The CPOMS Persistent Object Management System
Persistent Programming Research Group

A.L. Brown
Department of Computational Science
University of St Andrews
North Haugh
St Andrews KY16 9SS.

W.P. Cockshott
Department of Computing Science
University of Glasgow
14, Lilybank Gdns
Glasgow G12 8QQ.

Persistent Programming Research Report 13

Contents

Chapter

1 Introduction

2 Pids, lons and the PIDLAM
2.1 When are pids translated into lons
2.2 The organisation of the PIDLAM

3 External Addressing
3.1 The UNIX file system
3.2 Databases
3.3 The structure of a pid
3.4 The Partitions file
3.5 Pids, partitions and index files
3.6 The Directory file

4 Internal support for external addressing
4.1 Recording opened databases
4.2 Database file descriptors
4.3 The Partition to Database and Index Map

5 Summary of external addressing

6 Opening a database
6.1 Addressing using a database name
6.2 The database root object
6.3 How an open uses the root object
6.4 Recursive opening of databases
6.5 Automatic recovery used by open

7 Creating a database
7.1 Initialising the address mappings
7.2 Creating the database's files
7.3 Opening the new database
7.4 Automatic recovery used by create

8 Committing changes to the persistent store
8.1 Alternative commit algorithms
8.2 The commit algorithm used by the CPOMS
8.2.1 Finding all the open databases
8.2.2 Check all the imported objects
8.2.3 Allocate pids to new objects
8.2.4 Copying objects to the persistent store
8.2.5 Resetting the change markers
8.2.6 The result of commit

9 Automatic recovery used by commit
9.1 Before look files
9.1.1 Finding the participating databases
9.1.2 Check all the imported objects
9.1.3 Allocating pids to new objects
9.1.4 Copying objects to the persistent store
9.2 Invoking the automatic recovery mechanism
9.3 Automatic recovery invoked during a commit
9.3.1 Finding the participating databases
9.3.2 Check all the imported objects
9.3.3 Allocating pids to new objects
9.3.4 Copying objects to the persistent store
9.4 Automatic recovery invoked during an open
9.4.1 Analysing the before look file header
9.4.2 Recovery actions for status value 0
9.4.3 Recovery actions for status value 1
9.4.4 Recovery actions for status value 2
9.4.5 Recovery actions for status value 3
9.4.6 Tidying up after the recovery actions
9.5 Automatic recovery explicitly invoked

Appendices

1 The CPOMS interface to the UNIX file system

2 The runtime initialisation of the CPOMS

3 Error handling and reporting by the CPOMS

4 The CPOMS error records

Preface

This manual presents the design of the persistent object management system that is used in distributed copies of PS-algol released in early 1985. It is hoped that the report will be of help to people considering porting PS-algol to new machines as well as to people wishing to understand our work. However, people intending to port PS-algol should contact the Persistent Programming Research Group for information on reports of porting in progress and for details of changes in the abstract machine.

This report describes the persistent object manager that is written in C and used by the PS-algol interpreter written in C. Implementations have been completed for DEC VAX computers running Berkeley UNIX 4.2 and for the ICL PERQ computer running PNX. An implementation is in progress for the ICL 2900 and 3900 series computers running VME.

The reader of this report is advised to read it in conjunction with PPR11 "The PS-algol Abstract Machine Manual" and PPR12 "The PS-algol Reference Manual : second edition" which describe different aspects of the same release of PS-algol and which will be published at the same time.

For further information please contact the addresses given above.

1 Introduction

The CPOMS is a persistent object management system written in C that is used by the PS-algol interpreter written in C. Its function is to provide the interpreter with an implementation of a transactionally secure persistent store as described by the high level user interface in chapter 12 of the PS-algol Reference Manual. There are four distinct interfaces to the CPOMS:

a) The high level interface seen by the user.
b) A lower level interface to implement parts of the higher level one.
c) Several internal C-routines shared between the CPOMS and the interpreter.
d) A small set of C-routines to access the UNIX file system.

The purpose of this document is to describe in detail how these interfaces operate in terms of the underlying architecture of the CPOMS.

The architecture of the CPOMS supports several address mappings for special pointers called persistent identifiers, hereafter called pids. It also supports a concurrency control mechanism over collections of objects in the persistent store, called databases and the movement of these objects between their databases and the heaps of PS-algol programs.

The architecture of the CPOMS is further influenced by the fact that the CPOMS is part of the PS-algol interpreter and therefore there is one incarnation of it for every active PS-algol program. A description of the CPOMS architecture can therefore be divided into three parts:

a) The internally held address maps for each PS-algol program.
b) The structure of the data held on disk.
c) How these components interact with each other to provide a secure transaction mechanism.

It should be noted that the following description of the CPOMS relates to an implementation under the UNIX operating system. Further, the design was a first iteration aimed at producing a functionally correct implementation hence some of the algorithms are not yet efficiently tuned.

2 Pids, lons and the PIDLAM

The persistent store implemented by the CPOMS is an extremely large PS-algol heap with objects being addressed via the special pointers called pids. These pids are identical in size to the normal pointers used by the PS-algol interpreter but have their most significant bit set. In this document the normal pointers will be referred to as lons, local object numbers.

Since pids are pointers to objects held outwith the interpreter's heap, the objects they refer to cannot be directly addressed by the interpreter. Hence if a pointer may be a pid it is tested before it is used.
If the pointer is a pid the object referred to by the pid is copied into the interpreter's heap and given a lon. Thereafter the lon of the copy is used in place of the pid. To prevent more than one copy being made of an object a data structure called the PIDLAM, pid to local address map, is kept. There is one PIDLAM for every incarnation of the CPOMS since the mappings it records may be different for every invocation of a PS-algol program. When a pid is first used and the object it refers to copied, the pid is entered into the PIDLAM along with the lon of the copy. Therefore, if the pid is used again the lon of the copy can be quickly found from the PIDLAM and used in its place.

2.1 When are pids translated into lons

To make the translation of pids into lons efficient a set of rules is required to ensure that only the minimum number of checks for pids and translations of pids are performed. The rules devised were specifically designed to suit the PS-algol abstract machine architecture. The recursive application of these rules is sufficient to guarantee that the interpreter will never encounter a pid in the frame of an executing procedure or block. If a frame does contain a pid then the frame must have been imported from the persistent store and so cannot belong to an executing procedure or block. Consequently abstract machine instructions that operate on the local stack frame need never deal with pids. The rules are as follows:

a) Once a pid has been translated into a lon, the location that held the pid is overwritten with the lon. Future use of this location should then yield a lon and so avoid the need for another translation.
b) When a pid refers to a code vector the pointer fields for the procedure and string vectors of the code vector are also translated into lons. This allows the store closure and literal string instructions to use these fields without needing to check them.
c) When a pid refers to a frame the pointers in the frame's display are translated into lons. Therefore instructions that use a display can do so without needing to check the display for pids. Also when a frame is created the display it copies from its parent frame will never contain pids.
d) Whenever a pointer value is placed on the local stack frame it is checked and if it is a pid it is translated into a lon. Therefore all instructions using the local stack frame need never check for pids.
e) If the trademark field of a structure needs to be dereferenced it is checked to see if the pointer is a pid. The only operation performed on a trademark field is string comparison with a value on the local stack frame. A trademark is never copied onto the local stack frame.
f) Procedure closures are treated as two distinct pointer values. However since procedure closures are always used as a single unit both pointers will be either pids or lons.
g) No lon in the interpreter's heap will ever be overwritten by the pid it translates to. When a commit is copying an object back to its database any lons it contains are translated and overwritten in the I/O buffers during the copy. This prevents the effect of these rules being compromised.

2.2 The organisation of the PIDLAM

The PIDLAM records all mappings of pids to lons for its incarnation of the CPOMS. The PIDLAM can be indexed by both pids and lons. Indexing via pids is necessary to support the pid translation described above. Indexing via lons is used by a commit to translate all lons in an object into pids whilst copying the object back to its database.

The format of the PIDLAM currently maintained by the CPOMS is as follows.

```
         ---------------------------------
     ... | pointer to a PIDLAM table entry | ...
         ---------------------------------
                      4 bytes
```

Figure 2.1 A hash vector entry

```
-------------------------------------------------------
| heap   | dummy | next  | next  |     |     | part of |
|        | trade | lon   | pid   | lon | pid | disk    |
| header | mark  | entry | entry |     |     | address |
-------------------------------------------------------
                   all 4 byte fields
```

Figure 2.2 A PIDLAM table entry

Two PS-algol vectors of pointers are held, one to be indexed by hashing on the pid and the other by hashing on the lon. The actual PIDLAM entries are held in doubly linked lists of PS-algol structures pointed at by the elements of the two vectors. One link in these structures is to a linked list of PIDLAM entries whose pids all have the same hash index. Similarly a link is provided for a list of PIDLAM entries whose lons had the same hash index when they were inserted.

In addition to the lon, pid and list pointers in a PIDLAM entry part of the disk address of the object referred to by the pid is also recorded. This is in effect a cache to speed up calculation of the full disk address of the object during a commit. The disk address of an object will never change while being accessed by the CPOMS. Disk addresses are described in chapter 3.

Access routines to the PIDLAM are provided to support searching for a pid using a lon, searching for a lon using a pid and searching for the cached disk address using a pid.

Four update routines are also provided to operate on the PIDLAM. The first of these is used to recalculate the hash index for the lon of every entry in the PIDLAM. If necessary it will reposition a PIDLAM entry since the PS-algol abstract machine uses a compacting garbage collector allowing lons to be changed by garbage collections. Hence when the PIDLAM must be indexed by hashing on lons it is necessary to use this routine to ensure that hashing using a lon will work. This need only be done just before a commit and after any garbage collections that occur during a commit.

The second update routine is used to insert new entries into the PIDLAM. An important feature of the CPOMS is its ability to restore its data structures if a database operation such as an open or commit should fail. If this can be accomplished successfully a user's program can be allowed to continue after such an error. To support this all entries in the PIDLAM are tagged as temporary or permanent.

