment, at the request of the operator, by means of the environment menus displayed on the screens of the multi-functional work stations.

- At the request of the operator, by means of an interactive environment (Shell, BEAM, ...), or one of the language interpreters (BASIC, COBOL ICE, ...).

- At the request of an application, by means of an appropriate primitive.

- On termination of the system (shutdown), automatically and transparently to the operator.

Automatic activation/disactivation of activities is handled by Grandpa, who must thus know their name and characteristics.

This information is supplied to Grandpa by the user who configures the user sub-systems by means of a file known as the Grandpa configuration file.

DESCRIPTION OF USER ACTIVITIES TO BE ACTIVATED BY GRANDPA

By means of the Grandpa configuration file the user may specify, for each activity, the following essential information:

- The pathname of the l-module describing the activity, any initialization parameters required, and the logical name of the activity (only for activities to be activated by the operator).

- The activity type and execution class.

- The name of the user who may activate it, and of the work station by which it may be activated (only for interactive activities).

By means of this file it is also possible to ask Grandpa to execute certain special actions (for example, to obtain the date from the operator of the Master Work Station).

The syntax and conventions to be followed when creating the Grandpa configuration file are described in detail in the MOS System Software Generation and Installation Manual.

The present manual only provides general information on the file contents and an example of the configuration file. It also describes the key words with which the user specifies to Grandpa certain characteristics of the activity (for example, the type).

ACTIVITY TYPES

The activities described in the file may be of the following type:

- Initialization and termination programs
- Interactive and batch programs
- User packages

Initialization and Termination Programs

These are programs that are executed just once in the interval of time between IPL and shutdown: initialization programs when the Master Work Station (MWS) becomes ready, and termination programs when all the other user operations are concluded.

The initialization programs allow the use of double identifying keys: INIT and PINIT. PINIT type activities are activated before the MSW login program; INIT type activities are activated immediately after this program.

Termination programs are identified by the key word TERM.

User Packages

These are identified by means of the PCALL and CALL keys, which distinguish user packages to be activated respectively before and after the MWS login program.

Interactive Programs

As usual, these are programs associated to a work station (for example, environment software). The identifying key of this activity specifies the logical name of the work station/s to which they are associated. TTYx, RTxx and VTxy keys are accepted for identifying local, remote and virtual work stations respectively, as already described in the Section "Identification of Work Stations" of the Chapter "Work Station Management Interfaces".

If only one interactive program is associated to a work station, the latter is known as mono-functional; otherwise it is known as multifunctional.

It is possible to specify, in the configuration file, also the login name of users allowed to activate the interactive programs described in the file and the work station from which they can be activated. Consequently, a work station may appear mono-functional to one user and multi-functional to another. From this point on, when describing "multi-functional" work stations, this term should be understood as referring to a work station that is "multi-functional for the current user".

The logical names of the users, work stations and activities are used by Grandpa for preparation of the environment menus of multifunctional work stations. For further details on these menus consult the Section "Activation/Disactivation of the Application Environments".

Non-interactive Programs

These are, as usual, programs whose activation is not associated to any WS. They are started by Grandpa at initialization time, and terminated at shutdown time. A typical example is the "Transactional Environment" (TE) of a Transaction Handler. Other examples are spooling systems, batch job monitors, file servers, etc.

The identifying key of these programs identifies its execution class.

ACTIVITY EXECUTION CLASSES

For the execution of user activities the following classes are used:

- Server
- Foreground
- Standard
- Background

Each of these is characterized by different priorities and time slices.

Interactive activities are always executed in the "standard" class; non-interactive ones are executed in the class chosen by the user.

The value the system assigns as the activity priority at the ready processes scheduling time is inversely proportionate to the one the user assigns as activity priority using Spawn.

Descriptions are given below of the characteristics of each of the classes, together with information useful to the user for the correct allocation of programs within the class.

Server

This class should be used for the real time activities that must be executed as quickly as possible. In fact this class has the highest priority: only Grandpa is executed with a higher priority. The priorities that may be assigned to the activities belonging to this class range from 500 to 999. The time slice of the CPU assigned to the activities of this class is infinite: when these activities reach the running condition, they continue execution until they suspend or terminate.

The key word to use in the Grandpa configuration file for the insertion of an activity in this execution class is **SERV.**

Foreground

Also this class comprises real time activities, but these are of a lower priority than the Server class activities. The priority that can be assigned to them is in the range 1000 - 1999. The CPU time slice assigned to activities of this class is infinite, as is that assigned to the Server class activities.

The key word to be used in the Grandpa configuration file to insert an activity in this class is **FG** or **PFG** (see Note 1).

Standard

The priority of this class is lower than that of the Foreground; the priority that can be assigned to it ranges from 2000 to 2999. The time slice assigned to the activities executed in this class is 100 milliseconds, so they are handled in time-sharing mode. System activities such as the spooling system should be placed in this class.

The key word to be used in the Grandpa configuration file for the insertion of an activity of this execution class is **START** or **PSTART** (see Note 1).

Background

Also this class comprises time-sharing activities, but these activities have the lowest priority; the priority that can be assigned to activities to be executed in this class starts from 3000. The time slice is 100 milliseconds, like that of the standard class activities. Activities not requiring particular execution speed, for example batch activities in Shell, or statistics routines for system activities, etc. should be placed in this class.

Activities of this class are executed only when there are no executable activities available (families with at least one process ready) belonging to the Standard or Foreground classes.

The key word to be used in the Grandpa configuration file for insertion of an activity of this execution class is **BG** or **PBG** (see Note 1).

