A Programming System

by

Peter Allan Buhr

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Winnipeg, Manitoba

A PROGRAMMING SYSTEM

BY

PETER ALLAN BUHR

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Currently, a Programming System is divided into two major subsystems: one subsystem deals with device memories (i.e. files on peripheral devices), and the other subsystem deals with main memory (i.e. programming languages). This dichotomy has resulted in separate development routes in both these areas, which in turn has resulted in problems of maintaining type consistency between these two subsystems.

To achieve consistency throughout the entire system, particularly between the interactive command language and program language modules, it is necessary to have files as objects in the program language. Currently, programming language constructs deal with the manipulation of storage that is directly accessible (i.e. in main memory). However, files are not directly accessible and as a result constructs available to define files and subsequently to create and use them must all reflect this fact. Programming language constructs that provide these facilities are defined and their semantic action is discussed.

A single programming language, containing file definition and accessing features, is used to support both interactive program invocation and program definition. As well, all programming language features traditionally found only in conventional programming are available interactively. Interactive programming requires both an existing data area to store instances of data items and a symbol table containing information about these instances. An interactive data area and symbol table are indivisible, and must be retained between user terminal sessions. To support this capability a new entity is introduced into the system called an environment.

Maintaining system wide consistency also requires retention of the symbol tables for all predefined programs along with the symbol tables for interactive data items. The retention of symbol tables induces a new style of programming in which all declarative statements are made through a special editor. More importantly, the editor also allows modifications to be made to previously entered declarations. Consequently, the editor can detect changes that affect consistency and thereby cause automatic recompilation of affected programs.

Secondary issues such as addressing, security, and scheduling are all discussed in this new Programming System framework.

ACKNOWLEDGEMENTS

I would like to acknowledge the time, patience and commitment by Bob. You have taught me much during this thesis, much more than you think. I have always admired you ever since I met you, and my admiration has only continued to grow. Your patience with me over the years has been amazing and your commitment to the highest quality of work has driven me to achieve nothing less than my best.

I would like to acknowledge the tender loving care and unfailing encouragement of Lauri. She helped me through the deepest depressions, and the largest setbacks with constant reassurance and a never ending faith in me. As well, I owe her a lot for the extra work she did so that I could write. I may never have said thank you at the time, but I always deeply appreciated it. So let me say thank you now for all the sacrifices you made so that this thesis could come into being.

I would like to apologize to my friends for being grumpy, sharp, and distant at times. This behavior was not the result of anything you did, but just my frustration with myself and the thesis. I hope I will be a more congenial person in the future.

Lastly, I would like to acknowledge my poor fingers, who absorbed a tremendous beating during the course of this thesis.

CONTENTS

Chapter I

INTRODUCTION

> This is a story of how a Baggins had an adventure,
> and found himself doing and saying things altogether unexpected.
> He may have lost the neighbours' respect, but he gained
> – well, you will see whether he gained anything in the end.
>
> – J. R. R. Tolkien, The Hobbit, Chapter I

This thesis presents a design for an integrated programming system which provides a coherent and productive program development environment. It is expected that this system will be used by professional programmers to design and write large scale applications programs for both professional and non-professional computer users. I feel that there is a need for this type of development system because:

- Few of the existing development systems provide the tools that are necessary to provide a truly coherent and productive program development environment. The types of tools that I feel are poorly implemented or lacking in existing conventional systems (MVS [IBM's System 370 Multiple Virtual Storage operating system], UNIX [an operating system developed at Bell Laboratories], Multics [a Honeywell operating system]) are:
    - A command language for interacting with the machine which includes more than just simple control structures, character string variables, and macro substitution.
    - A variety of file types and the facilities to easily define or customize my own. This must be accompanied by the ability to use these new file types in the same way as the ones supplied with the system.
    - A programming language that supports abstraction and encapsulation facilities.
    - A powerful symbolic debugger that allows interaction in the programming language.

  - A program management subsystem that provides incremental program development and maintains consistency among constantly changing programs.

  While some of these facilities are present in conventional systems few have them all, and these facilities are not always as powerful as is necessary.

- Different parts of a system may be poorly integrated with one another; in particular:
  - Programs written in one programming language cannot always be used by another programming language, thereby causing duplication of algorithms in several programming languages. Particular areas of incompatibility among different programming languages are types, procedure invocation and access to files.
  - The programming languages and the interactive command language are largely non-compatible. This is because the interface between them is poor or ill defined.
  - Files are often defined outside of any user accessible programming language or in assembly language. As a result type checking cannot be done properly for file references.

  As a result, program development on such systems takes far too long since too much time and effort is spent in re-inventing tools.

Thus, I see a need for a powerful integrated programming system that can form the basis for much of the software that will be written in the future.

1.1 **THE SYSTEM**

Traditionally, programming language and operating system development has proceeded along independent paths. As programming languages became more complex, programming environments were introduced to provide the necessary non-language facilities that are needed to support program development and maintenance of application software [USDOD80a].

The programming environment, in turn, overlaps the conventional domain of the operating system, such as linkage editors or binders which must type check as well as bind programs together. As well, the programming environment is required to supply an interactive command language, that is itself a simple programming language, to provide dynamic interaction between programmer and the programming language.

Is it possible to extend the original programming language out to the user, so that the programming environment is provided by the programming language itself? By extending the language to provide the interactive interface between the user and the system as well as conventional programming, the user is given a <u>programming</u> <u>system</u> that is consistent within the programming language paradigm. To support this concept requires that certain functions conventionally associated with the operating system be incorporated into the programming language and vice versa.

1.2 MOTIVATION

I was motivated to work on an integrated programming system out of frustration which occured during development of different programming tools on several existing systems. My frustration resulted from having to constantly fight with these systems to accomplish my goals. Even if I was successful in developing a new tool, I found it was not supported by any of the existing tools in the system. Also, changes to the system often rendered my work useless because data structures and data organizations would change. These systems did not provide enough abstractions to shield me against changes in the system.

1.3 THE PROGRAMMING LANGUAGE

Ultimately, it will be necessary to define a programming language to support the programming system. However, the actual definition of the programming language is not an objective of this thesis. Yet, it is not possible to discuss the programming system without describing some parts of it with a programming language. To this end, a programming language that uses features from many of the new programming languages (ADA [USDOD80b], ALGOL68 [WIJNG77], CLU [LISKO79], EUCLID [LAMPS77], MESA [MITCH79], MODULA [WIRTH77]) will be used with only an informal definition.

1.3.1 Compilation

In general, the term compilation is used to describe the translation of a programming language from the external user form (<u>source language</u>) into a form (<u>object code</u>) that is acceptable to an interpreter which executes the generated object code. The compilation phase can be trivial, in which case the interpreter executes the source language directly. Alternatively, the compilation phase can be extensive, in which case the object code produced is interpreted (<u>executed</u>) by the hardware. The traditional names for these two types of compilation are <u>interpretation</u> and <u>compilation</u>, respectively.

However, there is a spectrum of actions that can be performed by the compiler, in which the compilation phase translates the source language into an internal form, but not a form that is directly executable by the hardware without assistance by a software interpreter. This type of compilation is usually used in programming languages where there is little or no typing of variables. Unfortunately, this type of compilation is usually not differentiated from the traditional compilation [MITCH70].

The motivation for compiling a program is to increase the efficiency of the execution of the program. The gain can be both in space and time. However, there is a trade off involved in doing this. First, only particular kinds of programming languages lend themselves to compilation. These are programming languages that are statically typed. Secondly, the greater the translated form of the programming language differs from the original, the more information is lost about how the executing object code relates to the original source language. To achieve certain types of operations such as symbolic debugging, information about the translation must be retained. Hence, the type of compilation can greatly affect the operation and organization of the programming system.

1.3.2 Characteristics of the Programming Language

The programming language that is used in this thesis is a general purpose strongly-typed programming language intended for compilation.

The programming language is divided into 2 parts: declarations and control structures. The declaration of variables is as it is in ADA: declarations are restricted in where they can appear in the source language program.

As well, the programming language must support a powerful set of constructs for creating new tools. Furthermore, these constructs must be designed to allow encapsulation of data items and their associated routines. Finally, these constructs must be generalized so that they can be used with multiple types of data. All of these requirements must be met within the restrictions imposed by strong type checking at all times.

1.4 OBJECTIVES

From the onset of the design of the programming system, several objectives were laid out:

- To have a high-level programming language for use in interaction with the system, rather than the interpretive command languages used now. As well, this programming language must be the same programming language used to write all other programs in the programming system.
- To perform type checking on all interfaces between user programs and between the system and user programs.
- To insure that all operations performed on objects are consistent with the definition of the object, which might involve dynamically checking consistency between an instance and its type definition.
- To unify files with other data structures in the system so that all of the system can be described by the programming language (except possible for a small part of the operating system kernel which is machine dependent).
- To be able to compile all entities in the programming system and to allow separate compilation.
- To deal with the addressing of objects on storage devices, and allow the paging of data between storage devices with different access speeds.

1.5 THE THESIS

The thesis of this dissertation is that the ideas introduced in conventional programming languages and the ideas introduced in conventional operating systems can be amalgamated to form one integrated programming system (InterLisp [TEITE75], Magpie [SCHWA84], Mesa-Cedar [MITCH79, TEITE84], Smalltalk [GOLDB83]).

Part of the thesis introduces new programming language constructs and features that are needed to extend the storage management facilities necessary to support the notion of files. And a large part of the thesis is the engineering of the system using these new programming language constructs. The engineering aspect attempts to demonstrate that the new programming language facilities are powerful and flexible enough to achieve the objectives. As well, the engineering aspect attempts to convince the reader that such a system is implementable.

Neither the programming language constructs nor the engineering design presented in the thesis have been implemented. This is due to the size of the problem addressed by the thesis. Implementing an entire system would require several programmers working for several years. But even with these resources, attempting that implementation under existing systems would be a very tedious procedure because they provide few or none of the capabilities described in this thesis.

1.5.1 The Thesis Organization

To demonstrate this thesis the dissertation addresses four major goals:

1. Incorporating file definition into the programming language. Chapter 2 shows that file definition can be supported in the programming language by introducing a small number of new programming language constructs. When file definitions are incorporated into the programming language, files become a programming language type and as a result special statements/constructs dealing exclusively with files are not needed.

2. Providing an environment in which the programming language can be used interactively. Chapter 3 discusses the environment necessary to allow programming language statements to be entered by the user and executed immediately. With the inclusion of files into the programming language and the ability to use the programming language interactively, the programming language can replace the conventional interactive command language. Maintaining this environment and all the programs in the system leads to a new method for entering programs.
3. Addressing the addressing needs introduced by the system. Chapter 4 discusses some of the hardware and software requirements for addressing data and programs in an integrated system. Maintaining consistency among these addresses in a dynamically changing system is also discussed.
4. Providing a programming language facility on which security and arbitration can be implemented. There exist natural functional groupings inherent in encapsulated definitions. These groupings can be used to grant access to certain functional components of a definition, thus providing a security mechanism based on the functions of the definition and not on some arbitrary criterion. As well, these groupings can provide access information that can be used to arbitrate user access to an instance, again based on the functions of the definition. Both security and arbitration are discussed in this new context in chapter 5.

Chapter II

FILES

Currently, files and the file system are considered to be in the domain of the operating system. Files are defined in assembly language or a low-level system language (BLISS [WULF71], C [KERNI78], Mesa [MITCH79], [PL/S [IBM79b]); allocated, manipulated and freed by a special command language (JCL [IBM79a], SHELL [BOURN78], TSO [IBM78]); and file contents are manipulated by a host of general purpose programming languages. This organization of the file system has several disadvantages:

1. No consistency checking is carried out when files are accessed. Except possibly for verifying that the length of records desired by the user's program is the same as that provided by the file, no checks are made that the fields in the record are in any way related to those required by the user's program. Even new languages like Pascal and Ada rely on the user to correctly associate a file from the operating system with the correct type of variables in the programming language. Unless a programming environment which is program language sensitive (such as APSE [USDOD80a]) is provided, the compiler or run-time routines do not possess the information to perform the type checking. This is a fundamental flaw as type checking can only guarantee consistency for data types if all data types are type checked.
2. Since file definitions are embedded in the operating system, it is usually not possible to implement new file organizations. Few primitives and tools exist to help design and implement new file organizations or augment existing ones, and those tools that do exist require at least the knowledge of a system programmer to be used. Even if new file organizations could be defined, the existing interface between command language or programming language and files is not flexible enough to allow integration of a new file interface into the system. The existing command

languages are not generalized; hence they require modification to implement any new file definitions that do not conform to the existing interface. In programming languages which have constructs that implement only the file organization that the operating system already provides, new file interfaces cannot be added.

3. Because operating system and programming language development occurred separately, the behaviour of files is usually subtly different from their description in the programming language. Designers of operating systems and programming languages have yet to define a precise interface. For example, the way programming languages deal with files depends on which operating system they are running under.
4. There are many aspects of the behaviour of files that arise from the way they are implemented in the operating system, but which are completely irrelevant to proper use by a program. Many features of the operating system must be known by a user in order to create or access files, and to understand failure and problems that arise during their operation. In some systems, creating and manipulating files is such an art that only special users are allowed to perform these operations. This compounds the problem, as this often results in inconsistences between what a user wants and what is created for him. These problems arise because there is no single complete interface to a file.

These problems can be alleviated by incorporating files fully into the programming language, that is to say, to allow definition as well as access of files in the programming language. To accomplish this, the programming language must define new primitive constructs to allow file definition. The compiler can then maintain information about the file definitions, hence making available the necessary information for the compiler to type check all usages of the file. Also, a programming language which allows file manipulation greatly reduces the need for a command language; by using the programming language as the command language, new file organizations can be immediately incorporated into all parts of the system (i.e. command language and programming language).

While incorporating files into the programming language may not be the only way of solving these problems, it does provide a solution that is both elegant and complete. These properties make this solution most desirable. Why then have files not been incorporated into the programming languages already? Because there exists a great deal of complexity and many complications in providing file definition capabilities in the programming language and of using the programming language as the interactive command language. The remainder of this thesis is devoted to solving the complex issues involved in providing file access and definition within the programming language, and dealing with the ramifications of this solution.

2.1 FILE CHARACTERISTICS

2.1.1 Basic Characteristics

The usual concept of a file is a group of data records. However, this only embodies part of the notion of a file. A file is as much the method in which the records are managed and stored as the records themselves. It is not exact enough to talk about a file of records; to be precise, it must be a sequential or indexed or direct access file of records. All of these qualifiers for a file define the method of storage management that is used for the records, as well as the manner in which records are extracted by the user.

Storage management is the process of managing the free and used areas of a particular piece of storage. As well as managing this space, storage management involves allowing access to this space by an application program. Here storage management must provide the ability to obtain storage for use and possibly to free storage that is no longer needed. These requests for storage can come from application programs as well as from the compiler. Thus, storage management must be integrated into the system in such a way as to allow general access through a standard set of routines.

The desire to explicitly define storage management of files strongly influences the way in which files are implemented. Implementing files using existing data structures (such as an array of records) is precluded. This approach does not maintain the storage management property for the file, but instead forces the storage management to be implemented implicitly by the compiler. In general, the compiler cannot hope to anticipate all possible storage management techniques.

The approach that I have taken is to maintain storage management as an integral part of the definition of the file. This should not imply that all storage management properties necessarily have to be apparent to a user of the file; by using the abstraction facilities that are defined in the programming language, the details of storage management can be hidden from the user. The user sees only those details necessary to access the records of the file.

2.1.2 Conventional Characteristics

Although conventional file systems have several disadvantages, they also have many desirable characteristics that should be preserved. The following characteristics are considered fundamental for all file systems and have been retained in this system:

1. Files must be independently accessible and manageable. Access to the file must be done through an accessing technique peculiar to that file's organization. The storage area of the file must have a storage management technique peculiar to each file organization.
2. The storage area for a file will be able to expand and possibly contract dynamically. The exact size of a file is usually unknown, and some capability for extension and reduction of certain types of files is necessary.
3. The accessing of the file is of a transient nature. A file need only be accessible while a program is processing the information in the file.
4. Concurrent access to the file by multiple users must be provided for. This allows sharing of information within large files by multiple users.
5. Some files have to be <u>persistent</u>, that is to say, exist independently of active programs on the system. This is to be contrasted with entities like a program stack and heap that are strictly under the system's control. While the user may specify

through declarations what is contained in a program stack and heap, the user is not directly in control of their creation or deletion. The user can only control them indirectly via program invocation. All entities of this type are said to be <u>transient</u>.

6. There must be a means of denying or restricting access to files.
7. A single file definition must be able to accommodate different record types.

2.1.3 <u>New Requirements</u>

In addition to the above desirable characteristics of conventional files, several new characteristics are needed to eliminate the disadvantages stated earlier.

1. The declaration and usage of the file in the programming language must be done in a type safe manner.
2. The definition of the file must support explicit storage management of the storage area. This requires that the storage area for the file appear as a single contiguous stream of bytes. A storage area such as this, which is independently addressable and appears as a single contiguous stream of bytes, will be referred to as a memory or an <u>address space</u> (<u>AS</u>).
3. The programming language must provide the necessary constructs to allow files to be defined in the programming language. The syntax and semantics of these constructs must be such that all the desirable file characteristics are maintained.

In the following discussion and examples of files, file definition will be based on memory mapping the storage for a file. Memory mapping of storage was chosen over the more conventional reading and writing with buffers because the storage management of the file is easier to perform and far more flexible. This would be particularly important in complex file organizations like an index-sequential file. This does not preclude file definition using reading and writing with buffers. However, because the former method is the more complex, the latter is not dealt with in detail.

2.2 CONSEQUENCES OF REQUIREMENTS

The main consequence of trying to retain all the desirable characteristics for files is that some of the requirements lead to conflicting objectives. Solutions to these conflicts cannot always be found without compromising the desired characteristics. However, it is possible to resolve most of the conflicts by dividing the definition of a file into two levels.

Storage management is an integral part of the file definition. However, storage management requires computations on pointer variables, which is a type-unsafe operation. Also, storage management must deal with details of the file's operation which are irrelevant to a user creating and accessing a file. These consequences conflict with the characteristics to have type safety throughout the programming language, and the desire to free the user from irrelevant details in using the file.