When a new PIDLAM entry is inserted it is tagged as temporary using the dummy trademark field of its PS-algol structure. Once a database operation has been judged as successful a third update routine is used to set all the temporary markers to permanent markers. However if a database operation should fail then the fourth and final update routine can be used to remove all PIDLAM entries marked as temporary. This update operator is used because the new PIDLAM entries refer to objects that will not be accessible to the user's program. Therefore the only pointers to them are in the PIDLAM. So if their PIDLAM entries are removed they can be removed from the interpreter's heap by the garbage collector.

Not all PIDLAM entries are inserted during explicit database operations such as commit. During normal execution of a program, objects will be imported as they are accessed by the user's program and possibly modified. Therefore these objects and their PIDLAM entries should be kept since they may be required for a future commit. However all new PIDLAM entries are marked as temporary so it is necessary to explicitly change the markers corresponding to these objects. To do this all the temporary markers in the PIDLAM are set to permanent markers before a database operation is performed. If the database operation then fails only the PIDLAM entries created for it will be removed. This use of the temporary and permanent markers allows a reliable means of automatically restoring the PIDLAM after a database error.

3 External Addressing

The addressing mechanisms used by the CPOMS can be divided into two kinds, internal and external. Internal addressing is used when a pid is encountered by the interpreter. It relies on the PIDLAM which is duplicated for every running PS-algol program. External addressing is used to address the persistent store via a pid or a database name. It is supported by data structures that are shared by every PS-algol program. The purpose of this chapter is to describe these shared data structures.

3.1 The UNIX file system

The CPOMS uses the UNIX file system as the basis for its implementation of the persistent store. To ensure the CPOMS always knows where to find the persistent store all the files used are kept in a single UNIX directory that has a predefined name. This predefined name is '/usr/lib/psalgol/dbs' but an alternative can be specified by defining a shell variable, i.e. logical name, called 'PSDIR'. The CPOMS will use an alternate UNIX directory if the shell variable is defined and the directory exists. A new directory should be empty since the CPOMS will automatically create the files it requires.

3.2 Databases

Within the persistent store PS-algol heap objects are grouped into larger objects called databases. These databases are implemented as pairs of UNIX files by the CPOMS. One of these files, called the data file, holds all the PS-algol heap objects contained in the database. Databases do not contain any other type of objects. The second file, called the index file, is used to contain the addresses of the objects in the data file. It is this address that is held in the PIDLAM entry for an object. The reason for having two files is to support object level addressing on a database. Using a separate index file allows objects to be found easily and also allows a straight forward compaction algorithm to be used by a disk garbage collector. The naming conventions for these files is given in appendix 1.

3.3 The structure of a pid

The CPOMS external addressing must be able to map a pid to a position in the index file of the appropriate database. Therefore a pid should be structured in such a way as to readily identify which database and which object within that database it refers to. A simple solution to this would be to divide a pid into a database number and an object number within that database. However this solution in common with all other types of segmentation is plagued by the problem of how big to make the segments. Either a database will be too small or a database will be too big. The solution adopted by the CPOMS to partly alleviate the problem is to use a relatively small segment size and allow arbitrary collections of segments to form a larger super segment. This is slightly different from most segmentation schemes in that the collection of small segments need not be contiguous. For the purposes of this document we will refer to the CPOMS segments as partitions and the super segments as databases.

The particular structure of a pid chosen by the CPOMS is as follows:

```
 ------------------------------------------
|  1 | Partition Number |   Object Number   |
 ------------------------------------------
           15 bits             16 bits
```

Figure 3.1 The structure of a pid

This particular choice of partition size allows for up to 32768 partitions each of which contains 65536 addresses. Partition number 0 is reserved for special objects used by the CPOMS and the PS-algol abstract machine, see appendix 2. These special objects include the one character strings, the standard identifiers and the structure trademarks declared in the PS-algol initialisation program. Partition 0 is never allocated to a database.

To avoid any ambiguity in addressing, a partition may only be allocated to one database at a time. Furthermore a database must always have at least one partition allocated to it. Hence the 32767 available partitions allow for 32767 databases with one partition each, one database with all 32767 available partitions or any distribution in between.

3.4 The partitions file
A shared data structure called the partitions file is used to record the database that each partition is allocated to. The structure of the partitions file is as follows:

```
    ---------------------
...|  Database number  |...
    ---------------------
           2 bytes
```

Figure 3.2 A partitions file entry

Each partition has an entry in this data structure containing the database number of the database it is allocated to. Every database is given a unique number by which it can be identified. How this database number is allocated will be described in section 3.6. If the partition has not been allocated then its entry contains -1. The first entry in the partitions file is for partition number 1, the second entry for partition number 2 and so on. No entry is required for partition number 0 since it is reserved for use by the CPOMS. To save space the partitions file is truncated so that it only contains entries for the partitions in the range 1 to the highest allocated.

The purpose of the partitions file is to control the allocation and deallocation of partitions to databases. As such it does not play any part in the normal external addressing mechanism. A separate data structure, described in chapter 4, is provided to perform the mapping of partitions to databases.

3.5 Pids, partitions and index files
When a partition is allocated to a database the partition is given its own section of the database's index file. This section of index file is contiguous and will contain entries for each of the partition's 65536 object numbers. A new partition is only allocated to a database when all the previously allocated partitions are full. Therefore a new partition is allocated space at the end of the index file and its starting position will be a multiple of 65536 entries. This allocation scheme enables the starting positions of partitions to be calculated from the order in which they were allocated to a database.

If the starting position of a database's partitions are known the database object number of any pids referring to the database can be calculated. This is done by simply adding the pid's object number to the starting position of its partition to yield its database object number.

To support this calculation the CPOMS records the starting position of every partition in the database to which it is allocated to. A PS-algol vector of integers is used to hold the information. For each partition allocated, an entry is placed on the end of the vector. Therefore the order of the entries is the same as the order of the partitions in the index file. A partitions section of the index file can then be found be multiplying its position in this vector by 65536 entries.

3.6 The Directory file
To allow the CPOMS to map database names to databases, a data structure called the database directory is used. Note this is a CPOMS data structure and should not be confused with the UNIX directory that holds the files used by the CPOMS. The format of the database directory is as follows:

```
-------------------------------------------------------
| root      |  locks  | owner  | database | database |
| partition |         |        |          |          |
| number    |  held   | number | name     | password |
-------------------------------------------------------
  2 bytes     2 bytes   2 bytes  14 bytes   12 bytes
```

Figure 3.3 A directory file entry

The database directory contains an entry for each database that has been created. Databases are implicitly numbered by the position of their entry in the directory. The first entry is numbered 0, the second entry 1 and so on. Once an entry has been made it never changes position. Hence every database has a unique number that can be used to identify it.

Each entry in the database directory contains five pieces of information.

a) The first partition allocated to the database.
This partition always has as its first object the root object of the database. Hence this partition number can be used to construct the pid of the root object. In addition this partition was allocated the first section of the database's index file. Therefore the full address mapping for the pid of the root object can be constructed.

b) A record of any locks held on the database.
The locking protocol used by the CPOMS is one writer or many readers. The locks held on each database are recorded as:
0 No locks are held.
-1 The database has been locked in write mode.
n The database has been locked in read mode by 'n' incarnations of the CPOMS.

c) The UNIX user identification number of its creator.
To control the renaming of databases and the changing of their passwords it is necessary to know who owns a database. The UNIX user identification number performs this task.

A privileged user facility is also supported that is similar to the UNIX super user. This user is identified as the owner of the PS-algol interpreter. To ensure security the PS-algol interpreter has the set user identification privilege set and all the database files are made only accessible to their real owner, the PS-algol user. The databases can then only be accessed via PS-algol programs.

d) A fixed length name.
This name is used to identify a database to human users of PS-algol. To simplify the first implementation of the CPOMS this name has a maximum length of 14 characters not including the null character, ascii 0.

e) A fixed length password.
This password is used to validate access to the database. It is only used if the access to the database is made using the 'open.database' standard function by someone other than its owner. The password is restricted to a maximum of 12 characters not including the null character.

4 Internal support for external addressing

The external addressing described in the previous chapter maps a pid to a database and database object number. First it must map the partition number of a pid to a database. It must then calculate the object number in the database from the partition number and the object number within the pid, see section 3.5. Once this has been done the database's index file must be read to find the address of the desired object in the database's data file.

To support this sequence of steps the CPOMS uses two data structures. The first of these is a linked list of entries for each open database. It provides access to the files of the open databases. The second data structure called the PTODI, partition to database and index map, is used to record the allocation of partitions to databases and the index file starting positions for these partitions. Unlike the other data structures involved in external addressing these two must be **duplicated** for every PS-algol program since they are different for each program. These data structures will now be described.

4.1 Recording opened databases

When a database is opened, see chapter 6, the CPOMS creates a data structure to hold details of the opened database. This data structure forms part of a linked list of entries for each open database. The purpose of the data structure is to allow access to the database's files as well as to record details of the state of the database. The structure of a linked list entry is as follows:

```
               Open database: entry fields 0 -> 5
  ----------------------------------------------------------
  | heap   | dummy | root   | copy | next     | data       |
  |        | trade |        | of   |          | file       |
  | header | mark  | object | root | database | descriptor |
  ----------------------------------------------------------
                     All fields are 4 bytes

               Open database: entry fields 6 -> 10
   -------------------------------------------------------
   | index      | database | type | next     | root      |
   | file       |          | of   | freelist | partition |
   | descriptor | number   | lock | entry    | number    |
   -------------------------------------------------------
                     All fields are 4 bytes
```

Figure 4.1 An open database data structure

a) Heap header and dummy trademark.
The heap header is that of a PS-algol structure. The trademark field is included to make the data structure look like a real PS-algol structure. It is probably unnecessary.

b) A pointer to the root object of the database.
The root object performs several house keeping functions required by commit. It is also required if the database is opened a second time by the same program. If this happens the root object is indexed to find the result for 'open.database'.

c) A pointer to a copy of the root object.
During a commit the CPOMS maintains a copy of the root object to speed up the automatic recovery mechanism it uses, see chapter 9.

d) The next database.
This is the pointer to the next data structure in the linked list.

e) Data and index file descriptors.
These are pointers to data structures describing the state of the I/O to the database's files. Their function is similar to that of ordinary PS-algol file descriptors and is described in the next section.

f) Database number.
This is the number given to the database by the database directory. It is used as the key when searching the linked list for a database's entry.

g) Type of lock.
This is set to 0 if the database is open for reading and 2 if the database if open for writing. This information is used when a database is opened for a second time by the same program and also by commit when it is identifying participating databases.

h) Next free entry in the index file.
This entry is only used during commit, see section 8.2.3.

i) Root partition number.
This is included to control an experimental version of open database. It is not used by the current version of the CPOMS.