Note

1. What has been stated relative to key words for identifying activity types applies equally to key words for identifying classes: presence in the keyword of the Prefix "P" indicates that the activity described must be activated before the Master Work Station becomes ready; absence of the prefix "P" indicates that the activity described must be activated after the Master Work Station becomes ready.

2. It must be emphasized that correct distribution, between the classes available, of the various activities that the user wishes to activate on initialization of the system, is of vital importance in ensuring that the system itself works in a "balanced" manner. In other words, it is the responsibility of the user, who may make use of the Grandpa configuration file for this purpose, to define all the programs that Grandpa must activate, and their insertion in the correct execution class, and thus avoid one class being penalized by others that have more activities and that monopolize use of the CPU.

SPECIAL GRANDPA DIRECTIVES

These directives (key words) make it possible for the user:

- to request Grandpa to execute particular actions

- to transmit particular information to it.

For example, it is possible in this way to:

- specify (using the MASTER and GMASTER directives) the master work station of a system

- request Grandpa (using the DATE directive) to get the date from the operator of the master work station

- request Grandpa (using the NEWVOL directive) to effect all operations that allow the user to remove the system disk before activation of any interactive user program

- specify (using the LOGIN directive) the user-written module for customizing the user identification in the login phase

- request the closing of system activities (shutdown)

- specify the activities to be activated in "unattended" mode (for example, on restarting after a shutdown).

AUTOMATIC HANDLING OF USER ACTIVITIES BY GRANDPA

Grandpa first activates the user activities belonging to the shared user environment (e.g, user packages); it then activates the specific activities of each work station and user.

ACTIVATION OF SHARED USER ACTIVITIES

After reading the configuration file, Grandpa operates as follows:

- It loads and activates the system servers such as, for example, the processes that allow remote file access.

- It loads and activates any user program not associated to any work station that must be activated before the master work station becomes ready. It carries out a similar function for user packages of the PCALL type. Activation of programs and packages is in the following order: PINIT, PCALL, PFG, PSTART, and PBG.

- It checks the existence of the Login database (containing the list of all the names of users authorized to access the system) with the aim of activating (if the database is present) the login program specified during the configuration phase (as described in the next Section).

- It loads and activates the remaining non-interactive programs defined in the file (with the exception of those having the TERM key) and the remaining user packages. The order of activation is as follows: INIT, CALL, FG, START, BG.

The modes for the loading and activation of packages are described in the Section "User Package Activation Characteristics"; those for the loading and activation of programs are described in the Section "Program Activation Characteristics".

USER PACKAGE ACTIVATION CHARACTERISTICS

As already stated in the Chapter "Preparation of a User Package", a user 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 / deactivating a package means calling its initialization / termination procedure; the remaining procedures, which guarantee the package's functions, can be called by any user program in the interval of time between initializing and terminating the package.

Grandpa guarantees correct execution of these phases for the user packages.

Each package is activated twice by Grandpa. The first time (using the PMM Load and Call primitives), when the system is initialized, allows the package to initialize its own global variables. The second time (using the PMM Call and Unload primitives) allows the package to correctly end its processing when system shutdown has been requested.

Grandpa provides a set of information for each package, using the Context and Initialization Parameters described below.

Context Parameters

As they can be called by any program, the procedures which provide the user package services are executed in the Program Context associated to the requesting program. Thus the files, work stations, printers, etc. available to them depend on which program is using them. The initialization and termination procedures of the package are executed in the Program Context of the calling program, which is Grandpa.

Initialization Parameters

The information contained in these parameters is provided to the package initialization procedure by Grandpa.

These parameters are passed to it by means of the "lineParam" Pascal+ implicit variable which contains the string of bytes found in the "initial string" associated to the package in the Grandpa configuration file.

The parameters must be specified in the format in which the procedure 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.

ACTIVATION OF LOGIN PROGRAMS

When Grandpa has prepared the shared user environment, it activates the login program of a work station as soon as one becomes ready. This has the aim of allowing users to start work sessions on the system in controlled mode.

If the login database does not exist (see the Section "Primitives for Accessing the Login Database" in the Chapter "User Activity Characteristics"), the login program is not activated. Grandpa activates the program stored on disk at the installation time with the pathname /IPL/ETC/$LOG.

There are two standard login programs, named LOGS and LOGN. Both carry out the following functions:

- They request, and check existence on the login data base, of the login name and password entered by the user interactively.

- They check that no other user already logged-in with the same login name and password exists in the same local system.

- They return the identity of the user who has performed login to Grandpa if the above checks are completed correctly. This identity is obtained from the login database.

- They highlight the application environment menus

- They communicate to Grandpa the screen splitting/unsplitting request.

- They highlight the application environment menus (only for multi-functional work stations).

The standard login programs differ only in that:

- When LOGS is used, the screens of all the work stations are split.

- When LOGN is used, no work station screens are split.

It is always necessary to associate to the master terminal a login program requiring splitting of the screen.

Warning: If none of the system terminals has the screen split, the functions of the master terminal cannot be used. For this reason it is best to specify Olivetti work stations as master work stations.

In the case of virtual work stations, the login program must be associated only to VT1 terminals during the configuration phase. This avoids the user having to effect multiple login operations at run time (one for each of the virtual work stations associated to the same physical work station).

In the case of remote work stations, the login is effected when they connect to a MOS system in order to operate as the latter's work stations.

A login program is made non-interactive when the user login name is specified in the Grandpa configuration file as an activation parameter (of keyed type, having the key "NAME").