The conflicts are resolved by separating the user aspects and the storage management aspect into two distinct levels. The user aspects are called the user level and the storage management aspects are called the storage management level. At the user level only those operations for creating and accessing the file are available, and all operations will be such that the compiler can verify consistency of types. The storage management level allows type-unsafe operations which are necessary to manage the contiguous addressable stream of bytes that are the data space (DS) for the file. In other systems, such as UNIX[1] [RITCH74] or the Multics system [ORGNI72] a DS corresponds largely to a file. The storage management level also contains all the details for storage management. Thus, type-unsafe operations and details of storage management are confined to the storage management level.

---

[1] UNIX is a Trademark of AT&T Bell Laboratories.

2.3 GENERALIZATIONS

These ideas developed from the desire to fully define files in the programming language, but they can be applied to a much more general situation. At the user level, a programming language construct called an OBJECT will be used to define all the operations permitted. At the storage management level, a programming language construct called a MEMORY will be used to define storage management for a DS. These constructs can be used to define other abstract entities that have unusual storage management requirements, such as a variable string package that implements string variables that can contain character strings of arbitrary size or a multi-precise number package that implements integer variables that can have an arbitrary number of digits. Thus, these programming language constructs provide more than just a solution to the problem of defining files.

2.4 OBJECT

An OBJECT is the programming language construct that implements the user level. An object is an instance of an OBJECT. The OBJECT definition contains the routines and data items (collectively called components) to access and modify a memory area, via type-safe operations. The OBJECT definition defines a new abstract type in the programming language, and is similar to the SIMULA class [DAHL72]. The object is not directly accessible, just as files are not directly accessible. Instead, it must be made accessible by an explicit declaration and is only accessible for a restricted period of time. This lack of accessibility reflects the fact that an object is in a memory different than the current program or data memory. Usually, the object creates a memory for each instance which contains its local variables; however, there may be multiple objects in a memory.

2.4.1 Object Definition

To illustrate the definition of an OBJECT, an example definition for a sequential file will be developed. A sequential file has the properties that records can only be read sequentially from first to last, and written by adding new records to the end of all the existing records. The basic definition of the sequential file begins:

```
OBJECT seqfile(record_type : TYPE, initial_size : ALENGTH <- 0)
  VAR first_record, last_record : REF record_type
   :
  first_record <- last_record <- NIL
END OBJECT
```

The OBJECT "seqfile" accepts a parameter that is the type of the records that the file can contain. This allows a single definition of "seqfile" to accommodate different types of records. As well, the initial size of the file can be specified as a multiple of the number of the records contained in the file. If no value is specified the parameter is initialized to zero. (To simplify most of the subsequent discussion the default action is used.)

The two pointer variables declared are pointers to the first and last records of "seqfile", respectively. Hence, these two pointers are pointers relative to the DS for the object. The "first_record" pointer is used to locate the first record for starting a sequential read of the file. The "last_record" pointer is necessary for detecting the end of file. These two pointers are local variables within the object. Conventionally, information about the size and structure of the file (called <u>control information</u>) has been separate from the file, and the file contains only the data records. I do not distinguish these two types of data, and hence information such as "first_record" and "last_record", which used to be considered "control information", is part of the object data. Lastly, there is the initialization of the record pointers, "first_record" and "last_record", which occurs when an instance of a "seqfile" is created.

2.4.2 Object Instantiation

A user declares a sequential file in the following manner:

```
VAR f : seqfile(INT)
```

The variable "f" is an instance of the OBJECT definition "seqfile" that can contain records that are each a single integer value. Instances of an object, such as "f", behave like instances of other abstract types in the programming language. This implies that the duration of the file depends on the duration of the block in which the file is declared. Thus, to have files persist for a particular period of time, it is necessary to declare the file in a memory that will persist for the required time. How such memories are created is discussed in detail in chapter 3. In the following examples, files can be declared in any scope in the programming language, and used in that scope. Thus, to use a temporary file for a program the file is declared as a local variable within a procedure and used during the execution of the procedure; it is automatically deleted at the end of the procedure as are other local variables.

2.5 OBJECT ACCESS

If an object were treated the same as a class, the definition of "seqfile" would include routines to manipulate the internal data items (i.e. read and write records). This in turn would imply that the file itself would always be accessible because statements such as:

```
f.write(12)
i <- f.read()
```

could appear anywhere that "f" was visible. Hence, the file would have to be accessible throughout its entire scope because the execution of accesses to the file could not be anticipated from the static structure of the program. While this might be acceptable for temporary files in procedures, it is not acceptable for files generally. One of the objectives was to make file access explicit and transient as it is in other languages using the equivalent of OPEN and CLOSE.

An alternative to having the file accessible continuously is to have each access make the file accessible, and then make the file inaccessible. As it will be shown, making a file accessible is a complex and relatively expensive operation. Hence, it is not practical to have individual calls such as "f.write" make the object accessible unless the hardware provides some support for this, such as in Multics. Even if the hardware provides this assistance, calls such as this that directly access the object are undesirable because of problems with concurrent access.

Thus, to make access transient and of a specified duration, the user is required to explicitly state when a file is to be accessed. This is accomplished by defining a class internal to the OBJECT definition, an instance of which the user declares when he wishes to access the file. This internal class contains the access components for the file, and hence is called the <u>access class</u>. The scope of an access class instance defines the duration that the object is accessible. Hence, the property that makes an access class different from an ordinary class is that the file remains accessible during the existence of the access class. As a result, references to routines in an access class do not have to make the file accessible. The execution of the declaration and the implicit release of an instance of an access class corresponds to OPENing and CLOSEing a file in other systems. However, the class is more restrictive, as the access class is associated with a particular scope whereas OPEN and CLOSE need not be; but this is not a serious restriction and, in fact, it is potential a positive feature.

2.5.1 Access Class for an Object

The access class is a class except for the implicit results it causes (i.e. making an object accessible) when an instance of it is created. The following is the basic definition of the access class definition for "seqfile". (The vertical line on the left margin of this and following examples indicates that new material has been added to a previously introduced example. I hope that this will allow the reader to save time by skimming the non-marked portion and reading only the newly marked portion in detail.)

```
OBJECT seqfile(record_type : TYPE, initial_size : ALENGTH <- 0)
  VAR first_record, last_record : REF record_type

  ACCESS seqacc
    VAR current_record : REF record_type
    ESCAPE end_of_file

    PROC read() RETURNS r : REF record_type
      :
    PROC reset()
      :
    PROC write(r : record_type)
      :
    PROC update(r : record_type)
      :
    PROC recreate()
      :

    current_record <- NIL
  END ACCESS

  first_record <- last_record <- NIL
END OBJECT
```

All the components necessary to access a "seqfile" are placed in the access class definition. This is necessary because these components are usable only after the file is made accessible by instantiating "seqacc". The pointer "current_record" is necessary to point at the current record being read from the file. When a read is attempted at the end of the file, the escape "end_of_file" is signaled.

To allow concurrent access, each accessor must have his own variables that contain information about the state of his access to the file. In the case of "seqfile" each user of a particular file must have his own current record pointer, and his own end of file escape. This is provided as each instance of "seqacc" creates a new "current_record" variable and "end_of_file" escape. Thus, the access class provides an essential capability needed for concurrent access; in addition, serialization of access to routines of the object will also be necessary.

The access routines provide sequential access to the records within the file. The following is a description of the basic function of each:

1. "read" - Reads one record starting at the first record of the file returning successive records with each invocation. The function "read" returns a pointer to the records because the records are potentially variable size. Returning a data item of unknown size on the stack (where most procedure results are returned) is difficult because the caller cannot allocate room on the stack for the return value before the call.
2. "reset" - Position "current_record" to the first record of the file to allow re-reading of the file.
3. "write" - Write a record after the last record in the file.
4. "update" - Change the contents of an existing record. The record to be updated must be the last record read. As well, because memory is managed to mimic sequential files as implemented on conventional systems, the new record must be the same size as the record to be updated. This restriction is not fundamental to OBJECT definitions but just to this particular example. A more complex storage management scheme could remove this restriction.

5. "recreate" - Delete all records from the file, and reset "first_record" and "last_record" to NIL.

Lastly, "current_record" is initialized to NIL whenever a "seqacc" is created.

Since "seqacc" is defined in "seqfile", normal static scope rules permit the routines in "seqacc" to refer to the variables "first_record" and "last_record" that are defined in the file. And since the dynamic creation of an access class causes the file to become accessible, the variables in the file are dynamically accessible.

2.5.2 Access Class Instantiation

When a block containing the following declaration is entered:

```
VAR pf : f.seqacc
```

the sequential file "f" becomes accessible. Here the name "f.seqacc" is used to refer to a type in "f's" definition, not to access the object. However, the creation of the instance of the subclass "seqacc" for "f" must make "f" accessible so that the components of "seqacc" can access the components of "seqfile".

The variable "pf" can now be used to access components associated with the access type. The scope of "pf" defines the portion of the program over which "f" is accessible. A user can now code:

```
pf.write(12)
i <- pf.read()
```

to read or write records in sequential file "f". For example, the following code shows the declaration of a local file variable and subsequent use of the file:

```
VAR f  : seqfile(INT)  ¢ declare file
VAR pf : f.seqacc      ¢ make file accessible
VAR i  : INT

FOR i <- 1 TO 100      ¢ write into the file
LOOP
  pf.write(i)
END LOOP

EXCEPTION              ¢ read the file
  LOOP
    i <- pf.read()     ¢ "read" signals escape "end_of_file"
    ¢ process the value of "i"
  END LOOP
ESCAPE pf.end_of_file
  ¢ end of file processing for "f"
END EXCEPTION
```

The EXCEPTION control structure is used to establish the catchers for signalled escapes. When escape "end_of_file" is signalled in "pf.read", it will be caught by the ESCAPE clause labelled "pf.end_of_file". Then its escape code is executed before the EXCEPTION control structure terminates. Thus, it is possible to associate a catcher with a statement but not an individual component of an expression or a call.

2.6 COMPLETE DEFINITION OF "SEQFILE"

Now that the skeleton of "seqfile" has been given and discussed, it is possible to give the details that complete that definition. The routines prefixed by "sma" and "sm" deal with the storage management level, and are discussed shortly.

```
OBJECT seqfile(record_type : TYPE, initial_size : ALENGTH <- 0)
  VAR first_record, last_record : REF record_type

  ACCESS seqacc
    VAR current_record : REF record_type
    ESCAPE end_of_file

    PROC read() RETURNS r : REF record_type
      IF current_record = last_record THEN
        SIGNAL end_of_file
      END IF
      current_record <- sma.next()
      r <- current_record
    END PROC

    PROC reset()
      current_record <- NIL
      sma.reset()
    END PROC

    PROC write(r : record_type)
      seqfile.write(r)
    END PROC

    PROC update(r : record_type)
      sma.update(r)
    END PROC
```

```
         PROC recreate()
           current_record <- NIL
           seqfile.recreate()
         END PROC

         current_record <- NIL
       END ACCESS

       PROC write(r : record_type)
         last_record <- sm.write(r)
         IF first_record = NIL THEN
           first_record <- last_record
         END IF
       END PROC

       PROC recreate()
         first_record <- last_record <- NIL
         sm.recreate()
       END PROC

       first_record <- last_record <- NIL
     END OBJECT
```

It should be noted that while the access class definition "seqacc" refers to the variables in OBJECT definition "seqfile", it does not modify them. Any modification of an object variable is done by calling routines in the OBJECT definition. This restriction of not altering the object directly stems from the requirement to allow concurrent access. Clearly, it is not possible to allow multiple users of a file to make simultaneous changes to the data items in the object. A method of serializing these changes is necessary to provide data integrity within the file. To provide such a facility the object is implemented as a monitor [HANSE73, HOARE74]. Hence, all changes to the object are done in a serial manner. Notice that the mechanism for controlling this serial access does not occur between the user's program and the object, but between the access class (which the user must create) and the object. Thus, any object code to implement the serialization is not part of every user's program but part of the object code in the access class.

(While this feature is a fundamental aspect of concurrent access, it does not provide the type of access that is desired for a "seqfile". A sequential file may allow simultaneous access to the file for reading, but only one user can be accessing the file for writing. This type of access control must be done through a different mechanism and will be discussed in chapter 5.)

In this example it is appropriate not to directly update variables in the memory because it would present problems during concurrent access, however this is not always the case. If a file was designed to allow only one user at a time to access it then there would be no problem if changes were made to variables directly in the object. Thus, the compiler does not restrict such changes; it is the responsibility of the file writer.

2.7 TYPE PARAMETERS TO AN OBJECT

In order to make storage management possible for type parameters, the following restriction is imposed: type parameters cannot be used to create instances of variables. Hence, a type parameter cannot be used to declare variables, or results of procedures within the OBJECT definition or in its components. However, pointers of the type parameter may be declared and this gives the user almost as much flexibility as being able to create instances of it. This simplifies storage management as it is not necessary to dynamically create an instance of a type parameter.

Type checking is still possible both within the OBJECT definition and for instances of the object. Within the OBJECT definition only pointers to the type are allowed, and consequently the operations that can be performed using the pointer are reference and assignment. As long as these operations occur between the same type parameters the operation is consistent. All usages of the file can be type checked because the type of the access variable "pf" includes this information; it is "seqfile(INT).seqacc". Hence, calls such as:

```
i <- pf.read()
```

can infer the result from "read" indirectly through the type parameter to "seqfile". The compiler knows routine "read" returns a pointer to type "record_type", that type "record_type" is the first parameter of "seqfile", and that the first argument of "seqfile" in the type of "pf" is "INT". Without the type parameter, the compiler could not make these inferences, and hence could not verify type consistency. The restriction on the type parameter allows the compiler to verify type consistency statically.

2.8 OBJECT AS AN ARGUMENT TO A PROCEDURE

Traditionally files are passed from operating system to programming language using many unusual techniques, few of which resemble the normal method of passing information from one program to another in a programming language. When files are embedded in the programming language, they must be communicated to appropriate routines either by passing them via the argument-parameter mechanism supported by the programming language, or through the block structure as a global variable. Thus, passing files as arguments to procedures is one important way of using files in the programming system.

2.8.1 Formal Declaration of an Object Parameter

Normally, the formal declaration of an object parameter must include all of the type parameters in the object parameter list. For files, this corresponds to specifying both the type of the object that defines the file and the type of the records contained in the object. For example, to pass the sequential file "f" defined earlier to a procedure, the parameter declaration would be:

```
PROC p(g : seqfile(INT))
```

As a further extension of this syntax, a type such as INT above, could be declared generically by making it a subparameter, as in:

```
PROC p(g : seqfile(record_type : TYPE))
```

This would allow routine "p" to accept any "seqfile" as a parameter.

While it is possible to have a type subparameter as part of the object parameter declaration, only pointers to this subparameter can be declared. This restricts the type of operations allowed, but still allows some simple generic file routines to be written. For example, the following program will take any "seqfile" containing records of type "record_type" and copy them to another "seqfile" containing records of type "record_type".

```
PROC copy(g : seqfile(record_type : TYPE), f : seqfile(record_type))
  VAR pg : g.seqacc, pf : f.seqacc
  VAR r  : REF record_type

  pf.recreate()
  EXCEPTION
    LOOP
      r <- pg.read
      pf.write(r)
    END LOOP
  ESCAPE pg.end_of_file
  END EXCEPTION
END PROC
```

The syntax of the parameter declaration indicates that the type of the records of "g" is the one used to indicate the type parameter of the other parameters. In general, this forces the compiler to perform extra work at the call site to check if the arguments are of the same type. The operation is more complex as the compiler must use the type parameter to check the argument types.

2.9 ACCESS CLASS AS AN ARGUMENT TO A PROCEDURE

Once an access variable is declared for a file, it can be used to access the file's contents in any block where the access variable is visible, or it can be passed as an argument to another routine. This routine can then use the access parameter to access its associated object without having to perform any further declarations.

2.9.1 Formal Declaration of an Access Class Parameter

The definition of an access class parameter must include not only the type of the access class but the type of the object, too. The type of access is directly dependent on the object which is accessed, and so the type of the access variable must include the type of the object. For example, to pass the access class "pf" defined earlier to a procedure, the parameter declaration would be:

```
PROC p(gf : seqfile(INT).seqacc)
```

Here the type of "gf" defines not only the access class "seqacc" but the type of the object which it accesses, including any type parameters. Note the prefixing of the access class definition name by an OBJECT definition name and not by an object instance name. This is allowed because it is specifying the type of a formal parameter for which no instance is created.

2.10 PROTOTYPES

Within any strongly typed programming language, there is a problem of having two similar types; and yet having to write two completely separate routines to handle each type, repeating even the routines that are in common. What is needed is a way to factor out these similarities into a prototype that can be used to define the routines that deal with those parts of the types that are the same. Along with these routines each individual type can have its own routines to deal with the differences. This is like class concatenation in Simula [DAHL72], FORM extention in ALPHARD [SHAW81], and class implementations in the ADA extentions of [APPEL84].

2.10.1 Defining a Prototype

The prototype then defines a formal description of all the components that make up a subset of the components for an actual type. For example, the following prototype represents the prototype for "seqfile":

```
OBJECT PROTOTYPE seqfiletype(record_type : TYPE, initial_size : ALENGTH)
  ACCESS PROTOTYPE seqacc
    ESCAPE end_of_file

    PROC PROTOTYPE read() RETURNS r : record_type
    PROC PROTOTYPE reset()
    PROC PROTOTYPE write(r : record_type)
    PROC PROTOTYPE update(r : record_type)
    PROC PROTOTYPE recreate()
  END ACCESS
END OBJECT
```

This prototype defines the fundamental external view that a user of "seqfile" can see and what a file writer must provide in his definition of the file so that the file behaves like a sequential file. Notice that the parameter variable names are specified and the same names must be used in the actual definition.

The prototype can include more than just formal specifications that must be conformed to. The prototype can contain actual variable declarations and routine definitions (i.e. executable statements). This is done by using a normal PROC or variable declaration (i.e. without the keyword PROTOTYPE). These actual definitions are then automatically entered into any OBJECT definition that is defined using the prototype and they cannot be changed. This allows the prototype to require inclusion of critical variables and routines in an OBJECT definition.

For example, the above prototype requires all objects based on "seqfiletype" to have routines "read", "reset", etc. As well, the ESCAPE in "seqacc" is forced into all OBJECT definitions based on "seqfiletype" and cannot be changed. Thus, the prototype provides great flexibility in structuring an actual type definition.