4.2 Database file descriptors
The file descriptors mentioned above are used to access a database's files.
The structure of these file descriptors is as follows:

```
---------------------------------------------------------------------
| heap   | dummy  | UNIX       | file  | buffer  | buffer |          |
|        | trade  | file       |       |         |        | buffer   |
| header | mark   | descriptor | name  | address | flags  |          |
---------------------------------------------------------------------
 4 bytes  4 bytes    4 bytes   12 bytes 4 bytes  4 bytes  128 bytes
```

Figure 4.2 A database file descriptor

a) Heap header and dummy trademark.
The heap header is that of a PS-algol structure. The trademark field is included to make the data structure look like a real PS-algol structure.

b) The UNIX file descriptor for the disk file.
This is usually an integer in the range 0 to 19. It is used to perform the actual I/O operations on the file.

c) File name.
The UNIX operating system only allows a maximum of 20 files to be open at once. Therefore to overcome this a scheme of opening and closing files is required. To support this the filename is saved in the file descriptor. However this scheme has not yet implemented so the maximum number of open databases is 5.

d) Buffer, buffer address and buffer flags.
To improve access performance to the database files a buffering mechanism can be implemented. This is supported by providing a buffer and status flags to indicate changes to the buffer. The buffer address is also provided to indicate which part of the file the buffer represents. Currently only the index files use the buffers.

4.3 The Partition to Database and Index Map

The PTODI is provided to map a partition to its database and the starting position in the database's index file. The PTODI only contains entries for the partitions that its incarnation of the CPOMS
can access. Therefore it is duplicated for every PS-algol program. The format of the PTODI is as follows:

```
 PS-algol vector header          A PTODI entry
-----------------------   ---------------------------------
|     |     |         |     |          |        |       |
| VEC |  1  |  3 * n  | ... | part num | db num | index | ...
|     |     |         |     |          |        |       |
-----------------------   ---------------------------------
```

Figure 4.3 The structure of the PTODI

The PTODI is held as a PS-algol vector of integers with lower bound 1 and upper bound 3 times the number of entries. A dummy entry is made for partition 0 to initialise the vector. Each entry is a triple consisting of 3 things:

a) A partition number.
This is the key field that is used while searching the PTODI.

b) A database number.
This is the database number of the database that the partition was allocated to.

c) An index file offset.
The starting position of the partition's section of the database's index file.

Access to the PTODI is provided by three routines using a pid as parameter. One of these routines retrieves the database number for the pid's partition. Another one retrieves the index file offset and the last one returns the result of adding the index file offset to the pid's object number.

The PTODI is ordered by partition number, the partitions being represented as pids with object number 0. This permits a binary split search algorithm to be used.

Additions to the PTODI are done one at a time by creating a new PS-algol vector and copying. This applicative update provides a very simple recovery mechanism. To restore a previous version of the PTODI the CPOMS need only restore a pointer to the original version. Clearly this update technique will become very inefficient if a large number of new partitions had to be added.

Entries are never removed from the PTODI. The only cases where this may be necessary are circumvented by restoring a pointer to a previous version of the PTODI. This would only happen if a database operation, such as 'open.database' or a commit, should fail, see chapters 6,7 and 9.

5 Summary of external addressing

The preceding chapters describe the data structures used to support external addressing. The interaction of these data structures will now be described.

When a pid is encountered the PIDLAM is searched. If the pid is contained in the PIDLAM it will have a lon associated with it. This lon is used to overwrite the location that held the pid. However if the pid is not present in the PIDLAM the external addressing mechanism must be used to find the object pointed to by the pid. The steps involved in this are as follows:

a) Divide the pid.
The first step is to separate the pid into its two components, the partition number and the object number within the partition.

b) Find the database.
The PTODI is then searched using the pid's partition number as the key. This will find the partition's entry from which its database number is extracted. The list of open databases is then searched using the database number as the key. The open database entry provides the CPOMS with access to the files for the database holding the object pointed to by the pid.

c) Calculate the database object number.
The database object number is then calculated. First the index file offset is extracted from the partition's entry in the PTODI. The offset is then added to the object number within the pid to yield the database object number.

d) Find the object's address in the data file.
The database object number is used to index the index file. Object number 0 is the first entry in the index file, object 1 is the second and so on. The resulting index file entry is the byte address in the data file of the object pointed to by the pid.

e) Copy the object into the heap.
The object is then copied from its position in the data file to free space in the interpreter's heap. A pointer to this copy of the object is the lon of the object.

f) Update the PIDLAM.
The pid and the new lon are entered into the PIDLAM. Any future use of the pid will then result in the lon being found in the PIDLAM so avoiding the use of external addressing.

g) Apply the pid translation rules.
Once the PIDLAM has been updated the pid translation rules of chapter 2 are recursively applied. The first of these is to overwrite the location holding the pid with the new lon. The type of the object would then decide any further action.

6 Opening a database

When a PS-algol program starts it has no means of directly accessing the persistent store, it only knows the names of the databases. Therefore before the external addressing mechanism can be used the address mappings for the objects to be accessed must be constructed. This is done by opening the databases containing the objects. This chapter describes how the open is performed and how the address mappings are constructed.

6.1 Addressing using a database name

The first step in opening a database is to identify the database from its name. A request to open a database would be supplied with a database name, a password and an indication of the use of the database, "read" or "write". Using the database name as a key the database directory is searched. If there is no directory entry with the supplied database name an 'error.record' will be the result of the open.

Once the database directory entry has been found the password will be validated and the requested lock checked. If the database is owned by the user requesting the open then the password will not be checked. The check on the locking is to ensure that granting the lock to this incarnation of the CPOMS will not violate the protocol of one writer or many readers. In the case where the database is being opened for a second time the original lock is ignored for the purposes of this check. Therefore an attempt to read lock a database that is already open will always succeed. However an attempt to write lock a database that is already open will only succeed if no other incarnation of the CPOMS has the database open for reading. If either check fails a suitable message is returned via an 'error.record'.

If the checks on the password and lock are successful an entry is made in the CPOMS list of open databases. This entry is given file descriptors for the database's data and index files. These files are opened by calculating their names from the database number, see appendix 1. The file names and UNIX file descriptors allocated by the open are entered into the database file descriptors.

In addition to the open database entry the external addressing mechanism requires a corresponding entry in the PTODI. This entry is constructed from the root partition number held in the database directory. This is straight forward since the root partition starts at position 0 in the index file, and the database number is known. When this entry has been made in the PTODI the CPOMS has the necessary address mappings to address any object allocated to the root partition of the database.

6.2 The database root object

The database directory contains the address mapping for the root partition of a database. Any other partitions allocated to a database have their starting positions recorded within the database itself. Therefore the way in which such information is kept must be described.

The first object in every database has the job of recording the status of the database. This information includes the correct length of the data and index files, what other databases may be reached from this one and what partitions are allocated to the database. The structure of the root object is as follows:

```
-----------------------------------------------------------------------------
| heap   | trade | pntr | linked | parts  | partition | index | index | data |
|        |       | to   | data   |        | offset    | free  | file  | file |
| header | mark  | root | bases  | vector | for parts | list  | size  | size |
|        |       |      |        |        | vector    |       |       |      |
-----------------------------------------------------------------------------
```

Figure 6.1 A database root object

The root object is a PS-algol structure with the following structure class.

structure opdb.result(**pntr** root.of.db)

Only the pointer to the root object is mentioned in the structure class so that all the other fields are inaccessible to PS-algol programs. The root object pointed to by the visible field of the structure is the result returned by the 'open.database' standard function. It is usually a table package but this is irrelevant to the CPOMS.

A PS-algol vector of integers is used to record, using database numbers, which databases are reachable from this one. A database is reachable in this context if any of its objects are pointed to by an object in the database being opened. The lower bound of the vector is 1 and the upper bound is the number of reachable databases. If there are no reachable databases then the pointer value for **nil** is substituted for the vector.

A PS-algol vector of integers is also used to record which partitions have been allocated to this database. The lower bound of the vector is 0 and the upper bound is 1 less than the number of allocated partitions. Each entry is represented by the pid for object 0 of the partition. The entries are also in the order in which they were allocated. Therefore the position of an entry in the vector implicitly gives the index file starting position for the partition, see section 3.5.

When a database is first opened only the address mapping for the database's root partition is known. To construct the other address mappings the vector of partitions must be read. However the address mapping for the partition holding the vector may not be known. Therefore to allow the address mapping to be calculated the starting position for this partition is recorded.

The index file free list entry is the number of the first free entry in the index file. This value is used to initialise the value in the open database entry. It is only used by commit.

The database file sizes are included for use by commit so that it knows where the end of the files should be. If a commit fails while writing new objects to a database the files may be longer than they should be. Therefore the next commit must be explicitly told where the end of the files should be.

6.3 How an open uses the root object

Now that the database root object has been described the remaining steps of opening a database can be described. Once the steps described in section 6.1 have been completed the remaining address mappings for the database must be constructed. First of all the database's root object is accessed using the external addressing mechanism. The address mapping is then constructed for the vector holding the list of partitions. This is possible because the database number is already known, the partition number can be found by extracting it from the vector's pid and the index file offset is held in the root object. If the partition vector is in the database's root partition this step can be ommitted. The vector is then accessed using the external addressing mechanism.

Once the vector of partitions has been accessed a PTODI entry is made for each of its entries. The index file offset used is calculated from a partition's position in the vector using multiplication, see section 3.5. No entry will be added for the root partition or the partition holding the vector of partition numbers. These entries would already have been made.

When all the address mappings for a database have been created the CPOMS returns a pointer to the database's root object as the result of the open. The pointer to the root object is used by PS-algol procedures to implement the high level user interface. In the case of the 'open.database' standard function the table package pointed to by the root object is returned as if it were the database's root object. This extra level of interface in PS-algol allows the CPOMS to be independent of the high level user interface.