The user has the possibility of personalizing a login program as regards user identification, by providing a user-written module (see the Chapter "Programming Facilities Related to System Start-Up and Shutdown", Section "Writing a User Module for Login Personalization".) The LOGIN directive notifies Grandpa of the pathname of this module.

ACTIVATION/DISACTIVATION OF APPLICATION ENVIRONMENTS VIA MENU

For non-multifunctional work stations, on termination of the login phase, Grandpa automatically activates the application environment (or the user program) which has been associated to the work station in the Grandpa configuration file.

In the case of multi-functional work stations, however, the login program presents the menu of the interactive user environments/programs, in order to allow the user to choose the component to activate.

The menu is constructed bearing in mind what was specified at the Grandpa configuration phase relative to interactive programs that are available to work stations and users.

All the interactive activities having the following characteristics are included in the menu:

- they are associated to the current work station
- the current user of the work station concerned is authorized to access it.

It is possible therefore, for example, to achieve the following situations:

- Any user can select any program from any work station.
- Any user can select defined programs from specified work stations.
- Specified users can select defined programs from any work station.
- Specified users can select defined programs from specified work stations.

When the user requests disactivation of an environment, Grandpa is aware when it terminates.

For multi-functional work stations, Grandpa acts in such a way that the Login program displays the activity selection menu again, thus enabling the user to select another environment (new or equal), or the LOGOUT activity. Grandpa inserts the latter automatically in the menu of all the multifunctional work stations. When it is chosen, it closes the current user's work session and initiates a new login phase.

For non-multifunctional work stations, termination of the environment gives rise to a new login phase.

If the login database is not present, the activities (or the activity selection menus) associated to the terminals are automatically activated by Grandpa.

PROGRAM ACTIVATION CHARACTERISTICS

When Grandpa activates a program:

- It creates a family (using the PMM Spawn primitive) for each program activated, assigning each one a priority and time according to the specifications in its configuration file.

- It loads a program from a file in the address space of the new family (using the PMM Load primitive).

- It creates a process for executing this program (using the PMM Start primitive).

Before effecting these operations, Grandpa temporarily assumes (using the PMM SetIdentity primitive) the identity of the user who has effected the login. The family and the process executing the program to be activated thus inherit this identity. For further details see the Section "Security" in the Chapter "Process and Memory Management Interfaces".

All the programs associated to virtual work stations based on the same physical work station have the same identity.

Grandpa provides the program to be activated with two types of information:

- Context parameters

- Activation parameters.

Execution Context Parameters

The information contained in the execution context guarantees the correct link between the objects used by a program, to which reference is made by logical names, and the physical names of these objects referenced by the system.

The execution context 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 family: so the same program is executed in different contexts if it belongs to different families (for example, when it is executed on different work stations).

The information present in the execution context 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.

- **localroot** is the name of the memory volume, created in memory when the operating system is loaded, present in each local 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 as follows:

  . 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/BKGn, 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. Any anomalies 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 (whether virtual or not) associated to the program. If the screen was split at log-in time, the identifier of the upper window is associated to this parameter. 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, ... , sysprtn** are the names of the "n" system printers available to the user.

"workdir", "workst",... can be connected using the context system identifier as "parent directory". See the FS "Connect" primitive in the Chapter "File System Management Interfaces".

The system identifier of the program directory is also stored in the execution context of a program belonging to a program directory. The files belonging to the program directory can then be connected using the context system identifier as "parent directory".

Initialization Parameters

These parameters are provided by Grandpa to the programs which are being activated without carrying out any checks. They are passed to it by means of the Pascal+ "lineParam" implicit variable which contains the string of bytes found in the "initial string" associated to the program in the Grandpa configuration file.

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.

Assigning the Owner to the Families and the Processes

The owner of the programs activated by Grandpa prior to the "ready" state of the master work station is the root user; interactive or batch programs subsequently activated by a user are owned by the user who effected the login.

If the Login database does not exist, all the programs are owned by the root user.

The "termination" programs are owned by the user who requested the shutdown.

Program Execution Characteristics

A program activated by Grandpa can:

- acquire the activation and context 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 system closure to Grandpa.

SHUTDOWN MODES AND FUNCTIONS

Shutdown is achieved in the following ways:

soft: all the activities in the system are terminated ONLY if no interactive program is active.

hard: unconditional shutdown: all the active programs are terminated.

It is in any case possible to define an interval time (timeout) that Grandpa will wait before terminating the activities in the system.

Shutdown Modes

The above functions are achieved when:

- the SHUTDOWN entry associated to the menu of a work-station in the configuration file is used, or

- the specific value of the completion code on termination of a program started by Grandpa is returned to it. If the shutdown is requested by program, the user can also request system sutdown with power off, and a subsequent unattended start-up at a certain time. See the Chapter "Programming Facilities Related to System Start-Up and Shutdown".

The shutdown procedure may only be requested by a program activated for a user belonging to the SYSTEM group. That is:

- the SHUTDOWN entry can only be chosen by a SYSTEM group user

- the program requiring shutdown on its termination has to be started by a SYSTEM group user.

If the shutdown request is triggered by a non-interactive program or by a program with the keyword PINIT, after the system has closed down a "blinking" number "77" is displayed on the screen.

When shutdown cannot be executed (because the user has not the appropriate rights or there are interactive programs running in the case of soft shutdown) diagnostics messages are:

- displayed on the terminal, if the shutdown has been requested by an operator

- sent to the Master work station, if the shutdown has been requested by a program.