The prototype can now be used to define multiple OBJECT definitions to have the same fundamental type but with different internal structure and possibly external structure. This is different from the Ada package, first because this is a description for a type not an instance of a type, and secondly because an Ada package only allows one package body for each package specification.

2.10.2 Associating a Prototype with a Type

The association of a prototype and an actual type is done using the normal declaration syntax, as in:

```
OBJECT seqfile( ... ) : seqfiletype
```

This syntax implies that OBJECT definition "seqfile" has formal prototype "seqfiletype". It is then required to conform to the prototype definition, and hence, allows "seqfile" to be used in any context that allows a "seqfiletype". (This is done by copying the symbol table structure provided by the prototype to the symbol table for the OBJECT definition, and these symbol table entries are marked as unchangeable.) This allows "seqfile" to be used in any context that allows a "seqfiletype".

The user is then allowed to augment any part of the definition, but not to change any of the component information defined in the prototype. In this example, the user can add declarations and initialization code to the OBJECT definition, and to the access class definition, and its routines. The user may also add new routines in either contexts. For example, an index-sequential file can be defined in the following way:

```
OBJECT indexseq( ... ) : seqfiletype
```

in order to be able to be used wherever "seqfile" is allowed. Internally, "indexseq" must provide the necessary "seqfiletype" components along with the components for indexed access.

As well, a prototype can be associated with another prototype. This allows prototypes to act as the generic interface for actual types and allows these generic interfaces to be built up to form a hierarchical relationship of the function they perform. For example, a prototype may be defined for index-sequential file and this prototype will depend on that for a sequential file, as in:

```
OBJECT PROTOTYPE indexseqtype( ... ) : seqfiletype
```

Here all the the component information from "seqfiletype" forms the basis of prototype "indexseqtype" and this component information can be augmented in prototype "indexseqtype".

2.10.3 Using the Prototype

Along with the ability to provide a formal type for type definitions, the prototype can also be used in a formal declaration context. For example, the "copy" routine defined earlier could be defined as:

```
PROC copy(g:seqfiletype(record_type:TYPE), f:seqfiletype(record_type))
```

This allows any OBJECT definition of prototype "seqfiletype" to be passed to this routine (eg. two index-sequential files). The compiler ensures that only the components defined in "seqfiletype" can be used in conjunction with parameters "g" and "f".

2.10.4 Implementation Considerations

In implementing prototypes, the following situation must be considered:

```
PROC p(f : seqfiletype( ... ))
  VAR pf : f.seqacc
```

In this example, many different types of "seqfiles" can be passed to routine "p". Since each of these types can have their own superset versions of "seqacc", the declaration of "pf" requires special consideration. Here the type of "pf" depends on a PROTOTYPE, and as a result, the compiler cannot determine statically how much space to allocate for "pf". Thus, the compiler must take this declaration situation into consideration when generating object code that is dependent on a prototype. Normally, the object code to allocate an entity is compiled as part of the entry code to the block that contains the declaration. Clearly, this cannot occur in this situation.

To properly handle this declaration situation, the compiler must generate object code for entities dependent on a prototype so that storage allocation for local data items is performed at the beginning of the initialization code. In this way a type definition can be allocated in a block where the size of the type is not known.

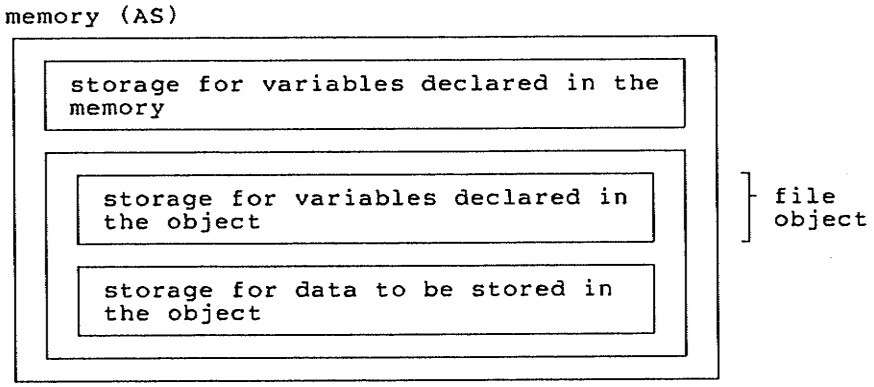
For the type to perform its own storage allocation for its local data items, it must allocate the storage on the heap. The stack cannot be easily used for allocating instances of unknown size, because the return address and the caller's registers would already appear at the top of the stack. Thus, the compiler must allocate all of the entities that are based on prototypes on the heap even if the size of the type can be determined, since the object code generated for an object may depend on whether it is allocated on the stack or the heap.

As well, in this situation there is the problem of accessing the object code for the particular type that is associated with the instance that is passed to "p". The compiler does not know where the object code for "f" and "pf" will be during compilation of "p". As a result, an implicit pointer to the object code must be passed along with the pointer to the instance. The calls to the routines in "pf" are then made through this pointer.

2.11 MEMORY

A MEMORY provides the description of the storage management level for an object. This is done by providing the components to manage the contiguous addressable string of bytes that form the underlying storage for an object or group of objects. It is through these components that the transition from the type "record_type" to the internal form of the record occurs for "seqfile". At this level, the MEMORY definition writer is allowed to manage the DS to provide the necessary high-level file operations specified in the object. Depending on the complexity of the file, storage management of the memory might be simple (as in the case of a sequential file) or it might be complex (as in the case of an index-sequential file). If records can be inserted or deleted anywhere in the file, it is the responsibility of the memory writer to allocate the necessary space to store the records that change size and are deleted. In the case of the index-sequential file both the index and the data records could be located together in the same memory, or they could be located in separate memories and managed separately. The choice is up to the file designer.

To perform storage management the memory writer must declare and manipulate pointer variables that point within the memory. Hence, explicit operations on pointer values are required; these operations are type-unsafe operations. As in the case of the object, variables declared in the MEMORY definition are allocated in the memory by the compiler. Thus, the variables in a memory might be laid out in the following way:


memory (AS)
storage for variables declared in the memory
storage for variables declared in the object
storage for data to be stored in the object
file object

Since the memory writer is allowed to arbitrarily alter pointers, he has access to all portions of the memory, including those allocated by the compiler. So, it is his responsibility not to alter the areas containing the variables for the memory and the file object.

These pointers are relative to the memory. This notion of a pointer being relative to a particular memory is an important point when there are multiple memories accessible. Within the object/memory the DS is accessible, and all pointers could be assumed to be relative to the DS. However, this would prevent pointers to items in the stack and/or heap. Thus, it is necessary as part of the declaration of a pointer to specify which memory the pointer is relative to. The syntax for this specification is the following:

REF(memory) type

Pointers to the heap will usually be the most common, and hence a REF without a memory specification is assumed to refer to the heap.

2.11.1 Relaxation of Type Consistency Rules

It is necessary during storage management to perform address calculations on virtual addresses. To do this requires a way of accessing the virtual address pointer underlying a typed REF. This address value is obtained by the attribute 'VALUE which de-references a typed pointer to it underlying address. In this way, calculations can be performed on virtual addresses using arithmetic operations, and 'VALUE can be used to assign the result to a pointer variable. Comparison operations can also be used. It is clear that in allowing any of these operations, there is the underlying assumption of a contiguously addressable memory.

As a consequence of allowing unrestricted pointer manipulation with all its associated consequences, the potential exists to generate a pointer value that does not point at the type of data item its declaration indicates. However, any virtual address, including invalid ones, can only be used to access the memory to which it is relative, and hence, cannot affect or be used to access any other memory. At this level, the programmer must share the responsibility that data values agree with their declared types. Type safety is possible through the joint action of both the compiler and the memory writer. Hence, only system programmers will likely be writing such code.

2.11.2 Memory Definition

An example definition of a memory will be given that manages the DS for a "seqfile" OBJECT definition. The routine prefixed by "dc" deals with the next level of storage management and is discussed shortly. The basic definition of the memory begins:

```
MEMORY seqmem(record_type : TYPE, initial_size : ALENGTH)
  TYPE encoded_address = 0..1

  RECORD internal_record
    length : POSINT,
    data : record_type
  END RECORD

  VAR initial_address, end_address : REF(seqmem) internal_record
  VAR extend_address : ADDRESS
  ESCAPE memory_overflow

  ACCESS seqmemacc
    :
  PROC allocate(s : POSINT) RETURNS a : encoded_address
    :
  PROC free(p : encoded_address)
    :

  initial_address'VALUE <- end_address'VALUE <- seqmem'SIZE
  extend_address <- dc.lastbyte()
END MEMORY
```

The MEMORY definition accepts a type parameter that is the type of the records that the file can contain. As in an OBJECT definition, the type "record_type" can only be used to declare pointers to records of that type within the MEMORY definition. Thus, a single definition of a sequential memory can accommodate different types of records. As well, the initial size for the memory is passed down, but in this case it is specified as the number of bytes of storage to be allocated not as a number of logical records.

The type "encoded_address" is used in addressing the object in the memory. The specification of this type, which represents addresses within the memory, is discussed in chapter 4. The structure "internal_record" describes the internal structure of each record for the "seqfile". In this example, the records are stored as varying length entities, with the length at the beginning of the record data structure. Other techniques, such as having a single length field and all records having the same length, or linking the records together could also be used. This latter technique could allow facilities at the user level not normally associated with a sequential file, such as updating a record with one of differing size or inserting and deleting records anywhere in the file, but this would complicate the example needlessly.

In the user's program, where the type of the record is known, so is the length of the record. However, at the storage management level the type of the record is unknown and the length of a record must be maintained explicitly. At the storage management level, the only operations on a variable whose type is a parameter is requesting its length and assigning it to another instance of that type, and this incidentally requires the length of the value. To allow these operations, the length of the instance must be passed to/from the routines in the object. This length is included with each parameter whose type is a parameter as well as with each REF to a type parameter. This implicit length parameter is accessible via the attribute 'SIZE. Inside the memory, the length of each record can be stored explicitly at the beginning of each record "internal_record" in variable "length". The data portion of "internal_record" contains the logical record.

The pointers "initial_address" and "end_address" are like "first_record" and "last_record" as they contain pointers that delimit the records in the file, and are local variables within the memory and hence are part of the local data of the memory. The pointer "initial_address" contains the virtual address of the first record of the file. The pointer "end_address" points at the first available byte after the last record in the file.

The pointer "extend_address" points at the physical end of the memory. This address is not necessarily the same as "end_address" because space for the memory is allocated in fixed sized blocks as will be seen later. Adding new records to the file advances "end_address" until it exceeds "extend_address", at which point the memory must be extended. Since it is not within the allocated part of the memory, "extend_address" is never used to refer to data within the memory. Thus, the type of "extend_address" is ADDRESS, where ADDRESS is a builtin type that describes an untyped virtual memory address (i.e. a 16, or 24, or 32 bit address).

The access class definition "seqmemacc" defines the components that allow access to the records in the memory. A detailed description of "seqmemacc" will be presented in a following section. Finally, the pointers "initial_address" and "end_address" are initialized to the first available byte in the memory after the storage for the local variables in "seqmem" (initial_address, end_address, extend_address).

This displacement from the beginning of the memory can be calculated at compile time, and is obtained by using the SIZE attribute for "seqmem".

The procedure definitions "allocate" and "free" must be provided for all memories. Procedure "allocate" allocates storage for each object within a memory and procedure "free" deallocates storage for each object within a memory. In the case of a sequential file there will be only one object "seqfile" in each "seqmem". However, in more complex situations there may be multiple objects in a single memory. The compiler can calculate the size of each object allocated in a memory at compile time or at execution time and make the appropriate call to "allocate" to create the storage for the object. For example, a memory for the stack and the heap have routines "allocate" and "free" to allocate and free space for variables. (However, because of the intimate relationship between the stack and the heap with the compiler and because of the desire for efficiency, particular properties of the stack and the heap may be “hard-coded” into the compiler itself.) Finally, the pointer "extend_address" is initialized to point at the byte allocated after the last byte in the memory, by calling a routine at the next level in the storage definition.

2.11.3 Memory Instantiation

A file designer will likely wish to have the memory for a "seqfile" invisible to the actual user. While a "seqfile" must be created in a memory, the user should not be concerned with this detail of storage management if it is not necessary. If the definition of "seqfile" were nested inside that of "seqmem", this nested structure would force a user to first create an instance of the memory to contain the file, as in:

```
VAR mem : seqmem
```

and then create an instance of "seqfile" within "seqmem", as in:

```
VAR f : mem.seqfile(INT)
```

This is undesirable for several reasons. First, the user should not have to be aware of nor have access to irrelevant details about storage management such as the creation of the containing memory since there is only one "seqfile" per "seqmem". Secondly, "seqmem" is not defined so that the memory can contain multiple sequential files, and yet there is nothing to preclude the user from entering another declaration, such as:

```
VAR f2 : mem.seqfile(INT)
```

Lastly, there is a problem of passing the type of the records and the initial size of the file from the file to the memory.

2.11.4 CREATES Clause

To circumvent these problems a new programming language clause is introduced. The CREATES clause allows the specification of dependence without requiring static nesting. For example,

```
OBJECT seqfile(record_type : TYPE, initial_size : ALENGTH <- 0)
       CREATES sm : seqmem(record_type, record_type'SIZE * initial_size)
```

states that the creation (declaration) of a "seqfile" implicitly creates a "seqmem" first to contain it. Thus, "seqfile" is not statically nested inside "seqmem". There are two points to note here. First, the memory, which contains the object, must be created first. Secondly, it is here that the "initial_size" request is translated from a number of records to a number of bytes.

Thus, a user's declaration of a "seqfile" does not contain irrelevant details about storage management. Since the user does not create the memory explicitly, it is anonymous to the user, and hence it is not possible to allocate multiple objects in a memory. The actual instantiation for a "seqfile":

```
VAR f : seqfile(INT)
```

occurs in the following manner. First, there is allocation of a "seqmem" memory to contain the sequential file "f". This creates space for the local variables in the memory, and initializations are performed on these variables. Only then can an instance of "seqfile" be allocated within "seqmem". This is done by calling the memory routine "allocate" requesting the necessary storage for a "seqfile". After this allocation in the memory, initialization of the object variables can be performed.

While the name of the containing memory is anonymous to the user, it must be accessible inside the "seqfile". Since "seqfile" is not nested in "seqmem", the scope rules cannot be used to allow unqualified access. Hence, qualification must be used in accessing routines in the MEMORY definition.

The CREATES clause causes storage allocation to be done in the same way as for class concatenation in Simula, however, in Simula the subclass can access all the variables in the containing class without qualification. This is not true for an object that CREATES a memory, because the OBJECT definition is not contained inside the MEMORY definition.

Since the memory in which "seqfile" was allocated was not previously specified in its definition, none of the REF variables in it could specify a memory (as they should have). Now the name "sm" can be used for this purpose. Since "first_record", "last_record", and "current_record" are all relative to "sm", they would be declared with type "REF(sm) record_type".

2.12 MEMORY ACCESS

Like an object, a memory is not directly accessible. A memory is only accessible for the duration of an instance of its access class. This is done to make access to the memory transient and of a specified duration. As well, the access class is desirable from a programmatic standpoint as it allows a memory writer to know and take any appropriate action for each access to the memory. (This is particularly useful when there are multiple objects in a memory.)

2.12.1 Access Class for a Memory

An example access class definition for MEMORY definition "seqmem" is:

```
MEMORY seqmem(record_type : TYPE, initial_size : ALENGTH)
  TYPE encoded_address = 0..1

  RECORD internal_record :
    length : POSINT
    data : record_type
  END RECORD

  VAR initial_address, end_address : REF(seqmem) internal_record
  VAR extend_address : ADDRESS
  ESCAPE memory_overflow
```

```
 ACCESS seqmemacc
  VAR current_address : REF(seqmem) internal_record
  ESCAPE update_length_error

  PROC update(r : record_type)
   :
  PROC next() RETURNS n : REF(seqmem) record_type
   :
  PROC reset()
   :
  PROC recreate()
   :

  current_address <- NIL
 TERMINATE
  free_record(end_address)
 END ACCESS

 PROC write(r : record_type) RETURNS w : REF(seqmem) record_type
  :
 PROC update(c : REF(seqmem) internal_record, r : record_type)
  :
 PROC recreate()
  :
 PROC allocate_record(s : POSINT) RETURNS a : ADDRESS
  :
 PROC free_record(p : ADDRESS)
  :
 PROC allocate(s : POSINT) RETURNS a : encoded_address
  :
 PROC free(p : encoded_address)
  :
 initial_address'VALUE <- end_address'VALUE <- seqmem'SIZE
 extend_address <- dc.lastbyte()
END MEMORY
```

All the components necessary to access the memory are placed in the access class definition. These routines perform only read access to the records that are stored in the memory. This is the same situation as for an object, that is to say, the access class does not assign directly into the entity it is accessing. All the components necessary to modify the memory are located in the MEMORY definition. The memory behaves as a monitor as does the object. Thus, calls to these routines from the memory access class will be handled as monitor calls. However, some of these components (as will be seen for "allocate_record") are not called from the access class. And, since "seqmem" is not directly accessible by the user, its components cannot be called by him. Thus, the compiler can determine that a call to "allocate_record" from, say, "write" need not be a monitor call.

The pointer "current_address" points at the internal record being read from the file. The exception "update_length_error" is signalled if a user attempts to update a non-existent record or replace a record with one that is longer than the one in the file.

A description of the routines in the access class definition are:

1. "update" - Assigns the new version of a record to the old one by calling a memory routine to change the record. A check is performed to see if the new record has the same length as the old. This check is performed at the storage management level because it depends on storage size.
2. "next" - This routine advances the "current_address" pointer by adding the length of the current record to the virtual address in the pointer "current_address". The address of the data portion of the internal record is returned.
3. "reset" - This routine resets "current_address" pointer to the beginning of the file.
4. "recreate" - The records are logically deleted by calling a memory routine to set "end_address" to "initial_address". The physical space for the records is not released until the access is terminated. Thus, if new records are written into the file after a "recreate", new storage may not have to allocated until the old storage is re-used. If all the storage is not re-used any excess will be released at access termination.