6.4 Recursive opening of databases

When a database has been opened the CPOMS has access to the address mappings for all the objects within the database. However the database may contain pointers to objects in unopened databases, i.e. the CPOMS may encounter pids for which it knows of no valid address mappings. There are two alternative approaches to the problem, raise an error or open every reachable database.

The simplest solution would be to indicate an error if such a pointer was ever dereferenced. However this is unsatisfactory for two reasons. Firstly, PS-algol contains no means of identifying where an object is stored. The databases only provide knowledge of the access path to an object. Therefore it is not possible for a PS-algol program to determine which database an object is in and so it cannot anticipate the problem. The second problem with raising an error is that PS-algol contains no exception handling mechanism that could deal with it. As a result such an error would cause a program to crash. The combination of these two problems means that it would not always be possible to anticipate if a program might crash, particularly if the programmer was not responsible for the databases the program used.

To avoid these problems with raising errors the CPOMS adopted the second solution, it opens every reachable database. This operation is performed just before the result of the open is returned. When all the address mappings for a database have been constructed the vector of reachable databases is accessed. Then each database contained in the vector has open recursively applied to it.

The main difference between the original open and the recursive ones is that the database number is supplied in place of the database name and password. The choice of lock for the subsequent opens is usually a read lock. However some databases may have had their names removed from the database directory but may still be reachable. These databases are those that have been be removed, however databases are only deleted when they no longer contain reachable objects. If any of these removed databases are opened the lock applied to them will be the same as that supplied to the original open. The reason for this is that every other database can be explicitly opened for writing by a user, an unnamed database cannot.

If any of the recursive applications of an open should fail then all the opens are undone and an 'error.record' returned as the result of the original open.

6.5 Automatic recovery used by open

If an open should fail the CPOMS will attempt to recover from the error causing the failure. Once this has been completed an 'error.record' is returned as the result of the open to indicate the cause of the error. This allows the calling program to continue and react to the failure. However if the failure cannot be recovered from then the calling program will be terminated with a suitable error message. Automatic recovery is also used by create and commit, see chapters 7 and 9.

The recovery mechanism used by the CPOMS restores all the data structures modified by the open. The following data structures are modified:

a) The pid to local address map, PIDLAM.
Entries will have been made for database root objects, vectors of partition numbers and vectors of reachable databases. Before an open is performed all the temporary markers in the PIDLAM are reset. Therefore entries made during the open will have temporary markers on them and can be easily removed to restore the PIDLAM.

b) The partition to database and index map, PTODI.
This map is modified each time a database is opened. It can be restored by keeping a pointer to the copy in use before the open. Restoring this pointer is sufficient to restore the PTODI because all modifications to the PTODI are done by copying.

c) The list of open databases.
As described above an entry is made in this list for each open database. However if a database is already open this is not done. Instead if the lock on the database is changed from a read lock to a write lock the existing entry will be modified. The modification takes the form of recording a write lock but negating the value used to record it. Therefore at the end of an open the databases with read locks that have changed to write locks can be identified. If an open is successful the negated values are made positive. However if an open fails the write locks are changed back to read locks and the lock fields reset to the value used for read locks.

Each new list entry represents a newly opened database. Therefore the new entries can only be removed once their databases have been unlocked and the databases' files closed.

All new list entries are placed at the front of the list. Therefore if the old start of the list is recorded the old and new entries are easily identified and the required action can be taken. The new entries are discarded by restoring the pointer to the old start of the list.

d) The database directory.
The only modifications made to the database directory are the recording of new database locks. In fact a copy of the database directory is updated, see appendix 1. Therefore if an open fails all the new locks can be discarded by simply discarding the copy. If an open does succeed the copy replaces the database directory, see appendix 1.

When the automatic recovery is complete the open will return an 'error.record' as its result indicating the cause of the failure. The PS-algol program calling the open can then continue normally.

7 Creating a database

Before any new part of the persistent store can be used a set of address mappings must be initialised for it. This is done by allocating a free partition to a database in one of two ways. Either by enlarging a database during a commit or by explicitly creating a new database. The purpose of this chapter is to describe how a new database is created and how its address mappings are initialised. The case of commit is described in the next chapter.

7.1 Initialising the address mappings

A request to create a new database is supplied with a database name and a password. The first step in the creation is to search the database directory for a free entry. While this is being done a check is made to see if the supplied name is already in use. If it is then an 'error.record' will be returned as the result of the create. Once a free directory entry has been found the database name and password are entered in it. At this stage the new database has its own number, its position in the directory, a unique name and a password.

The next step is to allocate a free partition to the database. This is done by searching the partitions file for an unallocated entry. When one is found it is marked as allocated by placing the new database's number in the partitions file. The partition's number is then entered into the directory entry as the database's root partition. If a free partition cannot be found the database directory entry will be removed and an 'error.record' returned as the result of the create.

7.2 Creating the database's files

As described in the previous chapter every database has a root object that records the state of the database. Therefore when a database is created the data and index files must be initialised to contain such a root object.

When a new root object is created its fields are given the following initial values.

a) The trademark field.
The trademark field is initialised with the trademark for the structure class 'opdb.result'. This trademark is given a reserved pid when by the CPOMS so it is this pid that is used to initialise the field.

b) The root object.
The pointer to the user interface's root object is initialised with the reserved pid allocated to **nil**. This value will be overwritten with a pointer to a table package by the high level user interface.

c) The vector of reachable databases.
Initially there are no reachable databases. Therefore this field is initialised with the pid for **nil**.

d) The partitions vector.
The partitions vector field is initialised with the pid for object 1 of the root partition. This is because a partitions vector will be created immediately after the root object. It is object 1 since the objects are numbered from 0. The vector is initialised with upper and lower bounds of 0 with one entry that is the pid for object 0 of the root partition.

e) The partitions vector offset.
This is initialised to 0 because the partitions vector is in the database's root partition.

f) The index file free list.
This field is initialised with 0 to indicate there are no free entries in the index file.

g) The index file size.
Initially the index file will have two entries, one for the root object and one for the partitions vector. Therefore the index file will be 8 bytes long and this field will be initialised to 8.

h) The data file size.
When the data file has been initialised it will contain the root object and the partitions vector. Therefore its initial size and the value for this field will be 52 bytes, 13 words of 4 bytes each.

It follows from the initialisation of the data file that the index file will be initialised with two entries. The first for the root object will be 0, the root object is the first object in the data file. The second will be 36, the partitions vector comes immediately after the root object which has a length of 36 bytes.

7.3 Opening the new database
When the preceding steps have been completed the actual creation of the new database is complete. However the high level user interface requires a database to have more structure than the CPOMS provides. Therefore the final step of the create is to open the database for updating. The result of this open is a pointer to the database's root object.

The high level user interface uses the result of the open to add the additional logical structure it requires. It does this by creating a new table package and assigning it to the root object field of the database's root object. Only when this has been done does the high level user interface consider the database to be created.

In fact the new table will only be added to the database if a commit is successfully performed. For this reason the high level user interface provided to user programs always checks for the presence of a table package whenever a database is opened.

7.4 Automatic recovery used by create
If a create should fail the CPOMS will attempt to recover and report the error to the calling PS-algol program. The automatic recovery used by create works as follows. In same way as open, the database directory is copied and only updated when the new database has been successfully opened. Therefore unless the new database is created and opened successfully the database will not appear to exist.

If an error occurs during the actual creation phase of create the new data and index files will be deleted. If the creation phase is successful an open is performed. The automatic recovery used for errors that occur during the open is described in section 6.5.

If a create fails at any point an 'error.record' will be returned as the result of the create. The calling PS-algol program can then continue normally.

8 Committing changes to the persistent store

The CPOMS allows a PS-algol program to access data objects in the persistent store by copying them into the program's heap. Therefore to allow a program to modify these objects a mechanism is required to copy them back to the persistent store. The purpose of this chapter is describe how the copying is done by the CPOMS. In addition to being able to modify the persistent store the CPOMS provides transactions by ensuring that any set of modifications form one atomic action. That is the modifications are either completely successful or have no effect. The next chapter describes the recovery mechanism used by commit to implement this.

8.1 Alternative commit algorithms

The first problem to be addressed by commit is the choice of what data objects should be copied to the persistent store. The choice of objects to be committed can be restricted to two groups. First of all only changed objects need to be included. Any object that has not been modified will be the same in a program's heap as it is in the persistent store.

Lons would be out of context in the persistent store so the only pointers allowed are pids. Therefore every object pointed to by an object in the persistent store must have a pid. It follows from this that an object imported from the persistent store can only point to an object without a pid if it has been updated. Hence the second group of objects are those that do not have a pid but are reachable from the objects in the first group.

It is not always desirable to commit every object in the two groups. For example a database opened for reading may contain data that should not be changed, so objects from such a database might be ignored. Alternatively changing objects from a database opened for reading might be considered an error preventing any objects being committed. Another restriction that may be desired is to exclude objects that could never be reached from the root of a database. After the program calling the commit finished these objects could never be accessed.

Giving varying degrees of consideration to these problems the following possible choices of objects were investigated.

a) Allow only one database to be opened for updating. Only commit changed objects reachable from the root of this database ignoring any objects in other databases.

b) Commit every object reachable from the root of a database open for updating. Ignore any objects in databases only open for reading.

c) Commit every object reachable from the root of a database open for updating. If any changed objects belong to a database only open for reading do not perform the commit.

d) Commit every object reachable from any changed object in the persistent store. This choice of objects is precisely the two groups mentioned above. If any changed objects belong to a database only open for reading do not perform the commit.

As mentioned in chapter 6 it is not always possible for a PS-algol program to know which database an object is actually in. Therefore any commit algorithm that involves ignoring changed objects in read only databases will prevent a programmer predicting what his program will commit. If a program is attempting to enforce some high level consistency between disjoint pieces of data this would invalidate the program's results. Hence this highly undesirable feature led to options a and b being discarded.

A consequence of the copying method of accessing the persistent store is that the CPOMS is unable to detect every path from a database root to an imported object. Initially every imported object will be reachable from a root. However in the course of executing a PS-algol program the imported path from an object to the root of a database may be broken. In such a case the CPOMS may not have enough information to decide whether or not the object is still reachable. This limitation of the CPOMS implementation led to option c being discarded.