Shutdown Functions

The Shutdown function basically "undoes" what was done by Grandpa during the user environment initialization phase. That is:

- it checks the identity of the operator asking for the system shutdown

- it asks the operator for the time Grandpa waits for, before starting any shutdown operation

- it terminates all the programs that are still working "softly", i.e. using the PMM "Interrupt" primitive. If no interrupt handling is provided by the affected program, this results in its termination. Otherwise, ad hoc operations may be executed by the program in order to properly handle the interrupt

- it terminates and unloads all the user packages, in the reverse order of their activation; "terminating" a user package means calling its termination procedure by means of a PMM "Call" primitive

- it loads and starts each user termination program in the same order as described in the configuration file, and waits for its termination

- it notifies the operator that the system has been properly shut down.

Note that all the operations described above are performed by Grandpa independently of the shutdown mode.

The shutdown function may be called by any program (in theory), but should be used only by an ad hoc program - possibly also called "shutdown" - that is associated with one or more work stations.

GRANDPA CONFIGURATION FILE AND RELATIVE OPERATIONS: AN EXAMPLE

An example is given below of the Grandpa configuration file configured for a system with four work stations, one of which is the master, and where activities can be carried out in different application environments; the system spooler is used and batch activities are executed.

The configuration file is as follows:

```
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:

- It associates the following activities to the master (TTYA):
  - Definition of the date and time for the system clock (DATE).
  - Substitution of the system disk, if required (NEWVOL).
  - System shutdown request, if required (SHUTDOWN).
  - Display of warning messages on the 25th line of the screen.
  - Shell application environment (MCL=/IPL/SYS/$VSM)
  - Interpreted BASIC application environment (BASIC=/IPL/DPC/CMD/BASIC).
- It activates the login program stored in the file $LOG (which must contain the "split" request for all the terminals). The standard user identification is carried out, as the LOGIN directive has not been specified in the Grandpa configuration file for TTYA.
- It loads the queue manager (IPL/$QM/CODES/QUEMAN) (giving it the PAR parameter list).

- It activates the spooling activity manager (/IPL/SP/SPGPA) in the Standard execution class.

- It activates the batch job manager (/IPL/BE/BATCHGPA) in the Background execution class.

- It associates the interpreted BASIC application environment (BASIC=/IPL/DPC/CMD/BASIC) to the terminal identified as TTYB.

- It associates the application environments COBOL ICE (ICE=/IPL/DPC/ICRTS) and Shell (MCL=/IPL/SYS/$VSM) to the terminal identified as TTYC.

- It associates the application environments Shell (MCL=/IPL/SYS/$VSM) and interpreted BASIC (BASIC=/IPL/DPC/CMD/BASIC) to the terminal identified as TTYD.

When the system is switched on, the user's login prompt will first be displayed on the master work station, and then on all the terminals.

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 re-displays the select activity menu on the terminal, allowing the user to enter another application environment.

In order to free the terminal so that it can be used by other users, the user must select the LOGOUT activity.

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.

INTERACTIVE ACTIVATION OF PROGRAMS, BY MEANS OF SHELL

Programs written by the user that were not activated at the time of system initialization by Grandpa may be activated using application environment software distributed by Olivetti, such as Shell, BEAM, ... The following description refers to Shell.

Shell creates a family for each program or program directory that must be activated whether in an interactive or batch environment; the program to be executed is then loaded into it and is executed.

The programs whose execution is requested in batch mode (BM command) are executed in the background execution class, under the control of the Batch Monitor. Shell passes a keyed parameter (with the "$batch" key) to them, so that the activity can check the means of activation (whether batch or interactive).

The family created will be the son of the Shell family associated to the work station. It will inherit from the father family the identity of the user owning it and the context of execution. Both will be assigned to any program loaded into it and executed. For the execution context, see the Section "Execution Context Parameters".

By using the special Shell built-in functions, the user may effect the following functions before a program is activated:

- modification of certain context parameters (e.g, the working directory) for the program that it is intended to activate

- insertion of new context parameters (e.g, file connections) relative to the execution of this program.

Shell makes it possible for the user also to specify the activation parameters of an activity when its execution is requested.

It is possible to define the activation parameters and the built-in functions for context parameter modification/insertion in a procedure written in the MOS Command Language (MCL), by calling which the user may activate a program in the desired context.

ASSIGNING A VALUE TO THE WORKING DIRECTORY

This may be carried out using the Shell built-in command SETWDIR, which places in the context of the family the system identifier (system_id) of the working directory specified in the command.

To activate a program, therefore, by making a set of files available to it, it is sufficient for the user to position himself in the directory that contains them by using the SETWDIR command, and then to activate the program. The program will then be able to access the files contained in that directory, by referencing them with their partial pathname. The program also has the possibility of accessing any other file, referencing it with its complete path name (whose first character is "/").

RE-DIRECTION OF STANDARD INPUT AND OUTPUT UNITS

The standard input and output can be redirected (thus altering the values of the stdin and stdout parameters in the context of the program being activated), using the symbols "<" and ">" respectively. For the stdin and stdout parameters see the Section "Program Activation Characteristics".

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.

CONNECTING TO FILES

The CONN built-in command makes it possible to perform connections; that is, to assign system identifiers to the files referenced in a program by means of logical names. Using these identifiers, stored by the command in the program context of the Shell son family, the runtime supports of the programming languages may execute I/O operations on the files referenced within the programs using logical names.

For files belonging to program directories, automatic connections between the logical names (used by the program) and the file identifiers (used by the system) are guaranteed by MOS. Use of the CONN command is not, however, prohibited; if it is used, the connections specified by it are executed instead of the automatic connections guaranteed by the program directory. It is thus possible to define the working environment of the program, incrementing it or modifying the set of objects available to the program. The program directory provides a static environment; the CONN enables it to be handled dynamically.

CONNECTING TO SPOOLING SYSTEM

The CONSP built-in command makes it possible for a file, produced by a user program, to be automatically output to a spool class when the program that created it has terminated.

EXAMPLE OF ACTIVATION OF A PASCAL+ PROGRAM

Once an executable Pascal+ program has been stored in a file, the user may activate it simply by specifying its name (e.g, "file-name.out" name), preceded by the pathname of the directory containing the file.

If the references to the files in the program are by logical names, instead of using the complete pathname, it is necessary - before activating the program - to effect the connection of the logical name by using the CONN command.

For example, if a program accesses the general "ARCHIVES" file:

```
reset (f, 'ARCHIVES')
```

it is necessary, before activating it, to effect the command:

```
CONN ARCHIVES /IPL/USR/ .../file name
```

REPLY CODES RETURNED BY THE PASCAL+ RUN TIME SUPPORT

On completion of program execution, Run Time Support sets the MCL "%STATUS" variable at one of the following reply code values (in the standard way):

0 : The program has terminated correctly. This value is returned by Run Time Support only if the program terminates by means of the "END" statement of a Pascal+ program.

1 : Not used.

2 : During execution of the program one or more "range check" values, or I/O errors, have been detected (non-existent file reset, use of I/O functions incompatible with type of file ...). In this case program termination is forced.

ACTIVATION OF PROGRAMS BY OTHER PROGRAMS

Programs activated by Shell or Grandpa may in turn activate programs by means of the EXEC primitive. See the Chapter "MOS Application Services", Section "Execution of MCL Activities from Shell or Grandpa". The activating program is suspended until the activated program has terminated.

10. PROGRAMMING FACILITIES RELATED TO SYSTEM START-UP AND SHUTDOWN

WRITING A USER MODULE FOR LOGIN PERSONALIZATION

If it is required to personalize the functions of user identification, it is necessary to specify, in the Grandpa configuration file, the pathname of a user-written module which replaces the user identification functions provided by the standard login programs. This pathname must be specified in the form of a keyed parameter. The key to be used is "CALL".

Even if this module is present, the check that there is no other user already logged-in with the same identity in the same local system is still effected by the standard login program, as well as the environment menu management.

The user module is activated in the same local system by the login program using the PMM "Call" primitive.

When preparing this module it is necessary to bear in mind the following:

- Its initialization parameters
- Its reply codes
- Its address space
- The primitives for accessing the login database.

INITIALIZATION PARAMETERS

The login program passes the following parameters to the user-written module:

- A positional parameter that specifies the name of the user previously logged in. This is not meaningful if the first log-in after system start-up is being effected.
- An optional keyed parameter, having the key "NAME" and the user's name value. This parameter makes it possible to make the login program non-interactive, and it is passed to the module only if it has been configured in the Grandpa file.

REPLY CODE VALUES

The user module should return the following information to the calling program:

- A reply code with the value 127 if the log-in operations have been effected correctly; the value 2 if they have been effected incorrectly (in which case the following data is irrelevant).

- The identity of the logged-in user, structured as follows:

```
info =  record
        case integer of
             1 : ( string : packed array [1..12] of char );
             2 : ( identity : T_identity;
                      name : T_login;

         end;
```

where:

info.identity is the identity (made up of two integers) that the system associates to the user

info.name is the user login name.

In order to communicate this information to the login program, the module written by the user may use the PMM "Halt" primitive or the function "stdenv.quit" of the Pascal+ language.

To use this function the following instructions must be placed in the program:

```
Import quit from stdenv;
procedure stdenv.quit (var msg  : packed array [ 1 ..u ] of char;
                           cc   : integer); definition;
```

Whether the parameters are passed by means of stdenv.quit or Halt, the parameter string to be passed must be compiled using the parameter formatting primitives described in the Chapter "Process and Memory Management Interfaces", Section "Parameter Passing".

Special system primitives, described in the next section, make it possible to obtain the identity of the user from the login database, and to carry out other functions relevant to writing a personalized login module.

USER-WRITTEN MODULE ADDRESS SPACE

This module is loaded in the calling program address space. It must be allocated in the logical segments specified as "free user segments" in the Chapter "Assigning User Segments".

PRIMITIVES FOR ACCESSING THE LOGIN DATABASE

The primitives described in this section allow the login database to be accessed in order to:

- obtain the identity of the user whose name is specified (UserByName)
- check that the user whose identity is provided is not currently logged-in to the system (GetLogUser)
- obtain information on the last login of a given user (GetLastLog).

The login database is made up of a number of files, some of which are in the /IPL/ETC directory, some in the memory volume (in the /TMP directory). The former contain permanent information such as the identity of users; the latter contain temporary information such as that concerning users currently logged-in.

To install the login database the MKLOGIN utility is available to the root user.

PASCAL+ INTERFACE

In order to use the above functions it is necessary to include the following files in the source program:

PROT.d
PROT.i
ulog.d

How to link the above functions is described in the Manual PASCAL+, Program Preparation.

Definition of Types

```
type
        T_owner = MININT .. MAXINT

        T_identity = packed record
                        primary   : T_owner;
                        secondary : T_owner;
                        end;

        T_login = packed record [1 .. 8] of char;