A description of the additional routines in the MEMORY definition are:

1. "write" - Allocates space at the end of the memory for the new record, and copies in the new record and its length. Write returns the address of the data portion of the newly created record.
2. "update" - Replaces an old record with a record of the same size.
3. "recreate" - Resets the end of record pointer to the beginning of the file.
4. "allocate_record" - Allocates space for the records in the file. While this routine allocates space in the memory like "allocate", it is called only by the above routines and not by implicit calls generated for declarations. Two allocation routines are necessary because they return different types of results and perform somewhat different storage management operations (i.e. one deals with objects, the other deals with records).

5. "free_record" - Frees space for the records in the file. This routine is different from "free" for the same reasons that "allocate_record" is different from "allocate".

Finally, when the memory access class is created, "current_address" is initialized to NIL.

Also, notice that some of the routines in the memory are called directly from the object circumventing the memory access class. This direct access is necessary because calls must be made from the object routines to the memory routines, but these calls cannot go through the memory access class because there are potentially multiple access classes. This direct access from object to memory is not a problem because the memory behaves as a monitor and hence serializes all accesses to itself. Thus, even if there are multiple users accessing an object and multiple objects in a memory, all modifications to the memory are serialized.

In the case of "seqfile" the object creates a new anonymous memory which is not accessed except by "seqfile". This is known by the compiler because the CREATES clause is used to establish the relationship between the object and the memory. In this situation it is unnecessary to have both the memory and the object behave as monitors. It is sufficient for the object to perform the serialization to ensure that concurrent access is performed properly.

2.12.2 <u>Access Class Instantiation</u>

A memory access class is created implicitly through the CREATES clause, as in:

```
ACCESS seqacc CREATES sma : sm.seqmemacc
```

This form for creating an access class must be used when an object uses the CREATES clause to specify its containing memory. This causes the access class for the memory to be created before "seqacc".

2.13 COMPLETE DEFINITION OF "SEQMEM"

The following is a complete definition of "seqmem".

```
MEMORY seqmem(record_type : TYPE, initial_size : ALENGTH)
  TYPE encoded_address = 0..1

  RECORD internal_record
    length : POSINT,
    data : record_type
  END RECORD

  VAR initial_address, end_address : REF(seqmem) internal_record
  VAR extend_address : ADDRESS
  ESCAPE memory_overflow

  ACCESS seqmemacc
    VAR current_address : REF(seqmem) internal_record
    ESCAPE update_length_error

    PROC update(r : record_type)
      IF current_address = NIL OR current_address.length ¬= r'SIZE THEN
        SIGNAL update_length_error
      END IF
      update(current_address, r)
    END PROC

    PROC next() RETURNS n : REF(seqmem) record_type
      IF current_address = NIL THEN
        current_address <- initial_address
      ELSE
        current_address'VALUE +<- current_address.length +
                                  internal_record'SIZE
      END IF
      n <- REF record_type(current_address.length,
                           current_address.data)
    END PROC

    PROC reset()
      current_address <- initial_address
    END PROC

    PROC recreate()
      current_address <- initial_address
      recreate()
    END PROC

    current_address <- NIL
  TERMINATE
    free_record(end_address)
  END ACCESS

  PROC write(r : record_type) RETURNS w : REF(seqmem) record_type
    VAR t : REF(seqmem) internal_record

    t <- allocate_record(r'SIZE + internal_record'SIZE)
    t.length <- r'SIZE
    t.data <- r
    w <- REF record_type(t.length, t.data)
  END PROC

  PROC update(c : REF(seqmem) internal_record, r : record_type)
    c.data <- r
  END PROC

  PROC recreate()
    end_address <- initial_address
  END PROC
```

```
 PROC allocate_record(s : POSINT) RETURNS a : ADDRESS
  IF end_address'VALUE + s > extend_address'VALUE THEN
   extend_address'VALUE <- dc.extend(end_address'VALUE + s)
   IF extend_address'VALUE > address'MAX THEN
    SIGNAL memory_overflow
   END IF
  END IF
  a <- end_address'VALUE
  end_address'VALUE +<- s
 END PROC

 PROC free_record(p : ADDRESS)
  extend_address <- dc.reduce(p)
 END PROC

 PROC allocate(s : POSINT) RETURNS a : encoded_address
  IF end_address'VALUE + s > extend_address'VALUE THEN
   extend_address'VALUE <- dc.extend(end_address'VALUE + s)
  END IF
  a <- 1
  end_address'VALUE +<- s
 END PROC

 PROC free(p : encoded_address)
 END PROC

 initial_address'VALUE <- end_address'VALUE <- seqmem'SIZE
 extend_address <- dc.lastbyte()
END MEMORY
```

It is possible to treat either the first or last record as a special case for "current_address". If the last record is treated as the special case then "current_address" would not start at NIL but at "initial_address", and it would always point at the next record to be read. However, this presents a problem in updating records as "current_address" would not be pointing at the record just read. For this reason, the first record is handled as the special case.

2.14 INTER-MEMORY REFERENCES

Routines that return pointer values from the object/memory introduces another level of complexity. Outside the memory, a pointer cannot be used on its own. Notice for example, that "read" in "seqfile" does not return a record but a pointer to the record. Usually this pointer is de-referenced immediately to copy the record into one of the user's variables. Here, the compiler can infer the memory that the pointer from "read" is relative to, and copy the data from that memory into the variable in the user's program (i.e. in another memory). This inference is possible because "read" is qualified by the access variable whose type specifies the memory "f".

However, if the user wished to retain the pointer for subsequent references to the record in the memory, then the memory must be specified along with the pointer so that when it is used the compiler knows which memory the pointer is relative to. In this way, the correct object code can be generated to access the necessary data in a different memory. This specification is made using the access class for the object/memory, as follows:

```
VAR pf : f.seqacc
    p  : REF(pf^sm) INT

p <- pf.read()
```

The notation "pf^sm", used in the declaration, indicates that it is declared to be relative to the memory for sequential file "f" through the access class "pf". Outside of the memory for "f", the memory is only accessible indirectly through the access class for the object. Because of the scope rules, a pointer such as "p" may only be declared within a block containing "pf", and hence is guaranteed to have shorter duration than "pf"; thus no dangling pointers can occur.

When "p" is de-referenced, as in:

```
i <- p
```

the compiler will have to generate object code to access the memory for "f" via "pf" and copy the necessary data from "f". To do this the compiler uses class "pf" to get access to the memory of the object.

2.15 BACKING-STORE

While "seqmem" performs storage management for "seqfile", it does so based on the fact that there is some underlying storage that implements the contiguously addressable memory. Ultimately the underlying memory must be an actual (physical) memory (i.e. disk pack, tape reel, etc.), and this actual memory is call the backing-store. Normally, the user is not (and should not have to be) concerned with the backing-stores and/or their configuration and properties. The user only deals with memories that contain information that he created or intends to use. These memories that contain user objects are called volumes. In this system, the memories that contain the user's information and their configuration do not have to correspond directly with the backing-store configuration as is done in conventional operating systems. A volume may correspond with a backing-store memory or it may be only part of such a memory or it may be composed of several such memories. As well, a volume does not have to store object memories in a contiguous manner. It need only give the impression to the storage user that the object's memory is contiguous. This is done by mapping the addresses from the overlying memory to the non-contiguous volume.

Maintaining the object memory in a non-contiguous manner is desirable because it allows the greatest flexibility in managing storage. Attempting to maintain contiguous areas for entities that can dynamically expand and reduce in size is very difficult. As was discovered with real memory storage management, dynamic allocation of non-contiguous pieces of memory and mapping them together to form a whole allows the greatest ease and flexibility for managing storage.

A volume is conceptually a memory for storing entities like "seqmem", which in turn are memories for storing records. Thus, there is a nesting of one memory within another memory. This latter memory can be further nested in another memory and so on, down to the actual backing-store. Any intermediate memory, while not essential, provides a means of logically subdividing and controlling storage of the actual backing-store.

2.16 CAPSULE

For each user object allocated in a volume (eg. a "seqfile" on a disk), an entity must be created to manage the non-contiguous pieces that form the object's memory. This entity is called a capsule and is simply an object allocated in a memory associated with the volume. However, in this case there will usually be multiple objects allocated in this memory, in contrast to "seqmem", which contains only one object.

2.16.1 Capsule Definition

The following is an example definition for a capsule that allocates storage on a volume with direct access properties, for example, a disk.

```
OBJECT dircap(initial_size : ALENGTH)
  CONST secondary_proportion : REAL <- 0.2

  VAR extent_size : ADDRESS    ¢ current size of the file
  VAR secondary   : ALENGTH    ¢ size of secondary extent
  VAR noext   : 1..16          ¢ number of extents
  VAR extent  : [16] RECORD    ¢ extent table for managing file pieces
      extent_length : ALENGTH,     ¢ length of the extent
      extent_address  : DISKADR    ¢ disk address of the extent
    END RECORD
  ESCAPE initial_allocation_error, extention_overflow

  ACCESS diracc
    :
  PROC lastbyte RETURNS l : ADDRESS
    :
  PROC extend(end_address : ADDRESS) RETURNS e : ADDRESS
    :
  PROC reduce(end_address : ADDRESS) RETURNS r : ADDRESS
    :

  noext <- 1
  secondary <- INT(CEIL(initial_size * secondary_proportion / pagesize))
                                                            * pagesize

  ¢ allocate primary space

  WITH extent[1] DO
    (extent_length,extent_address)<-vtocmem.allocate_block(MAX(pagesize,
                                                          initial_size))
    IF extent_length < initial_size THEN
      SIGNAL initial_allocation_error
    END IF
    extent_size'VALUE <- extent_length - 1
  END WITH
TERMINATE

  ¢ release all space before capsule is terminated

  FOR i <- 1 TO noext
  LOOP
    vtoc.free_block(extent[i].extent_address)
  END LOOP
END OBJECT
```

This object manages the fragments of the overlying memory as 16 separate areas or extents. The first extent is allocated on the basis of the initial size specification and the remaining extents are allocated based on a percentage of the initial request or a minimum amount related to the type of volume. This method of managing the space is done to illustrate that the object-memory mechanism is general enough to mimic the storage management of disk space done in the MVS operating system by IBM [IBM81]. Many other techniques could be used to manage the space, limited only by the operations available on the backing-store underlying the volume.

The volume in which the capsule allocates storage is assumed to be divided into discrete blocks of size "pagesize". A "DISKADR" is the address of one of these blocks. These addresses are used by the volume level routines to access the volume and are not altered by the capsule. The routines prefixed by "vtocmem" refer to the underlying memory which contains the capsule.

2.16.2 Capsule Instantiation

While a capsule could be created explicitly it will usually be created implicitly by the CREATES clause on the definition of a memory. This is because the capsule is strictly a storage management phenomenon and not relevant to the user. For example,

```
MEMORY seqmem(record_type : TYPE, initial_size : ALENGTH)
       CREATES  dc : dircap(initial_size + seqmem'SIZE)
```

causes creation of a new capsule to manage the space for "seqmem". The initial size of the space is specified as the sum of the user requested space plus the amount of space required for "seqmem". The initial amount must be available or else the capsule signals an exception. This is because space for the local variables of "seqmem" must be obtained so that they can be initialized. However, while this minimum must be achieved, more space may be allocated because allocation is done as an integral number of blocks. Since the volume is subdivided into fixed size blocks, allocation is done to the nearest block. The exact amount allocated initially is returned by the function "lastbyte". This amount is specified as the address of the last byte actually allocated by the capsule.

2.17 CAPSULE ACCESS

The capsule is just an object in a memory and hence is not directly accessible all of the time; thus, it too must have an access class.

2.17.1 Access Class for a Capsule

An example access class definition for the above capsule is:

```
  ACCESS diracc
|   ESCAPE readerr, writerr
|
|   PROC read(adr : ADDRESS, size : ALENGTH, vmadr : ADDRESS)
|     :
|   PROC write(adr : ADDRESS, size : ALENGTH, vmadr : ADDRESS)
|     :
| END ACCESS
```

The access class definition for capsule "dircap" provides the two routines "read" and "write". These routines transfer information between the volume and "seqmem's" memory and are not called in "seqmemacc" ("seqmem's" access class) because "seqmem" is read and written by paging. However they are needed in general, and could be used by "seqmemacc" if a different storage management technique were used in "seqmem".

Both routines are passed the starting address of the data to be read/written, the number of bytes to be transferred, and the location where the data is to be placed in "seqmem's" memory. The routines translate the starting address into a volume address via the extent table. This disk address is then used to perform the desired transfer of data.

2.17.2 Access Class Instantiation

When a capsule is created implicitly by a CREATES clause its ACCESS class will usually be created by a CREATES clause too. When the capsule access class is specified in a CREATES clause, the access class for the capsule is created automatically as part of the accessing of the memory that the capsule is backing. This is accomplished by using a CREATES clause on the access class for the memory, as in:

```
ACCESS seqmemacc CREATES dca : dc.diracc
```

The variable "dca" is then created during the access of the overlying memory.

2.18 COMPLETE DEFINITION OF A CAPSULE

The following is a complete definition of "dircap". Routines prefixed by "vtoca" and "vtocmem" deal with the memory that contains the capsule and are discussed in the next section.

```
OBJECT dircap(initial_size : ALENGTH)
  CONST secondary_proportion : REAL <- 0.2

  VAR extent_size : ADDRESS     ¢ current size of the file
  VAR secondary   : ALENGTH     ¢ size of secondary extent
  VAR noext     : 1..16         ¢ number of extents
  VAR extent    : [16] RECORD   ¢ extent table for managing file pieces
      extent_length : ALENGTH,      ¢ length of the extent
      extent_address  : DISKADR     ¢ disk address of the extent
    END RECORD
  ESCAPE initial_allocation_error, extention_overflow
```

```
ACCESS diracc CREATES vtoca : vtocmemacc
  ESCAPE readerr, writerr

  PROC read(adr : ADDRESS, size : ALENGTH, vmadr : ADDRESS)
    VAR vadr : ADDRESS

    ¢ look up "adr" in extent table

    vadr'VALUE <- extent[1].extent_length - 1
    FOR i <- 1 TO noext FINI SIGNAL readerr
    LOOP
    UNTIL adr > vadr EXIT
      vadr'VALUE +<- extent[i].extent_length
    END LOOP

    vtoca.read(extent[i].extent_address, size, vmadr)
  END PROC

  PROC write(adr : ADDRESS, size : ALENGTH, vmadr : ADDRESS)
    VAR vadr : ADDRESS

    ¢ look up "adr" in extent table

    vadr'VALUE <- extent[1].extent_length - 1
    FOR i <- 1 TO noext FINI SIGNAL writerr
    LOOP
    UNTIL adr > vadr EXIT
      vadr'VALUE +<- extent[i].extent_length
    END LOOP

    vtoca.write(extent[i].extent_address, size, vmadr)
  END PROC
END ACCESS

PROC read(adr : ADDRESS, size : ALENGTH, rmadr : RMADDR)
  VAR vadr : ADDRESS

  ¢ look up "adr" in extent table

  vadr'VALUE <- extent[1].extent_length - 1
  FOR i <- 1 TO noext FINI SIGNAL readerr
  LOOP
  UNTIL adr > vadr EXIT
    vadr'VALUE +<- extent[i].extent_length
  END LOOP

  vtocmem.read(extent[i].extent_address, size, rmadr)
END PROC

PROC write(adr : ADDRESS, size : ALENGTH, rmadr : RMADDR)
  VAR vadr : ADDRESS

  ¢ look up "adr" in extent table

  vadr'VALUE <- extent[1].extent_length - 1
  FOR i <- 1 TO noext FINI SIGNAL writerr
  LOOP
  UNTIL adr > vadr EXIT
    vadr'VALUE +<- extent[i].extent_length
  END LOOP

  vtocmem.write(extent[i].extent_address, size, rmadr)
END PROC

PROC lastbyte RETURNS l : ADDRESS
  l <- extent_size
END PROC
```

```
  PROC extend(end_address : ADDRESS) RETURNS e : ADDRESS

    ¢ add storage up to the specified virtual address

    IF end_address > extent_size THEN
      LOOP
      UNTIL end_address < extent_size EXIT
        IF noext = 16 THEN SIGNAL extention_overflow END IF
        noext +<- 1
        WITH extent[noext] DO
          (extent_length, extent_address) <- vtocmem.allocate_block(
                                                          secondary)
          extent_size'VALUE +<- extent_length
        END WITH
      END LOOP
      e <- extent_size
    ELSE
      SIGNAL extention_overflow
    END IF
  END PROC

  PROC reduce(end_address : ADDRESS) RETURNS r : ADDRESS

    ¢ release storage until storage that contains
    ¢ data for specified virtual address

    VAR vadr : ADDRESS

    vadr'VALUE <- extent_size - 1
    FOR i <- noext DOWNTO 2   ¢ cannot release EXTENT[1] as it
    LOOP                      ¢ contains storage for local variables
      vadr'VALUE -<- extent[i].extent_length
    UNTIL end_address'VALUE > vadr'VALUE EXIT
      vtocmem.free_block(extent[i].extent_address)
    END LOOP
    r <- vadr
  END PROC

  noext <- 1
  secondary <- INT(CEIL(initial_size * secondary_proportion / pagesize))
                                                            * pagesize

  ¢ allocate primary space

  WITH extent[1] DO
    (extent_length,extent_address)<-vtocmem.allocate_block(MAX(pagesize,
                                                          initial_size))
    IF extent_length < initial_size THEN
      SIGNAL initial_allocation_error
    END IF
    extent_size'VALUE <- extent_length - 1
  END WITH
TERMINATE

  ¢ release all space before capsule is terminated

  FOR i <- 1 TO noext
  LOOP
    vtocmem.free_block(extent[i].extent_address)
  END LOOP
END OBJECT
```

The two routines "read" and "write" in the body of "dircap" are similar to the routines "read" and "write" in the access class. However, the two routines in "dircap" will be used to transfer information between the volume and real memory as a opposed to the overlying memory. These two routines will be used by the paging routines.

2.19 VOLUME TABLE OF CONTENTS

Currently, a volume usually contains more memory than can be addressed by the machine hardware. This means that the entire volume can never be made to look like a single contiguous memory via the paging mechanism, as is done for most files allocated on the volume. (This would also be true for files that are larger than the addressing capabilities of the machine.) This problem affects the addressing of the data for the capsule and the storage for the memory mapped by the capsule.