As a result of the above problems the CPOMS implements option d. How this was done is now described.

8.2 The commit algorithm used by the CPOMS

Every pid that has been dereferenced by a PS-algol program will have an entry in the PIDLAM. Also every object created by the PS-algol interpreter or changed by it is marked as a changed object. Therefore during a commit the CPOMS can find all the changed objects in the persistent store by consulting the PIDLAM. However the algorithm chosen for commit requires a check on whether any of the changed objects belongs to a read only database.

To support the check on where objects belong, the CPOMS maintains a list of which partitions belong to databases opened for updating. This list of writeable partitions is held as a PS-algol vector of integers. Each partition is represented by the pid of its first object, object 0. When a database is opened for updating, its partitions vector is merged with the vector of writeable partitions. The merging is done to create a new vector ordered by partition number. In common with the other CPOMS data structures the applicative update operation provides a very simple recovery mechanism. That is an older version of the vector can be restored simply by retaining a pointer to it. The ordering of the vector enables a binary search algorithm to be applied to it. However the current CPOMS uses a linear scan.

The commit algorithm is divided into the following 6 steps.

a) Find all the open databases.
b) Check all the imported objects.
c) Allocate pids to new objects.
d) Copy objects back to the persistent store.
e) Reset the change markers.
f) Indicate success or failure to the calling program.

8.2.1 Finding all the open databases

The list of open databases is consulted and the root objects of the databases open for updating are marked as changed. This is because the commit may need to modify them while allocating new pids, see section 8.2.3. If no databases are open at all or no databases are open for updating the commit will terminate and return an 'error.record' as its result.

8.2.2 Check all the imported objects

The PIDLAM is scanned looking for all the imported objects that have been changed. If an object has been changed the partition number of its pid is checked against the vector of writeable partitions. If the object's pid does not belong to a writeable partition the commit will be aborted and an 'error.record' returned as its result.

8.2.3 Allocate pids to new objects

Every changed object in the PIDLAM has a pid allocated to those objects reachable from it. This is done by recursively applying a pid allocation routine to each changed object in the PIDLAM. The recursion stops when the routine encounters an object with a pid.

Each application of the allocation routine is supplied with the pid of the object pointing to the one being inspected. Two alternatives are then possible. Either the current object has a pid or it does not.

If an object without a pid is encountered a new pid must be allocated to it and the allocation recorded in the PIDLAM. As an attempt to keep lists of objects close together in the address space, i.e. in the same database, the new pid will be allocated from the database holding the parent object. The actual allocation of the pid is done in two steps. First of all a free entry is found in the index file of the parent object's database. This is done by first consulting the index file's free list. The free list is held as a list of forward pointers in the index file. Each entry is the negated position of the next entry except for the last one which is 0. The start of the free list is held in the database root object and in the database's entry in the list of open databases. If there is no free list the index file is extended by one entry.

When an index file entry is found it has the current length of the data file placed in it. This will be where the object receiving the pid will be stored. The data file length is then increased by the length of the object that was given the pid. Therefore the next object will be placed directly after it in the data file.

The next step in allocating a pid is to calculate which pid the index file entry represents. This is done by calculating which of the database's partitions the entry is in, i.e. which set of 65536 entries. The database's partitions vector is then indexed to find the partition number for the new pid. The new pid is then completed by using the entry's position in the partition as the pid's object number.

However it may be that when a new index file entry is added it will not be part of any of the database's allocated partitions. If this should happen then the CPOMS must allocate a new partition to the database. A new partition is allocated to a database in 4 steps.

a) A free partition is found.
A free partition is found by searching the partitions file. When a partition is found its entry in the partitions file is marked with the database's number. If there are no free partitions then the commit will be aborted and it will return an 'error.record' as its result.

b) The list of writeable partitions is modified.
The newly allocated partition belongs to a database open for updating. Therefore the partition number must be recorded in the list of writeable partitions.

c) The partitions vector is modified.
A copy of the database's partitions vector is then made to record the allocation of the partition within the database. A copy must be made because a vector is a fixed size object.

d) Update the database's root object.
Since the copied vector is a new object the root object must be changed to point to it. Also the new vector must be given a pid. The allocation of a pid to the vector requires a recursive call to the routine generating new pids. The next pid to be allocated will be object 1 in the new partition so this will always succeed. Therefore the pid of a partitions vector is always object 1 in the last partition allocated to its database. The last update necessary to the root object is to record the index file starting position for the new partition. This step is done for the benefit of open, see chapter 6.

When a new pid has been allocated to an object the pid allocation routine is called for each pointer field in the object.

If an object is encountered with a pid, a check is made to see if the object and its parent object are in the same database. If they are not then the child's database is reachable from the parent's and this must be recorded. This record is kept in the vector of reachable databases held in the parent database. When an inter database pointer is found this vector is searched to see if it contains the child's database. If the child's database is not present it must be added.

Modifying the reachable databases vector is similar to modifying the partitions vector. A new copy must be made that includes the new entry. In this case the new vector is ordered by database number. The database root must then be made to point to the new vector and the vector given a pid.

Once the new reachable databases vector has been made the pid allocation routine returns to its caller terminating the recursion.

8.2.4 Copying objects to the persistent store

When the allocation of pids is complete the PIDLAM will contain an entry for every changed object with a pid that must participate in the commit. Therefore the next step is to copy the changed objects to the persistent store.

During the copying of objects it is necessary to convert all their pointer fields to pids. For this reason it is necessary to be able to index the PIDLAM by hashing on lons. Therefore a commit will require the PIDLAM to be checked before performing this step and after each garbage collection. This is done by setting a flag to tell the PS-algol interpreter to call the checking routine after each garbage collection.

The external addressing used by commit is slightly simpler than the description in chapter 5. This is because when each PIDLAM entry is made the index file entry for an object is recorded. Therefore the part of the addressing involving the index files can be performed without the overhead of disk accesses.

8.2.5 Resetting the change markers

When all the changed objects in the PIDLAM have been copied to the persistent store their changed markers can be reset. If a commit is invoked at a later date they will only participate if they are modified between the two commits. Any objects with changed markers that are not in the PIDLAM will not be affected. They will not have a copy in the persistent store so they must be able to participate in a future commit even if they are not modified again.

8.2.6 The result of commit

If a commit is successfully completed the pointer value **nil** is returned. However should a commit fail an 'error.record' is returned indicating the error. If a commit does fail the recovery mechanisms described in the next chapter are performed. The calling program can then continue and react to the error.

9 Automatic recovery used by commit

Commit updates the persistent store as an atomic action. That is, it is totally successful or it appears never to have happened. The automatic recovery mechanism used by commit must therefore be able to undo the changes made to the persistent store if the commit should fail for any reason. If the error causing a commit to fail is not too serious the recovery mechanism must allow the calling program to continue. That is, it should also be able to restore the internal CPOMS data structures modified by the commit.

9.1 Before look files

The information requirements of the recovery mechanism will now be described by considering how each step of the commit algorithm can meet them. The 6 steps of the commit algorithm previously described are:

a) Find all the open databases.
b) Check all the imported objects.
c) Allocate pids to new objects.
d) Copy objects back to the persistent store.
e) Reset the change markers.
f) Indicate success or failure to the calling program.

9.1.1 Finding the participating databases

A commit may modify several databases at once. Therefore a mechanism must be provided to associate the participating databases with a commit. In addition it must be possible to destroy the record of all the changes made in one atomic action. This requirement is necessary to ensure that parts of a commit are not accidently undone so invalidating a commit's atomicity. The mechanism chosen by the CPOMS to meet these two requirements is called a before look file.

The second requirement is met by having only one before look file per commit. Therefore its information content can be destroyed by the single UNIX system call, 'creat'. The effect of 'creat' is to truncate an existing file to 0 length. It is performed as an atomic action. To meet the first requirement the before look file is given a UNIX filename, in the UNIX directory holding the databases, for each of the participating databases, see appendix 1. If a commit should fail there will appear to be a before look file for each participating database. The presence of a before look file identifies a database as needing restored. To complement the naming scheme the before look file contains the numbers of the participating databases. Therefore if a before look file is found it can be used to identify all the databases that must be restored.

The creation of the before look file is performed as part of the first step of a commit. When the first participating database is found the before look file is created and associated with the database, see appendix 1. A header is then written to the before look file containing three fields and the number of the first participating database. The format of the header is as follows:

```
---------------------------------------------
| number    | status | file  |   | database |
| of        | of     |       |...|          |
| databases | commit | size  |   | number   |
---------------------------------------------
   4 bytes     4 bytes  4 bytes     4 bytes
```

Figure 9.1 The before look file header

A status value field is included to indicate the progress of the commit. At each point in a commit only certain parts of the persistent store will have been modified. Therefore it is useful to know how much must be restored if an error occurs. When the header has just been written the status value is set at 0. The use of the file size field will be described later.

Once the header has been written the numbers of the other participating databases are appended to the before look file. When each additional participating database is encountered the before look file is given a UNIX filename associating it with the extra database. Once all the databases have been recorded the number of databases is written to the header and the status value incremented to 1.

9.1.2 Check all the imported objects

To enable every object changed by a commit to be restored, copies of the objects are made in the before look file. The source for each copy is an object's database's data file. This provides an object's value prior to the commit.

While the commit is checking the objects to be committed it will encounter every changed object imported from the persistent store. These are the objects that may require to be restored if a commit should fail. Hence it is convenient to merge the copying of the changed objects with this checking phase of commit.

For each object to be copied it is necessary to include the address of the object in the before look file. However a commit may modify the address mappings for a database's pids. If this modification should fail for whatever reason then the pids of the changed objects would not be sufficient to identify their positions in their databases' data files. Therefore the address information is held as the database number and address in the database's data file that the pid would map to. Hence the following format is used for each object copied into the before look file:

```
        -------------------------------------------------
        | size   | object's | object's   |    a copy    |
    ... | in     | database | address in |... of the ...| ...
        | bytes  | number   | data file  |    object    |
        -------------------------------------------------
           4 bytes  4 bytes    4 bytes      size bytes
```

Figure 9.2 An entry for a changed object

A size field is included in front of each object to speed up scanning of the before look file during recovery.