        T_logstr = packed array [1 .. 8] of char;

        T_utmp = packed record
                  login    : T_login;
                  Devname  : T_login;
                  Date     : longinteger;
                  end;

        T_UserEntry = record
                      loginName  : T_login;
                      password   : T_logstr;
                      userIdent  : T_identity;
                      reservNum  : integer;
                      reservStr  : T_logstr;
                      userName   : T_logstr;
                      loginDir   : T_logstr;
                      shellfile  : T_logstr;
                      end;
```

This function returns information about a user's last login.

Function Description

```
function   GetLastLog (     name    : T_identity;
                        var termstr : T_login;
                        var date    : longinteger ) : integer;
```

PARAMETER	DESCRIPTION
INPUT	
name	Name of the user on which it is required to obtain login information.
OUTPUT	
termstr	Name of the terminal from which the user identified by means of "name" effected the last login.
date	Date, expressed in milliseconds, starting from 1/1/1980, of the last login effected by the user identified by "name".

Reply Codes

The following table lists the integers returned by the function.

VALUE	MEANING
0	Operation effected correctly
5(RDERR)	Error in reading the disk file belonging to the login database, from which the primitive obtains the required information.
7(NUSR)	The user does not exist or he never logged into the system.
16(NLOG)	The disk file belonging to the login database, from which the primitive obtains the information required, does not exist.

This function checks if a user is already logged in.

Function Definition

```
function GetLogUser (    name      : T_identity;
                      var userEntry : T_utmp) : integer
```

PARAMETER	DESCRIPTION
INPUT	
name	This parameter specifies the name and the identity of the user it is intended to check.
OUTPUT	
userEntry	If the user, identified by the parameter "name", is logged in, contains the following information: - the user login name - the name of the terminal used - the log-in date, expressed in milliseconds starting from 1/1/1980.

Reply Codes

The following table lists the integers returned by the function.

VALUE	MEANING
0	Operation correctly executed.
5(RDERR)	Error in reading the disk file belonging to the login database, from which the primitive obtains the required information.

>>

VALUE	MEANING
7(NOUSR)	User not logged in.
16(NLOG)	The temporary file on the login database, from which the primitive obtains the information required by the user, does not exist.

This function returns the system identifier of the user whose login name is provided.

Function Definition