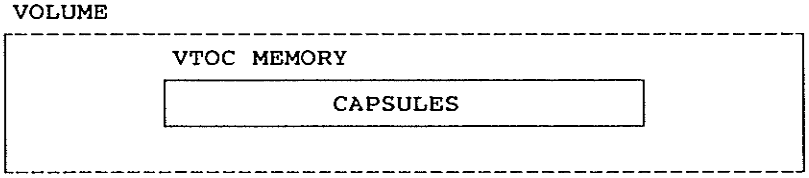
If a capsule is written like "seqfile" so that the local variables and the storage for its overlying memory are assumed to be in a contiguously addressable memory, then it would be necessary to be able to address the entire volume because the capsule data could be allocated anywhere on the volume. This cannot be done because the size of a memory addressable by the instructions of the machine will usually be much less than the size of a disk; in general, one cannot assume that it will be possible in the future for such memories to address the largest possible volume.

To circumvent this problem, the following two techniques could be used:

1. The capsule can assume that none of its data is directly addressable. Hence, the capsule would have to use reads and writes on the volume to access its local variables as well as the blocks of the overlying memory.
2. The capsule can assume that its data is directly addressable but not the blocks of the overlying memory. Since the amount of data is small, it could be addressed directly via paging; the larger overlying memory could be accessed by issuing reads and writes to the volume.

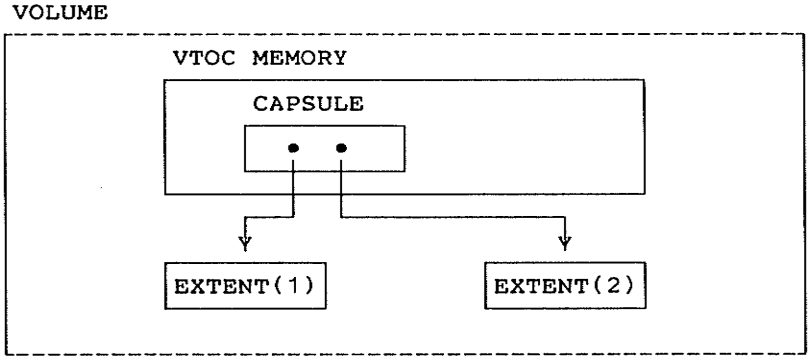
The main difference between these two approaches is that if the data is paged it is easier to write the code for the capsule. In any case, only one instance of the capsule data is used for access.

Rather than page each capsule separately, all the capsules can be allocated together into one memory on the volume, and this memory is then paged. This memory is called the volume table of contents (VTOC).



Hence, when it is necessary to allocate a new capsule, it is allocated in the VTOC memory by a call to "VTOC.allocate". Since a capsule's data is usually small, many hundreds of capsules can be allocated within the paged VTOC. Thus, when a capsule is being used, its data is directly accessible via the VTOC page tables. This allows the object code for the capsule to directly refer to the local variables of the capsule.

The storage for the overlying memory is still maintained in the volume outside the VTOC.



Each fragment of the overlying memory is in a portion of the volume that is not paged and hence must be accessed by reads and writes.

This same organization of storage could be used to construct a sequential file type that could store more data than is directly accessible by the machine hardware. Instead of maintaining the records and their length together in the directly addressable "seqmem", "seqmem" would maintain only the lengths and addresses of the records; the records themselves would be stored outside the directly addressable "seqmem" in the volume, and accessed by reads and writes on the capsule.

As long as the control information did not exceed the size of the addressable "seqmem", the data portion of the file could grow as large as the volume.

Since a volume has all the information needed to manage its space, it might be a separate process operating concurrently with other processes in the system. For example, if it were located on a disk drive that had its own processor this disk processor could perform storage management operations independently from the main processor (i.e. like a file server). The disk processor could be garbage collecting free space in the VTOC or in the block space and/or reorganizing blocks to amalgamate the blocks of a file as close together as possible to reduce access time. These operations are only possible if the volume has all the necessary information available to perform these operations.

To implement some of these ideas might require that each capsule provide routines that the disk processor could call to access information stored in the capsule, the amount of free space, or address of blocks allocated to the capsule, etc. These routines can be forced to appear in all capsule definitions via a prototype that all capsule definitions must conform to.

2.19.1 VTOC Memory Definition

The following is an outline of a MEMORY definition for a VTOC. Routines prefixed by "vol" and "vola" deal with the volume level and are discussed shortly.

```
MEMORY vtocmem(initial_size : ALENGTH)
  CONST MAXCAPS : INT <- 1000

  VAR vtocdir : [MAXCAPS] RECORD
      status : (exists, deleted)
      crdate : date
      length : POSINT
      adr   : ADDRESS
    END RECORD

  VAR initial_address, end_address, extend_address : ADDRESS

  ACCESS vtocmemacc
    PROC read(adr : DISKADR, size : ALENGTH, vmadr : ADDRESS)
      vola.read(adr, size, vmadr)
    END

    PROC write(adr : DISKADR, size : ALENGTH, vmadr : ADDRESS)
      vola.write(adr, size, vmadr)
    END PROC
  END ACCESS
```

```
   PROC read(adr : DISKADR, size : ALENGTH, rmadr : RMADDR)
     vol.read(adr, size, rmadr)
   END PROC

   PROC write(adr : DISKADR, size : ALENGTH, rmadr : RMADDR)
     vol.write(adr, size, rmadr)
   END PROC

   PROC allocate_block(s : POSINT) RETURNS length : POSINT, adr : DISKADR
     ¢ remove contiguous blocks from free list,
     ¢ and initialize blocks to zeros
   END PROC

   PROC free_block(s : DISKADR)
     ¢ keep track of contiguous free blocks
   END PROC

   PROC allocate(s : POSINT) RETURNS a : 1..MAXCAPS
     IF end_address'VALUE + s > extend_address THEN
       garbage()
     END IF

     FOR i <- 1 TO MAXCAPS FINI SIGNAL vtocdirovfl
     LOOP
     UNTIL vtocdir[i].status = deleted EXIT
     END LOOP

     WITH vtocdir[i] DO
       status <- exists
       crdate <- currentdate()
       length <- s
       adr <- end_address
     END WITH

     end_address'VALUE +<- s
     a <- i
   END PROC

   PROC free(i : 1..MAXCAPS)
     WITH vtocdir[i] DO
       status <- deleted
       crdate <- currentdate()
     END WITH
   END PROC

   PROC garbage
     ¢ move the capsules up in the vtoc,
     ¢ and adjust the vtoc directory pointers
   END PROC

   ¢ initialize the VTOC memory directory

   FOR i <- 1 TO MAXCAPS
   LOOP
     WITH vtocdir[i] DO
       status <- deleted
       crdate <- date(0)
     END WITH
   END LOOP

   initial_address'VALUE <- end_address'VALUE <- vtocmem'SIZE
   extend_address'VALUE <- vol.lastbyte()
 END MEMORY
```

The variable "vtocdir" is an array whose elements contain status information about capsules and the pointers to them. When a capsule is allocated, a search is made of the array to locate a free entry, and that entry is used along with the allocation of the space for the capsule's data items. When a capsule entry is freed, the table entry is marked as deleted and can subsequently be reused; the space for the capsule's data items is then eligible for garbage collection. The variables "initial_address" and "end_address" point at the beginning and end of the area from which capsules are allocated, respectively; while "extend_address" points at the last byte allocated in the memory (i.e. the memory size). Like the access class definition for "dircap", the access class definition for memory "vtocmem" provides the two routines "read" and "write". These routines transfer information to and from the volume to the capsule's overlying memory. However, the address of the block to be read or written is given as the address of a volume block (which was translated via the extent table).

The routines in "vtocmem" are similar to those in "seqmem", but the space managed by "allocate_block" and "free_block" lies outside "vtocmem", whereas, "allocate" and "free" allocate space inside "vtocmem". The two routines "read" and "write" transfer information to and from the volume to real memory. The routine "garbage" is called when the VTOC memory fills up in order to reclaim space from deleted capsules. (This routine is not defined in detail as it is not important to the definition of the file.)

2.19.2 VTOC Memory Instantiation

The VTOC memory is allocated explicitly by a declaration such as the following entered by the system administrator:

```
VAR vtoc1 : vtocmem(500K)
```

Thus, "vtoc1" is the actual memory for all capsules allocated on a particular volume.

2.19.3 VTOC Access Class Instantiation

The VTOC memory access class would probably be created explicitly by the operating system as part of its start up process. (This would be done for all VTOC memories on the system.) Hence, there would be only one access class for the VTOC memory even though there will be multiple accessors. This is in contrast to access to "seqfile" where each accessor creates three access classes, one for each of "seqfile", "seqmem", and "dircap". The access classes are only necessary if information peculiar to each user's access must be created. In this VTOC memory definition no such information is required.

2.20 VOLUME

Conceptually a volume is a memory implemented by some physical backing-store (but his need not be so). When a memory such as the VTOC is allocated on the volume, a data structure similar to a capsule is created for it, as is done when a "seqmem" is allocated in the VTOC. However, unlike a normal memory, the volume memory is never directly addressable because of its size. Hence, there will not be a MEMORY definition for the volume as there is for other memories. Therefore, the volume capsule that must be created for the VTOC on the volume is really the lowest-level programming language manifestation of the volume. This volume capsule then deals directly with the backing-store device to perform I/O operations.

Hence, the volume capsule provides the transition point from a volume address (eg. the 24 bit address of a 4K block) into a physical disk address (eg. a cylinder #, track #, record #). Thus, this operation is isolated in one routine. This would allow the volume capsule to physically position the VTOC in the middle of the disk (to optimize access), simply by mapping of the logical address into the proper physical address.

2.20.1 Volume Capsule Instantiation

A volume capsule is created when a VTOC memory is declared. This is specified in the definition of the VTOC memory by means of a CREATES clause, as in:

```
MEMORY vtocmem(initial_size : ALENGTH) CREATES vol : volcap
```

This implicitly declares an instance of a "volcap" for each instance of a "vtocmem", as in:

```
VAR vtoc1 : vtocmem(500K)
```

Here, since the volume capsule is anonymous, the VTOC is the only access to the volume. As well, in this case the access class for the volume is defined by means of a CREATES clause, as in:

```
ACCESS vtocmemacc CREATES vola : vol.volacc
```

In the case of a disk, the data for the "volcap" instance may be allocated at the beginning of the disk. As well, the initialization code for the "volcap" may initialize the remainder of the volume. The data for a "volcap" object is not paged and hence is only accessible by direct read and write operations. This data contains the information about the location of the VTOC memory for the volume.

If an installation wished to divide the disk into two logical volumes, each with its own VTOC memory, this could be done but it would be necessary to change the definition slightly. To handle this, a volume object would be created explicitly, as in:

```
VAR vol01 : volcap
```

and each VTOC would have to be declared within the volume, as in:

```
VAR vtoc1 : vol01.vtocmem(250K),
    vtoc2 : vol01.vtocmem(250K)
```

Each declaration would allocate another volume capsule at the beginning of the disk which would point to the new VTOC memory. The definitional structure to support this would require the VTOC definition to appear in the volume definition, as in:

```
OBJECT volcap ...
  :
  MEMORY vtocmem ...
```

2.21 DEVICES

The backing-store must ultimately be mounted on a particular device, such as a disk drive or spindle. When requests are made to access data on the volume, these requests are ultimately passed to the device which performs the operations. Each type of device will have an OBJECT definition and each actual device will be declared, as in:

```
VAR spindle01 : disk3350
```

To mount a volume on a device it is necessary to associate the two. In particular, this association must be made at the time the VTOC memory is declared so that the volume can be initialized. This association can be done in two ways:

1. The volume definition could appear inside of the device definition. This means that the definition of "disk3350" must contain a definition of "vtocmem". This would make the volume declaration look like:

   ```
   VAR vtoc1 : spindle01.vtocmem(500K)
   ```

   This form of the declaration implies that the volume is always mounted on "spindle01", which may be fine for unmountable volumes.
2. The volume could be passed as a parameter to the "vtocmem", as in:

   ```
   VAR vtoc1 : vtocmem(spindle01)
   ```

   This form of the declaration allows the association between volume and device, and yet does not permanently bind the two together. The address of the device would be stored in a variable in the volume. Mounts and dismounts must still be done by a program that exists outside the "vtocmem", and this might be done as:

   ```
   mount(vtoc1, spindle02)
   ```

   which causes the volume "vtoc1" to be moved from its current device to device "spindle02". The routine "mount" causes the physical dismount and mount of the desired volumes, and then calls a routine in "vtocmem" to update the variable that contains the address of the device on which the volume is mounted.

2.22 STRUCTURE OF BACKING-STORE

Since the VTOC is created independently of the objects allocated within it, it must be allocated first. As well, there will be multiple objects allocated in the VTOC. Hence, the basic definition of the VTOC will contain the definitions of the overlying memories and their associated capsules. For example, a general prototype for a VTOC may begin:

```
MEMORY PROTOTYPE dirmemtype
  OBJECT PROTOTYPE capsule1
    :
  OBJECT PROTOTYPE capsule2
    :
```

where the objects "capsule1, capsule2, ..." are the capsules for the different types of overlying objects stored in the VTOC memory.

This structure requires the definitions of the overlying objects to be defined in "dirmemtype" so that they can refer to the appropriate capsule definition, as in:

```
MEMORY PROTOTYPE dirmemtype
  OBJECT PROTOTYPE dircap( ... ) ...
    :
  MEMORY seqmem  ... CREATES dircap ...
    :
  OBJECT seqfile ... CREATES seqmem ...
    :
  other file types
```

This allows different implementations of "dircap" on different volumes; thus, different backing-stores (like a tape memory) can allow the definition of "seqfile" provided an appropriate "dircap" is defined. A MEMORY definition for this prototype must supply the necessary capsule definition(s) and the appropriate storage management to support the capsule(s) in the memory. Thus, the definition of "vtocmem" is of type "dirmemtype", as in:

```
MEMORY vtocmem(initial_size : ALENGTH) : dirmemtype
  :
  OBJECT dircap( ... ) ...
    :
```

This MEMORY definition automatically inherits the definitions of all objects in "dirmemtype" (eg. "seqfile", "indexseq", etc.).

Since the overlying memory, such as "seqmem", no longer determines the specific underlying volume to be used, it is now necessary for the user to specify the volume (possible through the VTOC) that is to contain the "seqfile". Thus, a declaration such as:

```
VAR f : seqfile(INT)
```

is really incomplete, as there is no indication of the underlying volume in which the file is to be created, and hence which capsule definition will be used. A complete declaration must make it possible for the system to determine the memory in which allocation is to take place; a user might specify this explicitly, as in:

```
VAR f : vtoc1.seqfile(INT)
```

Here "vtoc1" is the name of a VTOC memory of type "vtocmem" that will contain the file. Since "seqfile" indirectly creates a "dircap", the the capsule named "dircap" from the type "vtocmem" will be used. In chapter 3 it will be seen that there are situations where the underlying volume in which the file is to be created can be inferred and hence no explicit volume need be specified.

The capsule specified in the VTOC memory can then accept the calls from "seqmem" and perform the appropriate actions to implement them in the particular volume where the capsule is defined. For example, in the declarations:

```
VAR f1 : disk01.seqfile,
    f2 : drum01.seqfile,
    f3 : tape01.seqfile
```

the appropriate capsule definition in each type of volume would be used. Each of these capsules manages the discontiguous space for "seqfile" in a manner appropriate to the particular backing-store.

A more detailed description of the prototype "dircap" as given in "dirmemtype" is:

```
OBJECT PROTOTYPE dircap(initial_size : ALENGTH)
  ACCESS PROTOTYPE diracc
    ESCAPE readerr, writerr

    PROC PROTOTYPE read(adr : ADDRESS, size : ALENGTH, vmadr : ADDRESS)
    PROC PROTOTYPE write(adr : ADDRESS, size : ALENGTH, vmadr : ADDRESS)
  END ACCESS

  PROC PROTOTYPE read(adr : ADDRESS, size : ALENGTH, rmadr : RMADDR)
  PROC PROTOTYPE write(adr : ADDRESS, size : ALENGTH, rmadr : RMADDR)
  PROC PROTOTYPE lastbyte RETURNS l : ADDRESS
  PROC PROTOTYPE extend(end_address : ADDRESS) RETURNS e : ADDRESS
  PROC PROTOTYPE reduce(end_address : ADDRESS) RETURNS r : ADDRESS
END OBJECT
```

This prototype must specify all the routines that are needed by "seqmem" (i.e. "lastbyte", "extend" and "reduce").

2.23 **USER MEMORY**

The current description of the system has each user's files allocated throughout the system on any of the available volumes. This creates two problems. First, the user must be aware of these volumes which is really an irrelevant storage management detail. Secondly, this creates a management problem as it is difficult to restrict the amount of space a user is allowed to acquire. This is because it would be difficult to keep a record of which capsules belong to which user. To solve these problems an intermediate storage level can be introduced between the volume and the user created objects.

Each user would be allocated one of these intermediate memories called a "user_volume", in which the user's files would be allocated. The intermediate memory would be similar to the VTOC memory, but would make it possible to restrict the amount of space a user could use. A restriction could be implemented because all the user's capsules would be allocated in his memory. To allocate entities in a "user_volume", the "user_volume" name would be used as a prefix in a declaration instead of some irrelevant VTOC name, for example:

```
VAR f : buhr.seqfile
```

which allocates the sequential file "f" in user memory "buhr". As will be seen in chapter 3, the user volume name "buhr" may be inferred in some situations.

From a storage management standpoint, the intermediate memories are really superfluous. However, from the user's standpoint it removes irrelevant storage management details and from the system administration standpoint it allows better control of the storage resources on the machine.

2.24 REPRESENTATIVE

The storage that comprises an instance of "seqfile" on the volume is not directly addressable. Thus, when data in "seqmem" is accessed by the hardware, these references do not access the data on the volume directly. While "dircap" manages the space for the file on a volume, it does not support hardware-instruction references to transfer data to and from the volume into memory. This real memory address to volume address translation is normally referred to as paging and is performed by an entity called a pager. To allow the data in a file to be addressed directly by the hardware, it is necessary to create an entity that is prepared to make "seqmem" accessible to the hardware.

As well, to support the type of concurrent access required by a file (such as multiple readers but only one writer as for "seqfile"), it is necessary to provide an entity that is prepared to manage this concurrent access. This support of concurrent access is discussed in detail in chapter 5.