Once this phase of commit has been completed the status value in the before look file is set to 2. Up to this point in commit no modifications have been made to the persistent store. However the next step of the commit does modify the persistent store. Hence a flag is set in the CPOMS that will cause the before look file to be preserved if a fatal error occurs. This ensures that the commit can be undone if any error should occur later in the commit.

9.1.3 Allocating pids to new objects

The allocation of pids to new objects may modify two parts of the persistent store, the index files of the databases pids are allocated to and the partitions file.

Any modifications made to a database's index file can be undone using the old value of the database's root object. This provides the correct lengths of the data and index files together with the start of the index file free list. To restore the index file it is sufficient to restore the free list. Any information past the recorded ends of the data and index files is simply ignored. The free list can be restored by scanning it from the old start of the free list. Since the free list is strictly ordered the scanning can stop when a free list entry is found or the correct end of the index file is reached, see section 8.2.3. When an object is allocated to a database it is put at the end of the data file. Therefore old free list entries can be identified because they will contain data file addresses greater than the correct data file length.

The root object of each participating database is marked as changed in step 1 of a commit, see section 8.2.1. Therefore the step 2 of commit will have made a copy of the participating databases' root objects in the before look file. Hence this step of commit need not record any further information to support the restoration of database index files.

The partitions file is modified when a database is allocated a new partition. A record of this allocation is made in the partitions file and in the partitions vector of the database. The partitions file does not contain information describing the order in which partitions are allocated to databases. Hence it cannot be used to free partitions allocated during a failed commit. The new partitions vector cannot be used either, because a commit may fail before it can write the partitions vector to its database. Hence modifications made to the partitions file must be explicitly recorded in the before look file.

When each new partition is allocated during a commit an entry is appended to the before look file. To identify where the partition entries start in the before look file the file size field of the header is used. This field is set at the end of step 2 of commit. Any entries in the before look file after the recorded length are partition entries.

Each partition entry consists of a partition number and a database number. The reason for identifying the database a partition was allocated to is to control the recovery mechanism. The recovery mechanism must be resilient to failures, so if it fails it can be retried. So if an attempt to restore a database frees a partition and subsequently fails it may be possible for that partition to be reallocated before the next attempt. Hence a check is included to see that a partition has not been reallocated to a different database.

The format of the partition entries is as
follows:

```
   --------------------------------------
...| partition number | database number |...
   --------------------------------------
         4 bytes            4 bytes
```

Figure 9.3 An entry for a newly allocated partition

Once this phase of the commit is complete the status field in the header of the before look file is set to 3. At this point in a commit the before look file will contain all the information necessary to undo the commit. Therefore the before look file can now be closed.

The complete format of the before look file is as follows:

```
-------------------------------- --------------- ------------ ---------
| number    | status | file   |   | database |     changed      new   |
| of        | of     |        |...|          |...          ...        |
| databases | commit | size   |   | number   |     objects      parts |
-------------------------------- --------------- ------------ ---------
  4 bytes    4 bytes   4 bytes      4 bytes      figure 9.2   figure 9.3
```

Figure 9.4 The complete before look file.

9.1.4 Copying objects to the persistent store

This part of commit performs the final set of modifications to the persistent store. When it is complete the commit can be considered successful. The before look file should then be destroyed.

The method used to destroy the before look file is made up of two steps. First of all the UNIX system call 'creat', is used to truncate the before look file to 0 length. If a failure occurs after this has been done the commit will remain intact. The UNIX file names given to the before look file are then removed from the UNIX directory holding the databases. When the last file name is removed the before look file is deleted by the UNIX file system.

This completes the modifications to the commit algorithm necessary to support commit's automatic recovery mechanism.

9.2 Invoking the automatic recovery mechanism

The automatic recovery mechanism can be invoked on three occasions.

a) During a commit.
If an error occurs during a commit the CPOMS will attempt to undo the modifications made to both the persistent store and the internal CPOMS data structures. If this can be successfully done the program calling the commit can be allowed to continue. However if the recovery mechanism is not successful or the error is too serious the program calling the commit will be aborted. The commit's before look file will then remain associated with each participating database.

b) During an open.
When an open is performed a check is made to see if a before look file is present. If a before look file is found then the recovery mechanism must be invoked because the database being opened may not be in a consistent state. The recovery mechanism will then attempt to restore all the databases that participated in the commit that created the before look file. Only if the recovery mechanism completes successfully will the open be allowed to continue.

c) When explicitly requested.
As an alternative to restoring a database during an open a standard function is provided. The standard function ignores the CPOMS locking protocol. The reason for providing this standard function is to allow the persistent store to be restored after a program or system crash that prevented some databases being unlocked. In such a situation the ability to force the recovery mechanism to be invoked is essential.

The operation of the recovery mechanism will now be described for each of the three methods of invocation. The reason for doing this is that each invocation has a different set of information available to it and also a slightly different function. For example if the recovery mechanism is invoked during a commit, the information used to construct the before look file will be available.

9.3 Automatic recovery invoked during a commit

When an error occurs during a commit the recovery mechanism must restore both the persistent store and the internal CPOMS data structures used. Since the work involved in performing the recovery mechanism is so closely tied to the progress of the commit each step of a commit will be considered in turn.

9.3.1 Finding the participating databases

In this step in a commit the persistent store is not modified. Therefore the only action required to restore the persistent store is to destroy the before look file. This is performed as described above using the UNIX system call 'creat'.

This step of commit modifies the list of open databases by adding copies of the root objects of the participating databases. Therefore the list can be restored by deleting the pointers to the copies from the list.

9.3.2 Check all the imported objects

This step in commit does not modify any internal CPOMS data structures or any part of the persistent store. Therefore the action required is the same as for the previous step of commit.

9.3.3 Allocating pids to new objects

This step of commit will modify the index files of the participating databases and possibly the partitions file. These changes are reflected in the changes made to the following CPOMS data structures. Hence the index files and the partitions file can be restored while restoring the affected data structures. These data structures are:

a) The pid to local address map, PIDLAM.
The PIDLAM will contain an entry for every object allocated a pid during the commit. Before a commit is performed all the temporary markers in the PIDLAM are reset. Therefore entries made during the commit will have temporary markers on them and can be easily removed to restore the PIDLAM.

b) The partition to database and index map, PTODI.
This map is modified every time a new partition is allocated during the commit. It can be restored by keeping a pointer to the copy in use before the commit. Restoring this pointer is sufficient to restore the PTODI because all modifications to the PTODI are done by copying.

c) The list of writeable partitions.
This list will be modified at the same time as the PTODI. All updates to the list are performed by copying. Therefore the list can be restored in the same way as the PTODI.

When this data structure is restored the old and new versions of the list are compared. The differences are those partitions allocated during the commit. Therefore when the list of writeable partitions is restored the partitions file can also be restored. Any newly allocated partitions are freed by placing the value -1 in their position in the partitions file.

d) The list of open databases.
The list of open databases includes information on the current lengths of the database files in use. To allow these values to be restored a copy is made of the root objects at the start of a commit. Therefore if an error occurs the root objects can be restored to their values prior to the commit. It should be noted that the value of a root object prior to a commit may be different to its value in the persistent store. However the information relating to the data and index files will be the same. Hence the old values of the file lengths and index file start positions are available and can be used to restore the index files, see section 9.1.3.

Once these data structures have been restored the persistent store will have been restored. The recovery actions for the previous step of commit are now used to destroy the before look file and so complete the recovery. It is important to ensure that the before look file is not destroyed until the recovery mechanism completes successfully. If the recovery mechanism fails the program calling the commit is aborted. Therefore it is necessary to preserve the information necessary to retry the recovery.

9.3.4 Copying objects to the persistent store

This step of commit modifies the data files of the participating databases. So in addition to the recovery action required by the previous step of commit all the objects that may have been changed must be restored. To restore these changed objects the before look file must be used to retrieve the copies made by step 2 of the commit. The before look file is scanned from the end of the list of databases to the recorded file length in the its header. Each object found is copied back to its position in its database's data file.

When this has been completed the recovery actions for the previous step of commit are performed. The persistent store and the internal CPOMS data structures will then have been restored to their state before the commit was called.

9.4 Automatic recovery invoked during an open

When the automatic recovery mechanism is invoked during an open the CPOMS has no information directly available. Therefore the first action taken is to open the before look file and analyse its header.

9.4.1 Analysing the before look file header

When the before look file is opened an attempt is made to read its header. If there is no header present either the before look file had just been created or the commit had completed successfully. In both cases no recovery action is possible and the before look file can be deleted.

If the header is present the status value indicating the progress of the before look file's commit is consulted. This progress information is used to determine what action is necessary to restore the databases that participated in the commit. There are 4 possible status values, 0,1,2 and 3. The recovery actions necessary for each one will now be described.

9.4.2 Recovery actions for status value 0

A status value of 0 indicates that the before look file had just been created. No changes would have been made to the persistent store so the only action required is to destroy the before look file.

The before look file is destroyed using a variation of the two step method described in 9.1.4. The first step, truncating the before look file to 0 length is the same. However the second step must ensure that it only removes UNIX file names belonging to the before look file that was opened. This is because the commit that created the before look file may have failed while giving the before look file additional UNIX file names. In such a situation the before look file may have recorded some participating databases that do not have a UNIX file name for the before look file. Any such databases may be freely used by other commits since they were not modified. If one of these databases participates in another commit that fails that commit's before look file will be given a UNIX file name for the database.

The UNIX file names used to associate a before look file with a database are unique to the database not the before look file, see appendix 1. Therefore in the situation just described the first commit would appear to have given its before look file a UNIX file name for the reused database. Hence the need to check which before look file a UNIX file name belongs to.

The checking is done by using the UNIX 'i-node' number given to the before look file. An 'i-node' number is a unique identification of a file on a device, e.g. a disk. Therefore each UNIX file name that may have been given to the before look file can be checked. If the 'i-node' number of the named file is equal to the 'i-node' number of the before look file then the named file is the before look file. If a UNIX file name passes this test it can be safely removed.

9.4.3 Recovery actions for status value 1

The status value 1 is set while the objects to be changed by the commit are being copied to the before look file. No changes will have been made to the persistent store but all the additional UNIX file names will have been allocated to the before look file. Therefore the recovery action required is simply the destruction of the before look file as described in section 9.1.4.