```
function        UserByName ( var name      : packed array [1 ..8] of
                                               char;
                             var userEntry : T_userentry ) : integer;
```

PARAMETER	DESCRIPTION
INPUT	
name	Null-padded string specifying the login name for the user being looked-up.
OUTPUT	
userentry	Record containing the following information, in the order shown:
loginName	Login name.
password	Password.
userIdent	Identity.
reservNum	Reserved for future use.
reservStr	Reserved for future use.
userName	Name of the login directory.
loginDir	User login name.
shellfile	Name of the file containing Shell.
	See also "Characteristics".

Reply Codes

The following table lists the integers returned by the function.

VALUE	MEANING
0	Operation correctly executed.

Characteristics

1. The identity of the user comprises: primary_id and secondary_id. Refer also to the Section "Security Primitives" in the Chapter "Process and Memory Management Interfaces".

2. The password is encrypted.

3. The login directory is the user's home directory.

PROGRAMMING SHUTDOWNS

In order to communicate this to Grandpa, the terminating program can use the PMM "Halt" primitive, or the function "stdenv.quit" of the Pascal+ language.

The following completion code values are allowed:

CODE	MEANING
128	Hard Shutdown, executed with a timeout of 90 seconds.
129	As above plus immediate system re-start.
130	Allows specifying both the type of shutdown (hard/soft) and the timeout. This information must be inserted in the character string that the program returns when it quits or halts. This string must comprise: - a positional parameter specifying waiting time - the keyed parameter MODE specifying HARD or SOFT shutdown. If these parameters are not supplied, default values are 90 seconds and HARD.

>>

CODE	MEANING
131	Shutdown and power off. It is possible to define shutdown type and waiting time. This information must be inserted in the character string that the program returns when it quits or halts. This string must comprise: - a positional parameter specifying waiting time - the keyed parameter MODE specifying HARD or SOFT shutdown. If these parameters are not supplied, default values are 90 seconds and HARD mode.
132	Enables shutdown, machine start-up at a pre-defined time and power-down to be requested; it is also possible to define mode type and waiting time. This information must be inserted in the character string that the program returns when it quits or halts. This string must comprise: - the keyed parameters RDATE and RTIME specifying date and time, in the form MM/DD/YY and HH/MM/SS (numeric strings with a slash between each pair of digits). - a positional parameter specifying waiting time. - the keyed parameter MODE specifying HARD or SOFT shutdown. If the last two parameters are not supplied, default values are 90 seconds and HARD; the former two are mandatory.

To pass these values using the Pascal+ "stdenv.quit" procedure, the following instructions should be inserted 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 parameter string; the variable "cc" contains one of the above reply codes.

Whether the parameters are passed by means of stdenv.quit or Halt, the parameter string to be passed must be compiled using the parameter formatting primitives described in the Chapter "Process and Memory Management Interfaces", Section "Parameter Passing".

Full information on the Halt primitive and the Parameter Formatting primitives can be found in the PMM and Driver Primitives, Reference Manual.

A. 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.

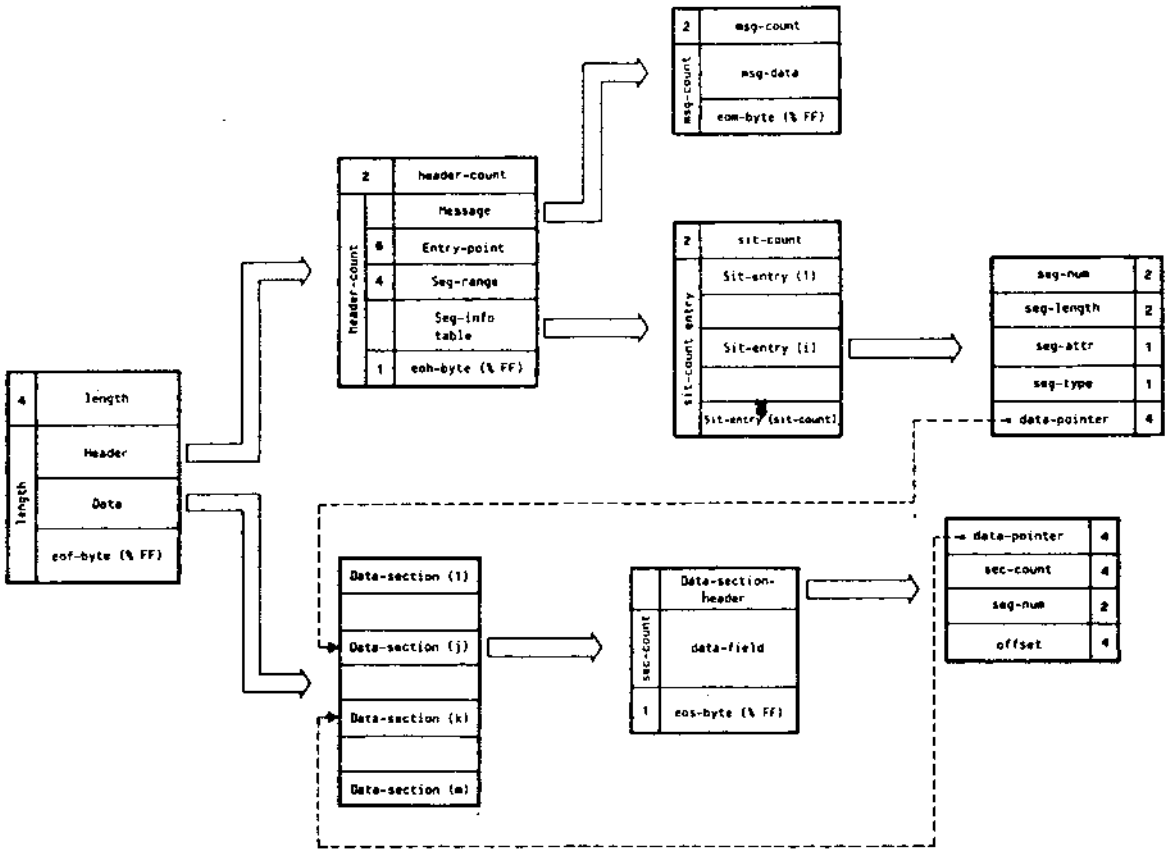


Fig. 2-3 Format of an l_module

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 this manual, and the HEXED utility in the Manual MOS System Software Maintenance Tools, User Guide.

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 OLINK Linker, User Guide and PASCAL+ Program Preparation).
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 to initialize the segments used by the l-module. The user should bear in mind that not all the segments can be initialized and also that each segment may not be initialized 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 OLINK Linker, User Guide and PASCAL+ Program Preparation).

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 OLINK Linker, User Guide and PASCAL+ Program Preparation).

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, 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.

B. M5LDUMP UTILITY

The M5LDUMP utility reads a MOS l-module (see Appendix C) identified by the pathname, and the following information is output:

- The length and header of the l-module
- The DATA SECTION headers contained in every segment followed by the "data" part containing the related data/code; at the linking phase it is possible to insert in the same segment sections having different contents
- Additional information present in the file

The output produced is partly in symbolic format and partly in hexadecimal format.

The "execute" access right to the father directory and the "read" access right to the file containing the l-module are required for execution of this utility.

ACTIVATION

The utility call syntax is as follows:

M5LDUMP filename ["H/"h]

---

where:

file-name identifies the pathname of the l-module. It may be expressed either as an absolute pathname, or as a pathname relative to the working directory of the station that activated the command.

H or **h** is an option that limits the display of the l-module to the DATA SECTION header, without the "data" part.

When M5LDUMP is activated it checks that the pathname specified relates to an l-module: otherwise it sends a warning to the user.

EXAMPLE

```
 OLIVETTI MSLDUMP -- REL 0.1
  DUMPING FILE : SHDATE

<LENGTH>                  4074
<HEADER>
    <HEADER_COUNT>        71
    <MESSAGE>
        <MSGCOUNT>        23
        <MSGDATA>         .. -SHDATE-D60-Jun11-19
        <EOMBYTE>         ##FF

    <ENTRY_POINT> :       <SEGNUM>  = 51     <OFFSET> = 0
    <SEG_RANGE>   :       <SEG_LOW> = 50     <SEG_UP>  = 63


    <SEG_INFO_TABLE>
       <SIT_COUNT>    = 3
       <SIT_ENTRY> ..= <SEGNUM> <SEGLENGTH> <SEGATTR> <SEGTYPE> <DATAPOINTER>
            0             50         6         0         5             75
            1             51        15         1         2            294
            2             63         0        32         3              0
    <EOMBYTE> = ##FF


<DATA SECTION> at 75
    <DATA_POINTER>    = 0
    <SEC_COUNT>       = 204
    <SEG_NUM>         = 50
    <OFFSET>          = 1158
    <EOSBYTE>         = ##FF

0000 0000 0000 00CC  0032 0000 0486 0000
0000 0000 0000 4000  0000 4000 00C0 0000
0000 0000 0000 0000  0000 0000 0000 2049
4E56 414C 4944 204F  5054 494F 4E00 1F1C
1F1E 1F1E 1F1F 1E1F  1E1F 1F10 1F1E 1F1E
1F1F 1E1F 1E1F 008A  00C6 00D2 00DE 50EA
00F6 0102 012E 013A  0146 0152 015E 016A
0176 0182 018E 019A  01A6 0182 0280 02BC
          .                    .
          .                    .
          .                    .
```

```
                .       .       .
                .       .       .
                .       .       .