In both these situations the entity that is going to provide either paging or concurrent access must have the following properties:

1. The entity must be unique for an object and it must persist as long as the object is being accessed no matter how many users access the object during the access period. In the case of the pager, this means that one pager will be created for all accessors to the file. This is important as it prevents creation of multiple page tables for an object which in turn precludes multiple copies of a page of a file. In the case of concurrent access, the entity collects together information about all the accessors to an object. This is necessary to properly share the object.
2. The entity's data must be transient and separate from the object. If information about how an object is being accessed is stored in the object, then there is the problem of dealing with this information in the event of a system crash. When the system restarts, it may be difficult to know which files were being accessed before the crash and hence which must have old access

information deleted from them. If the data is transient and separate, then its loss after a crash does not affect the object.

To provide these properties a new entity is introduced called a representative. A representative's purpose is to contain volatile data associated with accessing an object. A representative is only needed for an object. A representative is not needed for a memory because any representative information for a memory can be associated with its underlying capsule.

The representative is specified on an OBJECT definition by the REPRESENTATIVE clause, as in:

```
OBJECT seqfile(...) CREATES sm : seqmem(...) REPRESENTATIVE arb : seqrep
```

and

```
OBJECT dircap( ... ) REPRESENTATIVE pt : seqpager
```

The representative for "seqfile" will be discussed in chapter 5. The representative for "dircap" will provide the necessary facilities to make the overlying "seqmem" addressable by the hardware. The variable "pt" contains all the routines that manage the page table and handle page faults from the hardware and page-outs from the operating system.

2.24.1 Representative Instantiation

The operating system creates a representative whenever an object is accessed either implicitly during declaration or explicitly by a direct reference or by creation of an access variable. When a memory is accessed, a representative will also be created for its underlying capsule which is also accessed. The operating system will guarantee that only one representative is created during access to a particular object. All other concurrent accesses that occur during an access will use the unique representative created during the first access. Representatives are allocated in system memory and their creation is discussed in detail in chapter 4.

2.24.2 Type of the Representative

For the operating system to interface with the representative for paging, the representative must conform to a fixed prototype for a pager. Only then can the operating system ensure that all the proper routines are present that may be called by the operating system and hardware during paging. As has been mentioned, not all representatives are pagers. For example, "seqfile's" representative does not perform paging; it only deals with concurrent access issues. The pager for "seqfile" is part of the representative for the capsule that underlies "seqmem". This is because a "seqfile" object is only addressable via the underlying memory. This is expressed syntactically through the CREATES clause on "seqmem". A CREATES clause on a MEMORY definition implies that the underlying capsule is managing its space in another memory and that the capsule representative manages its pages in real memory during access. Hence, it is possible for the compiler to discern which representatives must conform to the pager prototype.

2.24.2.1 Pager Prototype

The pager prototype that is defined in the system must be consistent with what is required by the hardware. In fact, the pager prototype is derived from the definition of the machine.

For the compiler to generated proper object code it must be aware of the particular hardware for which it will be generating code. One important aspect of the hardware will be its use of page tables for performing dynamic address translation. Traditionally this page table is a data structure pointed to by an operating system data structure, or by a special hardware register. This page table is manipulated by routines in the operating system.

In an object-oriented system, the hardware page-table pointer will point at more than just the data structure for a page table. It will point at a representative, which defines not only the page table but the routines that manipulate the page table and other data items necessary in handling the paging process. The paging prototype then is the fundamental prototype for any object that can be referred to by the hardware page-table pointer. Thus, the definition of the hardware that is assumed by the compiler would contain the pager prototype definition, along with other definitions. This definition would be similar to the following:

```
OBJECT PROTOTYPE pagetabtype
  ACCESS PROTOTYPE pageacc
    PROC PROTOTYPE pagefault(adr : ADDRESS)
    PROC PROTOTYPE pageout(ptno : Pageno)
  END ACCESS

  PROC PROTOTYPE extend(end_address : ADDRESS) RETURNS e : ADDRESS
  PROC PROTOTYPE reduce(end_address : ADDRESS) RETURNS r : ADDRESS
  PROC PROTOTYPE pagefault(adr : ADDRESS)
  PROC PROTOTYPE pageout(ptno : Pageno)
  PROC PROTOTYPE pageflush
END OBJECT
```

The "pageout" routine in the access class would be called by the operating system to cause a page to be written back into the file. The "pagefault" routine would be called by the hardware to handle a page fault from a data reference in a program.

2.24.3 Referencing the Representative from an Object

Procedure and class definitions in the OBJECT definition can refer to the representative by the name specified in the REPRESENTATIVE clause. The representative contains volatile information about the object necessary for accessing the object. Some of this information must be derived from the object itself. Clearly, this cannot be done if the object does not yet exist. Hence, the representative must be created after the object is created during an object declaration.

This implies that no references can be made to the representative during the initialization of the object because the representative does not yet exist. However, the definition of an object, such as:

```
OBJECT seqfile(...) CREATES sm : seqmem(...) REPRESENTATIVE arb : seqrep
```

makes it appear that the representative should be able to be referred to. As well, it is difficult to detect such references at compile time because the intialization code could call a procedure that refers to the representative. Hence, this situation must be handled as a dynamic check, and can be handled by setting the representative pointer to null before beginning the initialization code.

A similar argument can be made about referring to the representative in the termination code for an OBJECT definition. When this code is executed, the object is being deleted and hence will no longer be accessed. Hence, a representative is no longer needed and could provide no logical function. As a result, the representative cannot be referred to during the termination code.

2.24.4 Referencing the Object from the Representative

While the object may refer to the representative, so too might the representative refer to the object. For the representative to call routines in the object it must have a means of refering to the object. If a representative needs to refer to the object it represents, the object address can be passed as a parameter to the representative, as in:

```
OBJECT dircap( ... ) REPRESENTATIVE pt : seqpager(SELF)
```

where "SELF" is a builtin keyword denoting the current object.

2.25 PAGER

The following is an outline for a representative for "dircap". Routines prefixed by "systemem" deal with the memory in which a representative is allocated.

```
OBJECT seqpager(cap : dircap) : pagetabtype
  VAR pages : [MAXPAGES] RECORD
      status : (loaded, unloaded),     ¢ the structure of this table
      rmadr  : REF page                ¢ is dictated by the hardware
    END RECORD

  ACCESS pageacc
    ¢ these two routines must be in the access class so that
    ¢ multiple asynchronous pagefaults to the same object will
    ¢ all go through the monitor mechanism provided by the
    ¢ access class, as well, any special paging optimizations
    ¢ would be placed in these routines.

    PROC pagefault(adr : ADDRESS)
      seqpager.pagefault(adr)
    END PROC

    PROC pageout(ptno : Pageno)
      seqpager.pageout(ptno)
    END PROC
  END ACCESS
  ¢ these routines must be in the object because several users
  ¢ may be using the same page table to access an object, and
  ¢ only one at a time may access these routines.

  PROC extend(end_address : ADDRESS) RETURNS e : ADDRESS
    ¢ the page table is expanded to the closest multiple of the
    ¢ specified address
  END PROC

  PROC reduce(end_address : ADDRESS) RETURNS r : ADDRESS
    ¢ the page table is reduced to the closest multiple of the
    ¢ specified address
  END PROC

  PROC pagefault(adr : ADDRESS)
    WITH pages[adr'VALUE >> pagebitlength]
      IF status ¬= loaded THEN
        rmadr <- systemem.allocate_page(adr'VALUE >> pagebitlength,
                                        PAGESIZE)
        cap.read(adr, PAGESIZE, rmadr)
        status <- loaded
      END IF
    END WITH
  END PROC

  PROC pageout(ptno : Pageno)
    cap.write(ptno, PAGESIZE, pages[ptno].rmadr)
    systemem.free_page(pages[ptno].rmadr)
    pages[ptno].status <- unloaded
  END PROC

  PROC pageflush
    VAR vadr : Pageno
    VAR i : 1..MAXPAGES

    vadr <- ADDRESS(0)
    for i <- 1 TO MAXPAGES
    LOOP
      IF pages[i].status = loaded THEN
        cap.write(ADDRESS(vadr), PAGESIZE, pages[i].rmadr)
      END IF
      vadr'VALUE +<- PAGESIZE
    END LOOP
  END PROC

  ¢ initialize the page table and expand it to the current
  ¢ size of the object.

  FOR i <- 1 TO MAXPAGES
  LOOP
    pages[i].status <- unloaded
  END LOOP
  extend(cap.lastbyte())
TERMINATE
  pageflush()
END OBJECT
```

Notice that there are routines "pagefault" and "pageout" in both the access class definition and the OBJECT definition. This is simply a result of allowing concurrent access to the same pager. Each user will have his own access class for a pager, and it is through this access class that the operating system and the hardware refer to the pager for that user's "pagefault" and "pageout" routines. Since both these operations affect the page table, the operations must be serialized and this is accomplished by the corresponding routines in the object.

"Segpager" is one of a small set of paging objects provided by the system. These paging objects are probably part of the nucleus. Other paging objects are provided to take advantage of inherent properties of other storage management techniques used by other file types. Normally, the user and the system programmer will not be writing their own paging objects, because any error would have catastrophic effects on the system. However, new paging schemes are not precluded, and if written, would probably require recompilation of the nucleus to insert them.

2.25.1 Optimizations via Representatives

Because each capsule can specify its own representative, a representative may be written in such a way as to optimize the paging of a file based on some knowledge of the way the file is accessed. It is through the pager access classes, that are created for each user's access, that optimizations can be performed. For example, due to the sequential manner in which a sequential file is accessed, it is probable that pages more than two before the current page (i.e. the page that was just paged in) will not be referenced (unless the file contains a very large record). These pages could be paged out to free up real memory. This could be accomplished by having each pager access class have a queue of the last two pages referenced and explicitly call "pageout" in the pager object for pages no longer needed. (In general, it is necessary to maintain usage counts on pages so that, during concurrent access, pages that are in use by another user will not be paged out.) This is just the old concept of buffering done at the paging level. Similar types of optimizations can be done for other file types.

More than one access class definition may be defined for an object. Each access class definition can provide a different type of access to the object. For example, in the case of an index-sequential file, it may be desirable to have an access class definition for indexed accessing and an access class definition for sequential accessing. Each access class definition provides only the necessary components for that particular type of access. These two access class definitions can appear in both the memory, capsule and representative for the file; hence, it is possible to associate different page table access class definitions with each. In this way a different type of access class can be provided by the page table routines to optimize the particular type of access to the file.

Another form of optimization that could be applied during access to a sequential file is predictive paging. Because of the sequential access of a sequential file it is almost always true that the next page fault will occur in the page after the current page. Thus, the pager could initiate a read for the next page before returning from processing the current page fault. When the next page fault occurs, the pager can test the status of the faulted page, which is one of:

1. unloaded : Which means that the prediction was incorrect, and the fault occured on a different page. Thus, a read will be issued for this page and the pager waits for the read to complete.
2. reading : Which means that the faulted page is still being read in and the pager must wait.
3. loaded : Which means that the faulted page has been read in while processing the previous page.

Clearly, this predictive paging algorithm requires not only extra code in the pager, but the ability for the pager to function asynchronously of the capsule.

2.26 <u>VTOC</u> <u>REPRESENTATIVE</u>

The representative for a "seqmem"/"dircap" is created whenever it is necessary to directly address an overlying memory that is managed by a capsule. This same notion also holds between the VTOC memory and the volume capsule. Here a representative will be created to allow direct addressing of the VTOC memory that is managed by the volume capsule. Thus, the VTOC representative provides paging of the VTOC memory so that it can be accessed directly by the code for the VTOC, and so that the capsule's data can be addressed directly by the code for the capsule defined in the VTOC.

The code for the VTOC representative is basically the same as for "dircap" with the possible variation that the pager might fix pages in real memory. This is desirable because the capsule's data items contained in a page may be accessed frequently when the capsule's overlying memory is being accessed. Clearly, there would be one VTOC representative allocated for each volume in the system to allow direct access to the VTOC(s) on each device.

2.27 ASYNCHRONOUS PROCESSES

To have a programming system, it is necessary to support multiple asynchronous processes. This requires programming language features to facilitate the different types of asynchronous processing. It is not part of this thesis to investigate in detail the programming language mechanisms to support how asynchronous processes communicate. However, some thought has been given to the major programming language constructs used to identify what entities can be separate processes. These programming language features are mentioned only briefly to allow this and other examples requiring asynchronous processes to be understood.

2.27.1 Process Attribute

Currently, several different type constructors (CLASS, ACCESS, OBJECT, MEMORY) have been introduced. Their main difference has to do with the memory in which instances of the type definition are allocated. These differences will be essential for processes, as well. It is desirable then to augment the existing type constructor to specify asynchronous processing and retain the different memory allocation features. This might be done by adding the PROCESS attribute to a type constructor, as in:

```
OBJECT PROCESS dircap ...
```

Instances of such objects are separate processes, and the subsequent access of the entity either directly or through an access class invokes the task. In this example, "dircap" would execute in parallel with other processes, such as the user.

It would be naive to think that the only change necessary to change from a serial entity to an asynchronous entity is to add the attribute PROCESS. The entire algorithm for the type may have to be redesigned to achieve the parallelism desired. This will require additional programming language features that allow the process to communicate with other processes and to proceed after communication or to wait [ANDRE83].

2.28 SUMMARY

The following is a brief summary of the definitional structure of the system:

```
MEMORY systemem
  ¢ constant declarations for the system

  OBJECT dirpager
    :
  OBJECT seqpager
    :
  OBJECT seqrep
    :
  MEMORY PROTOTYPE dirmemtype
    OBJECT PROTOTYPE dircap(...) CREATES ... REPRESENTATIVE pt:seqpager
      :
    MEMORY seqmem( ... ) CREATES ...
      :
    OBJECT seqfile( ... ) CREATES ... REPRESENTATIVE arb : seqrep
      :
  OBJECT disk3350      ┐
    :                  ├ for each type of device
    :                  ┘
  OBJECT volcap REPRESENTATIVE pt : dirpager   ┐
    :                                          ├ for each type of volume
    :                                          ┘
  MEMORY vtocmem ...   ┐
    OBJECT dircap ...  ├ for each type of VTOC
      :                ┘
  PROC allocate_page( ... )
    :
  PROC free_page( ... )
    :
  PROC allocate( ... )
    :
  PROC free( ... )
    :
  VAR spindle01 : disk3350   ┐
    :                        ├ for each device
    :                        ┘
  VAR vtoc1 : vtocmem( ... )    ┐
    :                           ├ for each volume
    :                           ┘
END MEMORY

VAR sys : systemem    <- magic declaration (i.e. must be made
                         before the system can exist)
```

"Systemem" is the definition of the system memory. This memory will contain data items for declarations such as devices, volumes, and user memories, etc. When new devices or volumes or users are added to the system, it is this memory that is augmented to contain the new data item. As a result, this memory must persist, and be loaded into real memory when the system is initially loaded. The system then uses the remaining portion of real memory to allocate pages and representative objects.

The programming language constructs OBJECT, MEMORY, and ACCESS class in conjuction with the CREATES clause provide a powerful facility for defining files. The OBJECT provides a type-safe interface between the user and the file. The MEMORY provides a general facility to implement any storage management scheme required to support the high level functions of the object. The ACCESS class defines the operations needed to access the object. Its duration determines the duration of access to the file. These programming language constructs can be used not only to define files that the end user will see, but also the underlying structure that supports files.

While the example "seqfile" may not demonstrate completely the power of these constructs, I have tried to indicate how more powerful operations could be performed (eg. index-sequential file). However, the main point here is not the detail of the file definition but that the file can be defined within the programming language. "Seqfile" and "seqmem" are defined in two to three pages of source language. They are available anywhere that they are visible and each usage can be statically type checked by the compiler. The code for the underlying capsule, representative and volume will usually exist already, and hence can be used by any file designer, thus, even further simplifying this task. Thus, the main goal of incorporating file definition into the programming language has been achieved.

Chapter III

INTERACTIVE SYSTEM

<u>Statement execution mode</u> (<u>SEM</u>) is the mode in which programming language statements are entered and are executed immediately. This is in contrast to <u>program definition mode</u> (<u>PDM</u>) where statements are entered to be explicitly invoked at a later time. The commands that a user uses in SEM are defined via an interactive language. The objectives of interactive languages vary greatly, hence the commands they provide are peculiar to the applications the interactive language is supporting. The applications of interactive languages generally fall into two groups: those that deal mainly with access to files (called command languages), and those that deal with conventional data types (called interactive programming languages).

Traditionally, most command languages (TSO [IBM78], Shell [BOURN78]) support only a few data types other than files, principally those necessary for conditional control, and only a few control structures, principally program invocation. Because of the simple design of the command languages, they are usually not used for defining programs. Instead, the command language acts as an interface between general purpose programming languages, which are defined only in PDM, and files. This interface between general purpose programming languages and command languages is undesirable as it forces the user to learn another programming language, the command language, and the inter-language interface is error prone as there is usually little or no type checking between the file and its use in the programming language. This paradigm is forced on a user on the one hand as a result of the lack of file definitions in the general purpose programming language, and on the other hand as a result of the lack of other data types and control structures in command language's.

In interactive programming languages (APL [IVERS62], LISP [McCAR62], Smalltalk [GOLDB83]) the programming language statements are used as the interactive commands, in theory allowing the interactive programming language to be the same as the PDM language. In practice, however, there are usually differences between the interactive programming language and the PDM language. For example, in APL it is not possible to use the transfer operation to transfer to another interactive statement. As well, APL supports files and their manipulation only by using extra programming language commands, such as)LOAD,)SAVE, etc.; these extra programming language commands can be termed a command language. This interface between interactive programming language and command language is undesirable for the same reasons that it is undesirable between general purpose programming languages and command languages, and again is a result of lack of file definition capability in the interactive programming language.

I feel it is important that the programming language used in SEM be the same as the programming language used to define programs in PDM [WEGBR71, WEGBR74, HANSE76, MASHE76, JONES77, BEECH79, FRASE83]. Thus, the full programming language must be available for use as a command language. As well, the programming language constructs discussed in chapter 2 already accommodates the definition and access of files. Together these two ideas, that is, full file definition capabilities and full interactive programming language, form the basis of the proposed interactive system.

There are several substantial advantages to adopting this scheme. First, the user does not have to learn another language to communicate directly with the system. Secondly, the system has the potential to guarantee consistency between interactive statements and PDM. This is true not only for files but also for simple variables which now are available in SEM. Lastly, new file types or other type/data items defined in the programming language become immediately accessible in SEM.