9.4.4 Recovery actions for status value 2
A status value of 2 indicates that the index files of the participating databases and possibly the partitions file may have been modified. Therefore it is necessary to restore both the database index files and the partitions file. The action necessary can be divided into 4 steps.

a) Open the participating databases.
As described above the root object of a database contains the information necessary to restore its index file. Hence it is necessary to open the databases that participated in the commit. However the open is only progressed as far as retrieving the root object. This is because a database's root object and the access mechanisms for the database's data and index files are all that is required by the recovery mechanism. At the end of this step the list of open databases contains an entry for every database that participated in the commit.

b) Free newly allocated partitions.
Partitions allocated during the commit are now freed. A list of the partitions allocated is held at the end of the before look file, see section 9.1.3. Each partition's entry in the before look file is compared with its entry in the partitions file. If the entries are the same the partition is freed by using the value -1 as its partitions file entry. If the entries disagreed the partition would have already been freed so no action would be necessary. A partition could already have been freed if a previous recovery failed after this step.

c) Restore the index files are restored.
The index files are now restored using the information held in the database root objects. This is described above, see section 9.1.3.

d) Destroy the before look file.
The before look file can now be destroyed in the same way as for a status value of 1.

9.4.5 Recovery actions for status value 3
If a status value of 3 is found then an error occurred while the database data files were being updated. The data files are updated as the last part of a commit. Therefore the database data files, index files and the partitions file must all be restored. The recovery action necessary is almost the same as for a status value of 3 but with an additional step between steps a and b.

To be able to perform step b the database root objects must have their values restored to what they were prior to the commit. Therefore the database data files must be restored. This is done by copying the data objects held in the before look file back to their databases. When this has been done the database data files will have been restored. However step a, the opening of the databases, may have imported the new values of the database root objects. Therefore the last action of this step is to copy the old values of the database root objects to the list of open databases.

The recovery actions for a status value of 3 can now be continued from step b.

9.4.6 Tidying up after the recovery actions
When the necessary recovery actions have been performed the persistent store will have been restored to its state prior to the commit. However the process of opening the participating databases will have modified some of the internal CPOMS data structures. Therefore these data structures must be restored and any opened databases closed. This is done by using the mechanisms for recovering from a failed open, see section 6.5.

9.5 Automatic recovery explicitly invoked

If the recovery mechanism is invoked explicitly it will be for one of two reasons. Either a database has been left locked by an aborted PS-algol program or a system crash occurred and the system manager wishes to restore any affected databases. In both cases the first action of the recovery mechanism is to reset the locks held on the database it is restoring. If a before look file is present then the recovery mechanism employed during an open is performed and the affected databases are unlocked. The unlocking of a database is only performed if the database is associated via a UNIX file name with the opened before look file, see section 9.4.2.

Appendix 1

The CPOMS interface to the UNIX file system

The purpose of this appendix is to describe the naming conventions for the UNIX files used to implement the persistent store and the primitive transaction mechanisms used to protect them.

The naming conventions for databases

The UNIX files that hold the persistent store comprise three types of files, data files, index files and before look files. All of these file types must be able to be associated with each database. Therefore the naming convention used is a prefix to indicate the type of the file and a suffix to identify the database. The prefixes are 'DATA' for data files, 'INDX' for index files and 'BFRE' for before look files. A suffix for a particular database is formed from the database's number. The format of the suffix is the database number in octal, reversed and extended to 5 digits by appending 0's. For example database number 17 (21 octal) would have its three files named as: DATA12000, INDX12000 and BFRE12000.

The naming convention for the database directory file

The naming convention used for the database directory file is derived from the recovery mechanism used to protect it. The recovery mechanism is based on making a copy of the directory file and then swapping the copy with the original at the end of an update. The file name used for the original version of the database directory is 'DIRIN' whereas 'DIROUT' is used for the copy. Assuming the file 'DIRIN' is a consistent version of the database directory, an update could be performed as follows.

a) The 'DIRIN' file is read.
The 'DIRIN' file is copied into the PS-algol heap and any necessary modifications are made to this copy.

b) The 'DIROUT' file is deleted.
Files cannot be explicitly deleted in UNIX, only file names can. This is because files may be given multiple names. Hence this step deletes the file name 'DIROUT'. The reason for performing this step is given below.

c) A new 'DIROUT' file is created.
A new 'DIROUT' file is created and the new copy of the database directory is written to it. Any failures up to the completion of this part of the update will leave a consistent version of the database directory in the file 'DIRIN'.

d) The 'DIRIN' file is deleted.
The file name 'DIRIN' is deleted.

e) The 'DIROUT' file is renamed 'DIRIN'.
The act of renaming a file is made up of two stages. First the file to be renamed is given the new file name. Then the old file name is deleted. Therefore the file 'DIROUT' will be given the additional file name 'DIRIN'. Then the file name 'DIROUT' will be deleted.

If a failure occurs before step e is complete there may not be a file present with the name 'DIRIN'. However there will be a file present with the name 'DIROUT' that is a consistent version of the database directory. Therefore the above algorithm should be preceded by a check for this condition and if necessary the execution of step e.

If a failure occurs in step e just before the file name 'DIROUT' is deleted, there will be two file names for the same file. As described above the UNIX system call 'creat' truncates an existing file to 0 length. Hence if step c is performed on the file name 'DIROUT' the only copy of the database directory on disk will be destroyed. Hence step b is included so as to avoid any problems in step c.

The naming convention for the partitions file
The partitions file is named and updated in a similar way to the database directory. The only difference is that the file names used are 'PARTSIN' and PARTSOUT' instead of 'DIRIN' and 'DIROUT'.

Controlling concurrent access to the database directory and partitions file
The locking protocol used by the CPOMS on its databases guarantees that data, index and before look files will never be used incompatibly by different incarnations of the CPOMS. i.e. these files will never be written on if a different incarnation of the CPOMS is reading them. However this protocol involves updating the directory file and so cannot be used for the directory and partitions files. Therefore a different locking mechanism must be employed to ensure mutual exclusion on these files.

This is achieved by using a dummy file called 'LNKFILE' and giving it an additional agreed name when access to one of the files is required. The name used for access to the database directory is 'DIRLOCK' and the name used for the partitions file is 'PARTSLOCK'. The UNIX system call 'link' is used to give a file an additional file name. It is performed as an atomic operation and will only succeed if the additional file name does not already exist. Therefore this enforced use of 'link' ensures that only one incarnation of the CPOMS can access the database directory or partitions file at any given time.

When an incarnation of the CPOMS has finished operating on either the database directory or the partitions file it is required to delete the relevant file name. This is performed using the UNIX system call 'unlink'. If for any reason an incarnation of the CPOMS is unable to do this the database directory or the partitions file will remain permanently locked. In such a situation it is necessary for someone to explicitly remove the offending lock files. However great care should be taken to ensure that this problem has indeed occurred before taking such drastic action.

The interfaces used to the UNIX file system
The CPOMS uses the following UNIX system calls and library routines to access the UNIX file system: 'open', 'creat', 'lseek', 'read', 'write', 'link', 'unlink', 'stat' and 'sync'. UNIX files are assumed to be a stream of bytes addressed from 0. If a UNIX file is accessed past its end then the recorded length of the file is increased but no extra space is allocated to it. Access to unwritten parts of the file will yield the null character as the result of a read, whereas a write causes space to be allocated to the file. This file organisation enables the CPOMS to address files in bytes, using the 'lseek' system call, without worrying about any kind of file structure. To implement the CPOMS on some other type of file store would require the provision of a similar addressing facility.

The system calls 'link' and 'unlink' provide a file naming utility that can be used to implement a locking protocol for concurrency control. This is described above.

To enable the CPOMS to boot strap itself from an empty UNIX directory it requires the use of the system call 'stat'. The system call 'stat' is used to interrogate the status of a named file. If 'stat' cannot perform its function for the supplied file name then the file name is not in use. Using this information the CPOMS can create any files it needs and so establish a new persistent store in an empty UNIX directory.

The system call 'sync' is utilised to ensure that modifications to parts of the persistent store are up to date on disk. The effect of 'sync' is to force a UNIX system to flush its disk buffers and so check point the file system. Where possible a more selective synchronisation function should be used to reduce the performance degradation of the host UNIX system. For example on 4.2BSD a system call 'fsync' is available. This restricts the operation of 'sync' to a single file.

Appendix 2

The runtime initialisation of the CPOMS

The CPOMS user interface

The CPOMS user interface is comprised of two levels. The first is a set of standard functions, 'opendb', 'createdb', and 'commit'. Each of these is described above by chapters 6,7 and 8 respectively.

This interface shows the root of a database as an object containing a single pointer. A user must therefore evolve his own strategy for structuring the data he wishes to store in a database. To simplify a user's task a higher level interface is provided that presents a table package as the root object of a database. This provides the user with a convenient abstract data type to control the initial storage of his data in a database.

Creating the CPOMS user interface at system creation

When a PS-algol system is first set up there is no tables package available to place at the root of the system database. This problem is overcome by making the program that declares the tables package create the system database. It does this in 5 steps:

a) Create the system database.
The standard function 'createdb' is used to create the system database and return a pointer to the database's root object.

b) Create the tables package.
The tables package is declared as an abstract data type. When a table is created it is represented by a structure containing a procedure for each of the operations allowed on the table. All the data held by a table is encapsulated within these procedure.

c) Create a table to be the root of the system database.
An instance of a table package is created and the table handling standard functions are given entries in it. This program also declares the procedures that perform the real I/O for PS-algol. They are also entered into this table.

d) Assign the table to the root object of the database.
The root object of the system database is then made to point to the table.

e) Commit the changes made to the system database.
Finally a commit is performed that will copy the new table and all its entries into the system database. The system database has now been initialised.

Creating the CPOMS user interface at runtime

This organisation of the system database also presents a problem at runtime initialisation. The problem is how can the contents of the table held in the system database be accessed when the standard functions used to access a table are among the contents. The solution to this problem is provided by the low level interface. First, the system database is opened using 'opendb' and a pointer to the root object returned. The root object contains a pointer to the table holding the desired information. However this table is represented by a structure containing its own access procedures. So if the structure class of a table package is known these procedures can be accessed and therefore so can the contents of a table.