6572 726F 7220 2020  5072 6F67 7261 6D20
7465 726D 696E 6174  6564 7374

(DATA SECTION) at 294
    (DATA_POINTER)    = 0
    (SEC_COUNT)       = 3764
    (SEG_NUM)         = 51
    (OFFSET)          = 0
    (EOSBYTE)         = ##FF

0000 0000 0000 0E84  0033 0000 0000 5E08
B300 0006 030F 005C  0DE5 005C 5C09 0203
B200 0000 5F00 B300  0A8E 4005 B200 0008
007F 5F00 B300 0024  5006 B200 0012 5402
B200 0012 7604 B200  0016 5F00 B300 0108
7602 B200 0016 7604  B200 0022 5F00 B300
05DE 140C 246F 7074  500C B200 000A 5402
B200 0000 6104 B200  0004 6105 B200 0006

                .       .       .
                .       .       .
                .       .       .

5E08 0900 1408 0078  0000 5E08 0900 1408
0078 0008 5E08 0900  1408 0078 0050 5E08
0900 1408 0078 00E0  5E08 0900 1408 0078
0110 5E08 0900 1408  0078 0170 5E08 0900
1408 0078 0180 5E08  0900 1408 0078 0188
5E08 0900 1408 0078  01C0 5E08 0900 1408
0078 01C8 5E08 0900  1408 0078 01D0 5E08
0900 1408 0078 01D8  5E08 0900 1408 0078


01E0 5E08 0900 1408  0078 01E8 5E08 0900
1408 0078 01F0 5E08  0900 1408 0078 01F8
5E08 0900 1408 0078  0200 5E08 0900 1408
0078 0208

    (EOFBYTE) = FF

( ADDITIONAL INFORMATION FROM POS 4073 TO POS 4095
FF00 0000 0000 0000  0000 0000 0000 0000
0000 0000 0000

(                    MSLDUMP COMPLETE
```

C. DMPRINT UTILITY

This utility interprets, and directs to the standard output unit the contents of a dump file either interactively or in batch mode. The dump file contains the image of all the user segments, and is generated after execution of the "EnableDump" and "memoryDump" primitives described in the Chapter "MOS Application Services".

ACTIVATION

The DMPRINT utility is activated in the Shell environment by entering:

```
DMPRINT filename ["P/"p]
```

where:

filename is the name of the dump file ($DUMP)

"P" or "p" is an option parameter which, if specified, directs the contents of $DUMP to the standard output unit, without requiring operator intervention. If it is omitted, operator intervention is required.

CHARACTERISTICS

If no option is specified, the following information is sent to the standard output unit, page by page:

- A header containing:
  - the date (day/hour/minute) of the dump
  - the data segments
- A segment descriptor for each segment containing:
  - the segment number
  - the segment type
  - the segment length.
- The offset and the contents of the data segment in both hexadecimal and 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 end of 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.

If the option parameter (P/p) is specified, the output stream is directed to the standard output unit in continuous mode and the whole file is dumped, with no questions being put to the operator. This mode is useful when the standard output is re-directed to a disk file or a system printer.

DIAGNOSTICS

All error messages, like all other DMPRINT messages, are inserted in the centralized message database (CEM), in the CE2834 class.

A list is given below of the error messages; the text of each message is preceded by a numeric code that identifies it within the class.

---

57 USAGE: = DMPRINT DUMP PATHNAME x

This message is issued when an incorrect option is entered (e.g."x"); the option is displayed in the message.

---

60 ERROR: NOT A DUMP FILE

This message is issued when the file specified at the time of activating DMPRINT is not a dump file.

---

65 ERROR = INCONSISTENT DATA

This message is issued when the contents of the dump file cannot be interpreted.

D. MOS SUBSYSTEMS AND SCREEN TYPES

The various MOS subsystems allow all types of screens, which can be connected to the MOS 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 **CONTROL 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.

Any 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.

ICE COBOL

The ICE COBOL 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 ICE COBOL 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. Any error messages issued by ICRTS are compatible with any type of screen and format.

The services of the interactive ICE COBOL 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. Any error messages present 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).

Any 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.

Any 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.

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

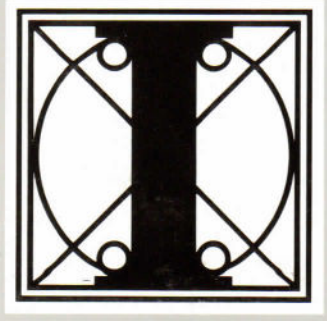
Programs in the MTS environment configuration can be executed with any type of screen and format. Formats cannot, however, be changed dynamically: the format must be selected before the program is activated via Grandpa or Shell.

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.

Code 4041670 S (0)
Printed in Italy



olivetti