Currently, implementors of command languages and interactive programming languages have recognized these desirable features and have attempted to augment their systems to incompass them. New command languages are introducing control structures, simple procedures, and more data types [BOURN78, IBM78]. These additions point to the desire to have a complete programming language for interactive use. In interactive programming languages, there is a trend toward augmenting the programming language to provide a more consistent and thorough approach to manipulating files [WHEEL81]. Again, these programming language additions are introduced so that the interactive programming language contains files. Thus, command languages and interactive programming languages appear to be converging towards the common objective of a single programming language for SEM and PDM. However, these additions in both command languages and interactive programming languages become simply cosmetic if the fundamental issue of file definition is not addressed.

However, this scheme has some minor disadvantages. Specialized command language features may not be able to be retained as they may not fit into the programming language satisfactorily. For example, in the UNIX[2] Shell, pattern matching is allowed on file names that are passed as arguments to Shell routines. But in a strongly-typed programming language, it may not be possible to mimic this facility. However, having a complete programming language that functions identically to the programming language in PDM, will more than compensated for these types of deficiencies.

3.1 COMPILEABLE INTERACTIVE PROGRAMMING LANGUAGE

Most command languages and interactive programming languages do not have declarations for variables and procedures, and thus are not able to be compiled (at least in the conventional sense); hence they are interpreted. The programming language presented in chapter 2 can be compiled and there is no reason why this cannot also be done in SEM. This decision to compile the programming language in both modes stems from the desire to have both modes appear not only functionally equivalent to the user but also functionally equivalent within the system. That is to say, the system should not be implemented as two distinct subsystems: one subsystem to handle SEM and another totally separate subsystem to handle PDM. The same compiler used in PDM should be able to be used in SEM. This does not preclude the use of software interpreters in SEM; for example, to provide facilities over and above the hardware interpreter for debugging purposes [TEITE84].

---

[2] UNIX is a Trademark of AT&T Bell Laboratories.

In SEM all the programming language constructs are available; hence all executable statements are useable. These executable statements refer to entities, such as procedures and variables, definable by programming language declarations that are made in SEM. Thus, all variables and consequently all data types declarable in the programming language must be available in SEM.

To make possible compilation of executable statements in SEM it is necessary to retain the names and the other information from the declarations that are made in SEM. Since the data structure needed to retain this information very closely resembles the symbol table created temporarily during compilation, I simply refer to it as a _symbol table_ (_ST_). A ST provides the compiler with the information necessary for type checking and object code generation during compilation.

To make possible the immediate execution of these compiled statements in SEM, storage for the entities referred to in the statements must exist. Thus, the data items declared in SEM must not only update a ST for subsequent references, but also cause immediate allocation of the necessary storage. Thus, there must exist a _data area_ (_DA_) during SEM containing instances of all the declared variables. This DA is similar to the storage area allocated during program execution for declarations made within a programming language block.

The preceding discussion suggests the need for two distinct entities: the ST, where names and declaration information is stored, and the DA, where values of variables are stored. In contemporary programming languages, the ST is created at compile time; the DAs are created during program execution. In contemporary command languages, these two entities exist in a somewhat limited fashion in the form of a file directory and files; the directory contains the name and type information for a file as well as a pointer to the file, while the record area for the file is allocated separately. In interactive programming language, such as APL, these two entities exist together in the workspace; the workspace contains the ST for all names as well as the storage for the corresponding variables.

In a compiled system the separation of the DA and the ST is important. This is because the compiler manages only the ST, since the DA is not created until execution time. Hence, the manipulation of the ST by the compiler should not imply existence of the DA. While the ST and the DA are conceptually separate entities, the two will still be associated with one another (i.e. the storage in the DA will be laid out according to the specifications given by the ST). This is essentially what happens in normal compilation except that the ST is usually lost at execution time when the DAs are created. My proposal is to simply retain both, since this is clearly required for SEM.

3.2 ENVIRONMENTS

Together the ST and its DA, form an environment. The environment appears at first glance to be an object because the environment's DA is not allocated in either the stack or the heap but in a separate DS. But there is one important difference. The DA of an object is laid out and fixed when its definition is compiled; the structure of the DA cannot be changed. The structure of the DA for an environment, however, is constantly being changed. New declarations are made in SEM which result in an extension of the DA. As well, facilities to remove or modify old data items must exist which can result in further changes to the DA.

Currently, in compileable programming languages, changes to the DA can only be made indirectly by changing the source language for the declarations and recreating the ST. Subsequent execution of the program creates a new DA that conforms to the ST. However, the environment DA cannot be re-created in the same way because all the current values contained in the environment must be retained. As well, the environment ST cannot be re-created because its recreation implies a potential re-mapping of the DA, and the DA that already exists must conform to the ST. Thus, the current technique of recreating a new ST and DA for compilation of the programming language statements and subsequent execution in SEM cannot be used.

Neither the ST nor the DA for an environment can be destroyed and subsequently re-created, and yet they must be able to be extended or changed; the only solution is to allow modification of an existing ST and have its corresponding DA modified appropriately. By modifying the existing ST, it is possible to arrange that incremental changes in the ST cause only the necessary incremental changes in the DA. Since the DA must correspond to the ST, the ST changes define the exact changes required in the corresponding DA. These changes can then be applied to the DA so that it corresponds to the ST.

3.2.1 Supporting Environment Changes

Modifications such as the declaration of new entities in the environment is relatively easy to perform on an environment ST. Declaring new variables simply requires augmenting the ST with the new information and causing the DA to expand to provide the storage for the new entity. Expansion of the environment DA is possible because it is in a DS. As well, other facilities such as displaying the ST information can also be implemented. However, over time data items in the environment may no longer be required or the type of a data item may need to be changed.

While declaring entities in an environment is analogous to declaring entities in a programming language, modifying and erasing a declaration have no analogue in a programming language. These functions are currently performed by modifying the source text for a declaration and re-compiling the program. Since these functions are clearly essential, they must be able to be performed on the environment ST. ST entries can be deleted or fields in ST entries can be changed. However, this results in a discrepancy between the DA and the ST. As a result, the DA must be modified to reflect the change because the DA always conforms to its ST.

3.2.2 Environment Primitive

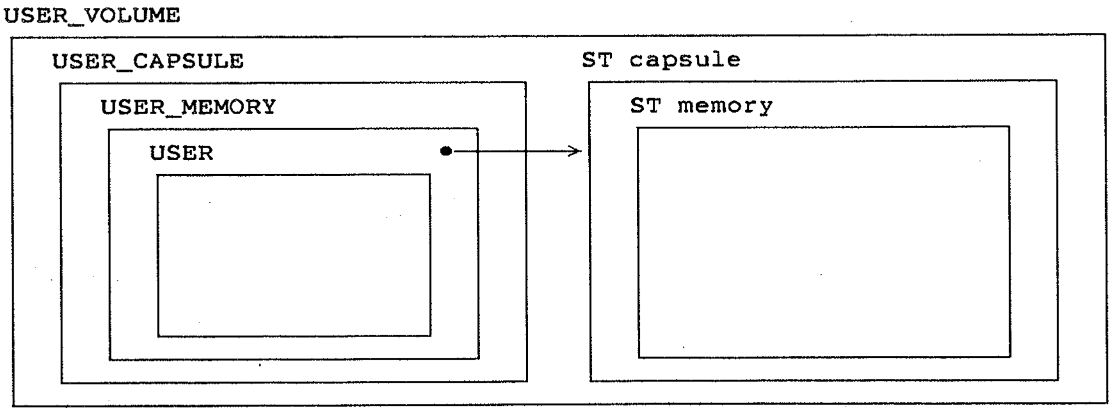
To support environments in the programming language, an ENVIR primitive is introduced. ENVIR is a primitive like OBJECT and it is used to define a type. Instances of ENVIR's can be declared and created as for objects. The characteristic property of an ENVIR, however, is that it also creates a separate ST to map its DA. This is different from all the type constructors discussed so far which have only one ST definition that maps all instances of that type. This is an important point, namely, that there is always a one-to-one relationship between ST and DA for an environment.

3.2.2.1 Using the Environment Primitive

An example outline for an environment called a "user" is presented. The "user" might be the entity created for each user on the system. Part of this entity is the memory that contains all the storage for a user.

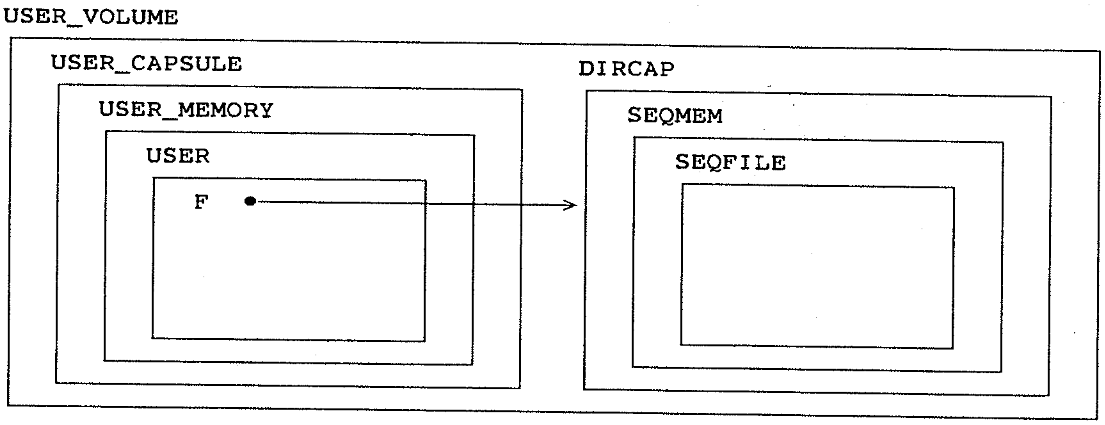
```
ENVIR user(maxsize : POSINT) CREATES umem : user_memory
  :
MEMORY user_memory(maxsize : POSINT) CREATES ucap : user_capsule
  :
OBJECT user_capsule(maxsize : POSINT) CREATES uvol : user_volume
  :
MEMORY user_volume(maxsize : POSINT) : dirmemtype
  :
```

"User" is the environment, and hence causes the implicit creation of a ST for that environment. (An implicit CREATES clause for the ST can be imagined on the ENVIR declaration). "User_memory" is the memory that contains storage for the variables declared in the environment. "User_capsule" is the capsule that maps the "user_memory" memory. "User_volume" is the memory that contains the blocks of storage for "user_memory" and the blocks of storage for any other memories allocated in environment "user".



Entities declared in environment "user" that create a DS must create their memory separately from the "user_memory". This is because "user_memory" is not meant to contain other memories. "User_memory" is meant to manage the DA for the environment which contains all the simple variables (i.e. variables that do not create separate memories). The storage for objects such as files comes from the user's volume which also contains both environment DS and ST memories.

When an object like "seqfile" is declared in "user", the compiler can determine that it cannot be allocated in "user_memory" because "user_memory" does not have the correct prototype and hence cannot support the allocation of another memory inside it. The compiler then follows the CREATE chain and discerns that a "seqfile" can be allocated in "user_volume", and hence creates a pointer to the "seqfile" in "user_memory" and creates the "seqfile" in "user_volume".



This is analogous to the situation of a procedure that allocates a local "seqfile". The "seqfile" is not allocated on the stack with the procedure's local variables but instead is allocated in the memory in which the program executes, and points to the file from the stack. This is how the volume for an object can be determined implicitly. As well, the volume may be explicitly specified to allocate the object in a volume other than the users (as was shown in chapter 2).

The type "user" can be used to allocate new users on the system. Declarations, such as:

```
VAR buhr, zarnke : user(100K)
```

would create two new environments each with its own ST and DA and two user memories each of which can be up to 100K in size. These could then be used by users "buhr" and "zarnke" to create new variables in SEM. Of course to allow declarations of users, the entity in which the declarations are made must itself be an environment which is ultimately contained in a memory that supports allocation of a "user_volume" (i.e. a memory of prototype "dirmemtype").

3.2.2.2 User Prototype

Many system routines will expect to use entities of type "user". To make the system more general, a prototype is defined for a general user. As a result, several different types of users, with different rights and facilities, can be defined in the system, and used by the system routines.

```
ENVIR PROTOTYPE usertype CREATES ...
   ACCESS PROTOTYPE useracc
      PROC signon ...
        :
      PROC locate ...
        :
   END ACCESS
END ENVIR

ENVIR user( ... ) : usertype ...
```

Procedures "signon" and "locate" will be discussed later.

3.3 PROGRAM EDITOR

In conventional programming languages, the compiler builds and destroyes the ST based on the block structure of the program and this ST does not change during the block. In an environment, however, individual symbols can be modified and deleted at any time. This makes management of the environment ST significantly different from the ST management done during conventional compilation. One way of resolving this difference is to have program language statements in SEM that modify the ST once it has been created. However, new operations such as erasing and changing ST entries are not part of the conventional ST management facilities provided by a compiler. It is not a good idea to augment the programming language so that the compiler could perform these operations in SEM, because these extra language constructs would then be available in PDM since the programming language is the same in both cases. These operations are undesirable in PDM because compilation would be impossible if declarations could be changed or deleted within a block. For these reasons, a special program called the program editor

(PE) is used to manage the environment ST. The PE can provide all the types of storage management operations needed for the environment ST [RUDMI82].

All the access to the environment ST is done through the PE. As a result, the functions of ST construction and the entering of ST information, collectively call ST management, are not performed by the compiler when it accesses the environment during SEM. Thus, the compiler in SEM need only deal with the executable statements. To compile the executable statements, the compiler must be able to access and read the environment ST created by the PE.

To manipulate the environment ST the user invokes the PE. Through PE commands, the user should be able to accomplish approximately what can be done with a text editor and recompilation, although perhaps in a different way. Just as statements in SEM interactively modify the environment, PE commands are entered interactively to modify the environment ST. The editor provides a structured approach to manipulating the ST, as opposed to the laissez-aller manipulation possible with a text editor. After editing the environment ST a user can return to SEM to perform operations on the data items in the DA.

PE mode and SEM interpret the names of variables in different ways. In SEM when an operation is performed on a name it means that the operation is applied to the value in the storage associated with the name. However, in the PE, an operation on a name is applied to the ST entry associated with that name. This distinction also appears in other interactive programming languages such as APL, where a user is either in function definition mode or in SEM.

3.3.1 Program Definition

The programming language in SEM is essentially the same as the programming language in PDM; hence, the PE must allow declarations of variables of any type, and the compiler must be able to compile references to variables of any type. However, to allocate space for variables declared in the environment ST, and to generate object code for references to variables, information about the type of the variable must be available. Since entities defined in PDM will be referred to in interactive mode, the same information will also have to be retained for these PDM entities. For example, ST information about the OBJECT definition "seqfile" is needed to allocate a "seqfile" when one is declared in the environment ST, and ST information is needed about the procedure definitions in the access class definition "seqacc" when calls are compiled in SEM. Therefore, it is necessary to retain at least part of the ST information for class, object, and procedure definitions to allow compilation of declarations and references in SEM.

In current systems, the ST for definitions in PDM are usually not retained after compilation of the definition. However, as has been pointed out, some of this information is needed for both interactive declarations and interactive compilation. Since it is not easily discernable what information for types and procedures must be retained, and since the entire ST for any definition is necessary to support any sensible high-level symbolic debugging, I have decided that the entire ST be retained in PDM.

With the decision to retain the PDM ST comes the same problems associated with retaining the environment ST. Since definitions are changed over time, the ST associated with a definition must be able to be changed and the function is obviously done most easily by the PE. Thus, the PE allows changes to both the environment ST and PDM STs; thus, this mode is simply referred to as PDM. As a result, the compiler no longer needs to handle any declaration statements, as they are all handled by the PE. Thus, the term compiler now refers to a program that only performs a portion of the work of a conventional compiler. That is, it generates object code for declarations made by the PE and for executable statements entered in SEM and PDM, but it neither builds nor updates the ST.

Along with the declarations for a definition in PDM are the source language statements. The source language statements are traditionally retained in a text-string form intermixed with declaration statements. However, in this system the textual form of the declarations has been replaced by PE operations. Yet, the source language statements for a definition must be associated with these declarations in order to provide the proper context for names in the source language. For this reason, it is necessary to associate the source language with its corresponding ST entry. The source language may then be entered using any technique (i.e. text-string or parse-tree).

3.3.2 Clusters

In a block structured language entities are nested within one another to restrict or obtain access to other entities. This is reflected definitionally by the static nesting of blocks, thus creating a definitional hierarchy. Any entity in the programming language that can have local symbol declarations can potentially add to this hierarchy. These entities that have internal structure are called cluster definitions. The programming language constructs OBJECT, PROC, MEMORY, etc. are therefore cluster definitions.

The PE supports this definitional hierarchy by creating a local ST (LST) for each cluster definition that is declared. The LST contains the declarations and associated source language statements for that cluster definition. Within a cluster definition there can be definitions for other entities that are themselves clusters. The PE will create subordinate LSTs for each such entity. In fact, each LST simply corresponds to the scope introduced by that cluster definition, and the LST structure corresponds to the static scope of nested blocks in the programming language.

For example, an OBJECT definition can have several internal procedure definitions. When the OBJECT definition is entered, a LST instance is created. When each procedure is defined within the OBJECT definition, a LST is created for each procedure definition. The LST for each procedure definition is subordinate to the LST of the OBJECT definition as it would be for STs created in ordinary compilation. The user can access this ST hierarchy by using the names of the cluster definitions (i.e. OBJECT definition name, procedure definition name, etc.).

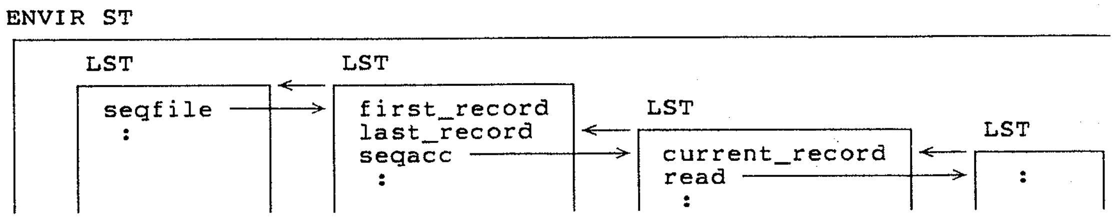
3.4 ST HIERARCHY

It is assumed that almost all of the system will be written in the programming language and hence STs will exist for all of the system programs. It is only with this information that the PE and the compiler can verify references throughout the entire system. These STs will form a hierarchy corresponding to the static structure of objects, procedures, etc. defined in the system, and this hierarchy is called the ST hierarchy.