This runtime initialisation is performed by a PS-algol program that is run before a user's program. Its function is to retrieve standard functions from the system database and use them to initialise the standard frame. It is written in such a way that it will continue even if it cannot find all the standard functions it requires. In this way the initialisation of the system database can be done from within PS-algol.

The standard functions 'create.database' and 'open.database'
The two standard functions provided by the higher level interface are 'create.database' and 'open.database'. The PS-algol initialisation program implements them as follows using the table standard functions it has retrieved from the system database. Note that if the system database was not opened correctly this interface will not be usable.

```
! the root object of a database has the following structure class.
structure opdb.result( pntr root.of.db )

! if the system database was opened successfully system is a pointer to a table package.
! tab.pac is the name of the table package's structure class.
create.database := proc( cstring path.name,pass -> pntr )
if system is tab.pac then
begin
        ! the tables are available so create the database.
        let result = createdb( path.name,pass )
        if result is opdb.result then
        begin
                let tab = table()
                result( root.of.db ) := tab
                tab
        end else result
end else
        error.record( "create.database","create.database not defined",
                              "the system database could not be opened" )

open.database := proc( cstring path.name,pass,mode -> pntr )
if system is tab.pac then
begin
        ! the tables are available so open the database.
        let result = opendb( path.name,pass,if mode = "read" then 0 else 2 )
        if result is opdb.result then
        begin
                if result( root.of.db ) is tab.pac then result( root.of.db ) else
                if mode = "write" then
                begin
                        ! there isn't a table in the database so give it one.
                        let tab = table()
                        result( root.of.db ) := tab
                        tab
                end else
                error.record( "open.database","root is not a table",
                   "open.database assumes the database has a table for its root structure." )
        end else result
end else
        error.record( "open.database","open.database not defined",
                        "the system database could not be opened" )
```

The allocation of reserved pids to objects
At runtime initialisation the PS-algol interpreter creates a set of single character strings, reserves a special set of addresses for non PS-algol standard functions and creates a structure for the pointer value **nil**. The main reason for doing this is to promote runtime efficiency. In the same way the CPOMS allocates a set of reserved addresses called reserved pids. There are 4 groups of these reserved pids.

a) The pointer value **nil**.
This pointer value is widely used in the persistent store so it is essential to have a unique value for it. It actually points to a structure so the trademark of the structure is also given a reserved pid.= This is necessary for error reporting when a **nil** pointer is accidently dereferenced.

b) The single character strings.
The single character strings are included for efficiency reasons. A string can never be modified so there is no need to have more than one copy of any given string. Hence the single character strings are all given reserved pids to prevent them being duplicated.

c) The non PS-algol standard functions.
The non PS-algol standard functions are all identified as integers with values less than 100. These integers are not legal pointer values so they are specially handled by the PS-algol interpreter. The reason for giving each one a reserved pid is to ensure that they do not need special treatment in the persistent store.

d) Some of the structure trademarks.
Every compiled PS-algol program has a prelude of structure definitions prefixed to it. These structure definitions are used by the PS-algol graphics and for the root of a database. Since all these structure definitions are string literals there is no need for them to be duplicated in the persistent store. Therefore a suitable selection of these structure definitions are given reserved pids.

When the PS-algol initialisation program has been loaded by the PS-algol interpreter a vector containing its structure definitions is also loaded. This program is always loaded with every user program and never changes.
Therefore its vector of structure definitions remains constant and can be used to allocate the reserved pids.

It is assumed by the CPOMS that the first structure definition in the initialisation program is for an 'error.record' and the second is for the root object of a database. This is important because this assumption is used to calculate the pids used when a database is created.

Appendix 3

Error handling and reporting by the CPOMS

Error reporting by the CPOMS

Where ever possible the CPOMS will report any errors it was able to recover from to the calling user program. This is done using the results of the standard functions that can access the CPOMS. The result values are in the form of pointers to instances of the structure class 'error.record'.

structure error.record(**pntr** error.context,error.fault,error.explain)

The purpose of the 3 fields of an 'error.record' are:

error.context:
This field gives the name of the standard function that was executing when the error occurred.

error.fault:
This field gives a short description of the error that occurred.

error.explain:
This field is intended to give a more detailed account of the error that occurred. At present this field tends not to be used.

The CPOMS has access to a list of possible 'error.records' in a 2 dimensional vector held in the standard frame. The first dimension is for the context and the second for a particular error in that context. This vector is kept in the system database and made accessible from the standard frame by the PS-algol initialisation program. If for some reason an entry is not present in the vector for a particular error then the CPOMS will report this with the indices it tried to use. A listing of the available 'error.records' can then be consulted to ascertain what the error was, see appendix 4.

Error handling by the CPOMS

If a runtime error occurs the CPOMS must be informed so that it can tidy up the persistent store. This usually involves unlocking any databases it has open. In addition if a commit is in progress that has not modified the persistent store the tidying up will include deleting the before look file.

The decisions on what tidying action is necessary is provided by a set of global flags maintained by the PS-algol interpreter. For example flags are present to indicate if the initialisation of the runtime system is complete, if the CPOMS has been initialised and so on. Therefore no matter when a runtime error occurs the CPOMS will be able to take the correct action to tidy up the persistent store.

Appendix 4

The CPOMS error records

```
! the structure class used to hold the vector of error records in the system database.
structure record.cont( c*c*cpntr whats.wrong )

! the structure class for an error.record
structure error.record( cstring error.context,error.fault,error.explain )

let the.records =
@1 of c*cpntr[
! the errors that may occur in a commit.
        @1 of cpntr[ error.record( "commit","no databases available","" ),
                error.record( "commit","cannot create before look file","" ),
                error.record( "commit","cannot initialise before look file","" ),
                error.record( "commit","cannot create links to before look file","" ),
                error.record( "commit","attempt to modify read only database","" ),
                error.record( "commit","write failed to the before look file","" ),
                error.record( "commit","error while allocating new pids","" ),
                error.record( "commit","write failed to a database","" ) ],

! the errors that may occur while creating a database.
        @1 of cpntr[ error.record( "createdb","no databases available","" ),
                error.record( "createdb","database directory locked","" ),
                error.record( "createdb","cannot read directory file","" ),
                error.record( "createdb","cannot write directory file","" ),
                error.record( "createdb","name is already in use","" ),
                error.record( "createdb","cannot add directory entry","" ),
                error.record( "createdb","cannot create data file","" ),
                error.record( "createdb","cannot initialise data file","" ),
                error.record( "createdb","cannot create index file","" ),
                error.record( "createdb","cannot initialise index file","" ),
                error.record( "createdb","cannot lock the new database","" ),
                error.record( "createdb","cannot open data file","" ),
                error.record( "createdb","cannot read data file","" ),
                error.record( "createdb","cannot open index file","" ) ],

! the errors that may occur while opening a database.
        @1 of cpntr[ error.record( "opendb","no databases available","" ),
                error.record( "opendb","database directory locked","" ),
                error.record( "opendb","cannot read directory file","" ),
                error.record( "opendb","cannot write directory file","" ),
                error.record( "opendb","cannot find database","" ),
                error.record( "opendb","password is incorrect","" ),
                error.record( "opendb","database is locked","" ),
                error.record( "opendb","cannot open data file","" ),
                error.record( "opendb","cannot read data file","" ),
                error.record( "opendb","cannot open index file","" ),
                error.record( "opendb","database needs restoring","" ) ],
! the errors that may occur while changing a database's password.
        @1 of cpntr[ error.record( "change.passwd","no databases available","" ),
                error.record( "change.passwd","database directory locked","" ),
                error.record( "change.passwd","cannot read directory file","" ),
                error.record( "change.passwd","cannot write directory file","" ),
                error.record( "change.passwd","cannot find database","" ),
                error.record( "change.passwd","password is incorrect","" ),
                error.record( "change.passwd","not owner of database","" ),
                error.record( "change.passwd","database is locked","" ) ],

! the errors that may occur while changing a database's owner.
        @1 of cpntr[ error.record( "change.owner","no databases available","" ),
                error.record( "change.owner","database directory locked","" ),
                error.record( "change.owner","cannot read directory file","" ),
                error.record( "change.owner","cannot write directory file","" ),
                error.record( "change.owner","cannot find database","" ),
                error.record( "change.owner","not PS-algol system","" ),
                error.record( "change.owner","database is locked","" ) ],

! the errors that may occur while removing a database.
        @1 of cpntr[ error.record( "remove.database","no databases available","" ),
                error.record( "remove.database","database directory locked","" ),
                error.record( "remove.database","cannot read directory file","" ),
                error.record( "remove.database","cannot write directory file","" ),
                error.record( "remove.database","cannot find database","" ),
                error.record( "remove.database","password is incorrect","" ),
                error.record( "remove.database","not owner of database","" ),
                error.record( "remove.database","database is locked","" ) ],

! the errors that may occur while restoring a database.
        @1 of cpntr[ error.record( "restore.database","no databases available","" ),
                error.record( "restore.database","database directory locked","" ),
                error.record( "restore.database","cannot read directory file","" ),
                error.record( "restore.database","cannot write directory file","" ),
                error.record( "restore.database","cannot find database","" ),
                error.record( "restore.database","restore failed","" ) ],

! the errors that may occur while renaming a database.
        @1 of cpntr[ error.record( "rename.database","no databases available","" ),
                error.record( "rename.database","database directory locked","" ),
                error.record( "rename.database","cannot read directory file","" ),
                error.record( "rename.database","cannot write directory file","" ),
                error.record( "rename.database","name is already in use","" ),
                error.record( "rename.database","cannot find database","" ),
                error.record( "rename.database","password is incorrect","" ),
                error.record( "rename.database","not owner of database","" ),
                error.record( "rename.database","database is locked","" ) ],

! the errors that may occur while listing the database directory.
        @1 of cpntr[ error.record( "list.database.dir","no databases available","" ),
                error.record( "list.database.dir","database directory locked","" ),
                error.record( "list.database.dir","cannot read directory file","" ) ],
! the errors that may occur while attempting to invoke the disk garbage collector.
        @1 of cpntr[ error.record( "garbage","no databases available","" ),
                error.record( "garbage","not yet implemented","" ) ] ]
```