This ST hierarchy will be large, too large to imagine it fitting into a single memory. Thus, the ST hierarchy must be subdivided into multiple memories. However, subdividing the ST so that each ST entry is in a memory would be undesirable. This would result in too much overhead in storage space and access time. Thus, some compromise in between the two extremes must be found. Clearly, the LSTs provide natural breaks in the ST hierarchy for sub-dividing. Currently, I have chosen to sub-divide the ST hierarchy at the LST for the environment ST. Each environment ST then is implicitly allocated as a memory. I think this level of division is approximately in the middle of the two extremes. However, if finer or coarser divisions are found necessary, then the PE can be augmented to allow explicit specification for allocation of memories for other LSTs. For example, in a multi-person project it might be desirable to allow simultaneous modification to a particular definition. This can only be done safely if each LST is a memory and the PE guarantees mutual exclusion among users editing each LST that is a memory.

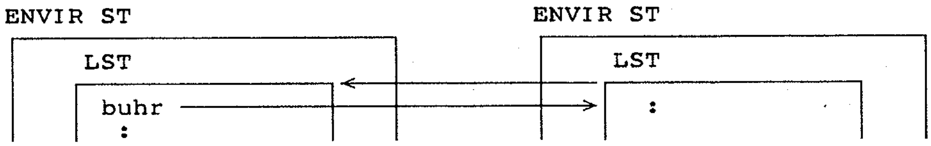
3.4.1 Structure of the ST Hierarchy

The ST for an environment contains multiple LSTs, one per cluster definition. The declaration for a structured entity must allocate a ST entry in the current LST and a new LST for its local declarations. The ST entry must contain a pointer to its subordinate LST. Also, there must be a pointer back from the subordinate LST to the originating LST, for example, in "seqfile":



These pointers are necessary to allow traversing both upwards and downwards in the ST hierarchy. This traversal is done by both the PE and the compiler.

However, the LSTs form only a portion of the complete ST hierarchy. Declaration of a variable that results in an environment will cause the LST for that variable to be allocated as a memory. This results in the pointer to the ST of the environment and the pointer back to the environment-variable ST-entry being an inter-AS pointer.



3.5 DS HIERARCHY

The DS hierarchy shows the relationship between entities that are created as memories. This relationship is a hierarchy because environment declarations can be nested within one another. These DSs and the data items that are declared within them persist until they are explicitly deleted by the user. Such DSs are controlled by the user; that is, the user decides when the data items in them are to be created and/or deleted.

There is a root environment created when the system is initialized. This environment contains the data items that define the system. Hence, the system environment persists for the duration of the system. A user's environment is defined as a variable in the system environment, and thus has the potential to persist as long as the system root persists; however, a user may explicitly modify his environment, adding to or deleting from it. This ability to persist within the system is not part of the programming language, but part of the system in which the programming language operates. As a result, it is not possible to specify via the programming language whether an entity should persist. An entity simply persists as long as its containing data area persists.

3.5.1 **Virtual System**

The system's environment contains all the data items that define the system and marks the root of the system DS hierarchy. It is conceivable to imagine more than one system environment defined on a machine, each defining a completely separate system and DS hierarchy. Thus, it is possible to imagine having separate systems running concurrently.

3.6 **VIEW RULES**

View rules define what action is to be taken when a name cannot be located in the current scope (i.e. LST). The scope rules of most programming languages specify that if a name cannot be located in the local scope the containing scopes are searched up to the level of the uppermost containing scope. The uppermost containing scope, in conventional programming languages, is the user's program or maybe a library scope in which a user's program is compiled. Thus, the automatic search is reasonably efficient because the number of containing scopes is relatively small and the STs to be searched are all within a single memory.

This is not the case for a ST hierarchy that represents all programs currently contained in the system. The number of containing scopes is potentially large, the amount of information in each scope is potentially large, and the STs are sub-divided into multiple memories. While it is conceivable to search upwards in the ST hierarchy until the root of the hierarchy is reached to resolve a name, it would, in practice, be inefficient to allow this unrestricted search. Also, this unrestricted search may lead to interpretation of a misspelled name.

To prevent this problem, the conventional scope rules could be abandoned altogether. This would require that all references not in an entity be qualified in such a way as to uniquely specify which entity in the ST hierarchy it is located in. However, this solution is unsatisfactory due to the excessive amount of qualification that the user would be forced to specify.

A compromise is to restrict the scope search to a certain level, after which explicit qualification is used. The level at which the ST search is stopped must not be so low as to force excessive qualification, and not so high so as to nullify any advantage of the restriction. I think that the most practical position would be to stop the scope search at the level of an OBJECT definition. This allows the conventional scope search in nested procedure and class definitions. Since an environment is treated like an object, this restricts the scope search for a user's environment to within the user's environment.

To refer to items outside an OBJECT definition the user must explicitly state that the ST search should continue outside the current OBJECT definition (called open qualification). The syntax to denote open qualification is:

```
^ standard-programming-language-qualification-list
```

The "^" indicates that the reference is outside of the default scope search.

Open qualification implies that the first name in the qualification list is looked up in containing scopes continuing outside of the current OBJECT definition until the name is found or the root of the ST hierarchy is reached. Hence, this is just the normal unrestricted scope search. However, since it must be explicitly requested it means that it will not occur for every undeclared or misspelled name in a program.

3.6.1 **Referencing Types**

Type definitions appearing as part of program development in one part of the ST hierarchy are available throughout the ST hierarchy for subsequent use for other program development. The only restrictions that a user must be aware of is whether the type is dependent on its containing DA. That is to say, if the type definition refers to a variable outside itself, then variables of that type cannot be allocated unless that variable is accessible where the type is used. Otherwise, if the type does not refer outside itself (a pure or self-contained type like "seqfile") then this type definition can be referenced and used anywhere in the ST hierarchy.

3.6.2 Qualification in Programming Language Statements and PE Commands

Both the programming language statements and the PE commands must be able to access throughout the ST hierarchy. The combination of view rules and open qualification provides the ability to make these accesses. Hence, both programming language statements and PE commands will adopt the view rule and allow open qualification.

One point needs to be made concerning qualification of names in programming language statements and in PE commands. Open qualification (i.e. using "^") is used to refer upwards, while normal qualification is used to refer downward in a scope. For qualification in programming language statements it is sometimes incorrect to refer downward into a scope because that scope may not have storage associated with it at the time the reference is made. For example, given the following program structure:

```
PROC p
  PROC q
    VAR x : INT
        :
  END PROC
  :
END PROC
```

it is invalid to refer to variable "x" outside of procedure "q" in a programming language statement. This restriction is imposed because the storage for "x" does not exist when the reference occurs. PE commands, however, do not have the same restrictions as programming language statements since they deal strictly with the ST and not with the data item instances. Hence, it is perfectly valid for a PE command to refer to "q.x", as that command is only dealing with the ST entry for "x". Thus, while the syntax of qualification is the same for PE commands and programming language statements, some references may be precluded in the programming language statements.

3.6.3 Deferred Resolution

There are some instances where a program must be generalized to access a cluster that is determined dynamically, for example, if a user writes a program that wants to write on the caller's terminal. Currently to do this, the caller would have to pass his terminal (i.e.

the object that represents his terminal) explicitly to the program, as in:

```
p(terminal, ... )
```

This is undesirable because it forces all containing cluster definitions to also have this explicit parameter, and it forces the caller to include this argument in all calls.

Instead it would be better if only the program that wanted access to the terminal had to refer to it. This can be done by specifying the type of terminal to be accessed and then establishing a dynamic connection to the actual terminal instance that is available at execution time.

To specify this dynamic connection the user refers to the type of the terminal instead of a terminal variable. For example,

```
PROC p( ... )
  :
  ^systemem.termtask.terminal.write( ... )
```

the qualification "^systemem.termtask.terminal" traverses the ST hierarchy to the type "terminal" in entity "termtask". The compiler knows that this refers not to a data item but to a type and hence must be found at execution time. The compiler handles this by generating object code to dynamically search for the first containing object that is of type "terminal". Once found the routine "write" can be selected out and called. This type of reference is called a <u>deferred reference</u>.

The implementation of the deferred resolution requires a dynamic search to locate the DA whose type is the same as the specified type. This can be done because all DAs point back to their corresponding ST entry to enable symbolic debugging. Hence, it is possible to search through these DAs for the ST address of the type specified in the deferred reference.

This search is performed by searching the stack for a procedure that is contained in a cluster that has the same type as the deferred reference. This is possible since all procedure definitions are contained in either an object or another procedure that is ultimately contained in an object. The dynamic search will search up the stack until it finds a procedure contained in an instance of the deferred type. It is possible to search up the stack and fail to find a procedure in a cluster of the correct type; at this point an exception is signalled.

This idea can be generalized further to allow specifying a prototype in a context that normally requires a cluster instance name. Here, the dynamic search would be more complicated as the prototype of each cluster must be located in order to compare its ST address to the prototype of the deferred reference.

Notice that while ST addresses are used during execution, the actual ST is not being searched. Hence, the STs do not have to be accessible at execution time in order to perform deferred reference only the ST pointers in the DAs of the instances.

3.7 MANIPULATING THE ST USING THE PE

The PE permits the creation and manipulation of STs (and incidentally LSTs). The PE can be implemented in many ways and it is not my intention here to detail how the editor is implemented, but rather to define, as completely as possible, the kinds of operations that are necessary and desirable in the editor.

These operations are presented as a set of editor commands, although this is not meant to limit the editor to operating only through such commands. It might provide analogous operations through a full screen editing approach. The functions that the PE supplies and the way these functions are performed by a user are, in general, independent issues. While a full screen, multi-window interface to the PE may be desirable to enhance user productivity, it does not affect the basic functions the PE supplies.

PE commands are entered by the user and are "executed" immediately. Execution in this case, however, refers to the ST changes implied by the operations, and not to changes to user variable values. Thus, the editor commands are interactive much like regular programming language statements entered in SEM. However, the meaning of names is different.

3.7.1 User-visible Structure of the ST

Each LST is a series of entries, one for each name appearing within the corresponding cluster definition. The exact contents of each entry depend on the type of the entry. However, the following is a list of the fields that will be in all ST entries:

1. name : The character string that forms the entity's name.
2. category : The information that indicates what additional information is contained in the ST for this entity, what special syntax is used to define this entity, and what are legitimate ST attributes. Each category has its own special syntax for definition and its own special ST data structure to store this information. Some different categories are OBJECT, MEMORY, VAR, etc.
3. PERSPECTIVE : The perspective information associated with the entity. (Perspective will be discussed in chapter 5.)
4. TYPE : The pointer to the type of the entity.
5. INITIALIZATION : Executable statements that are performed when the entity is instantiated, including code for a procedure. Also, termination code for class-like definitions is included here.
6. DOCUMENTATION : A description of the entity in this ST entry.
7. LST pointer : If the entry is a cluster, the pointer to the LST for that cluster definition.
8. address or displacement : The address of the variable in relation to the begining of a DA, or the displacement of the entity within a structure.

Unlike conventional declaration statements that specify all the necessary information to make a declaration, only the entity name and the category are necessary to create a ST entry. Each additional piece of information can be added in any order or modified at any time. Some of the attributes of a ST entry can be referred to by specifying the name of the ST entry and the name of the attribute using the following syntax:

x'INITIALIZATION, x'TYPE, etc., where x is an entity name

Associated with almost all definitions of entities is source language. For simple variables this would be intialization; for cluster definitions this code would be the initialization and termination statements. These executable statements are part of the definition of these entites and as such are handled by the PE. Also, I have chosen to associate documentation with the definition of each entity. While documentation is not needed for compilation it is certainly very desirable.

Because there are no declarations in the source language, the entering and modification of source language (or documentation) is largely independent of the PE operations. How the source text is stored or how it is modified does not affect the PE or the environment, and thus is not considered in great depth in this thesis. Likewise, the documentation is entered and modified in a similar fashion. Hence, the source language and the documentation may be manipulated by conventional text editing methods either line or full screen, or using a parse-tree editor.

3.7.2 PE Commands

With this brief description of the ST structure, the PE commands that operate on the ST will be given. The commands are presented based on the type of function that they perform on the ST.

3.7.2.1 Creation of new ST Entries

The creation operations create new entries in the ST. There are two broad kinds of new entities: clusters and simple (non-structured) items. The minimum information that can be supplied in a creation command is the name of a new ST entry and its category. The category is essential as it dictates what type of information can be associated with the name. For example, information and operations allowable for a procedure definition are different than for a variable declaration. The procedure definition introduces a new LST. Thus, the ST entry allocated for a new entity is determined from its category.

The syntax of the declaration statements for each category of cluster definition is different. To reflect these differences in the PE commands, the command name is the category name. The category name defines the remaining syntax allowable for that command. This eliminates the complexity of a single command that must handle all possible category forms, and emphasizes to the user the category, as it is the command name.

The syntax of new declarations is as follows. When syntax is specified, the notation used is the following:

- quotes (') enclose a terminal symbol
- [x] means that "x" is optional
- x | y indicates that "x" and "y" are alternatives (i.e. one of "x" or "y" is chosen)
- { } is used to group alternatives within a single definition

```
creation-cmnd ::= cluster-defn [ positioning-information ] |
                  type-defn [ positioning-information ] |
                  declaration [ positioning-information ]

cluster-defn ::= 'RECORD' [ 'PROTOTYPE' ] name |
                 'PROC' [ 'PROTOTYPE' ] name |
                 'CLASS' [ 'PROTOTYPE' ] [ 'PROCESS' ] name |
                 'ACCESS' [ 'PROTOTYPE' ] [ 'PROCESS' ] name |
                 'OBJECT' [ 'PROTOTYPE' ] [ 'PROCESS' ] name |
                 'ENVIR' [ 'PROTOTYPE' ] [ 'PROCESS' ] name |
                 'MEMORY' [ 'PROTOTYPE' ] name

type-defn ::= 'TYPE' name '=' type-constructor

type-constructor ::= type-name |
                     '(' ordered-set ')' |
                     subrange |
                     '[' type-list ']' type-name

type ::= type-constructor | cluster-type

type-list ::= type-list ',' type | type

cluster-type ::= 'RECORD' | 'CLASS' | 'OBJECT' | 'ENVIR' | 'MEMORY'

declaration ::= 'CONST' name-list [ attribute ] |
                'VAR' name-list [ attribute ] |
                'ESCAPE' name-list |
                'PARAMETER' name-list [ { attribute | ':' 'TYPE' } ] |
                'KEYWORD' name-list [ { attribute | ':' 'TYPE' } ] |
                'RETURNS' name-list [ attribute ] |
                'CREATES' name [ attribute ] |
                'REPRESENTATIVE' name [ attribute ] |
                'SEGMENT' name-list |
                'PERSPECTIVE' name [ attribute ]

attribute ::= [ '{' perspective-name '}' ]
              [ ':' type ]
              [ '<-' initialization ]
              [ '¢' comment-text ]

positioning-information ::= 'AFTER' name | 'BEFORE' name
```

In the cluster definition, all entities but PROC are essentially types; for example,

RECORD x

corresponds to Pascal:

TYPE x = RECORD ...

"Type" includes keywords RECORD, CLASS, OBJECT, etc. in order to permit composite types to be used in "declarations"; such types will be anonymous.

In a declaration, CONST, VAR, and ESCAPE declare a constant, variable, or escape respectively. PARAMETER and KEYWORD declarations specify parameters to procedures, objects, etc.; RETURNS specifies the result of a procedure. CREATES and REPRESENTATIVE declarations are used in defining objects and memories as discussed in chapter 2. A SEGMENT declares an memory that contains object code and is discussed shortly. PERSPECTIVE declares an entity used to control visibility and is discussed in chapter 5.

Each creation command can be followed by optional positioning information. The positioning information allows the user to locate a new ST entry among other ST entries within a cluster definition. For example, an entity can be added before or after an existing ST entry. The positional specification allows complete flexibility in specifying the ordering of ST entries both initially and for subsequent additions. If no positional information is given the entity is added to the end of the appropriate context: PARAMETER declarations are added to the end of the parameter list; all other declarations are added to the end of the LST.

In general, the ordering of ST entries is only important to the user in terms of documentation. Ordering is only critical for positionally dependent cases, such as parameters. This is because the ST is constructed by the PE before compilation, and not constructed during compilation as in conventional compilation systems. Also, because the PE does not extract names appearing in executable statements, such names do not have to be resolved until compile time. Thus, references in declarations and executable statements do not affect the ordering of ST entries, as they do when the declarations and executable statements are in a sequential file. Consequently, a construct like a FORWARD declaration is not necessary.

Because the PE only deals with the ST and not with executable statements, any executable statements are simply stored in source language form for subsequent processing by the compiler. There are several contexts where expressions can occur other than within initialization code, for example in a subrange. The type definition:

```
TYPE r = j/2..n+1
```

specifies that "r" is a subrange with lower bound "j/2" and upper bound "n+1". Here, the PE does not have to fully parse this range, it simply retains the range for the compiler to process at the point of the type declaration.

3.7.2.2 Modification of ST Entries

The modification operations allow changes to existing ST entries including deletion of a complete ST entry. Changing the attributes of a ST entry requires as complicated a command as its initial specification. Thus, the modification commands must have almost identical syntax to the creation commands. However, it is not possible to have one command handle both situations. This is because it would not be possible to:

1. differentiate between adding a new ST entry and attempting to modify an entity that does not exist.
2. differentiate between creating a new ST entry with the same name as an existing ST entry, and modifying the ST attributes of an existing entry.

To be able to detect these error situations, the PE should be able to syntactically distinguish between a creation and modification operation, so that the appropriate semantic rules can be applied.

The modification commands available are:

```
modification-cmnd ::= mod-cmnd  |  delete-cmnd  |  rename-cmnd
mod-cmnd ::= 'MOD' name [remaining syntax depends on category of name]
delete-cmnd ::= 'DELETE' { name | nameattr }
rename-cmnd ::= 'RENAME' name 'TO' name
nameattr ::= name''''attrsel
attrsel ::=  'PERSPECTIVE' | 'TYPE' | 'INITIALIZATION' | 'DOCUMENTATION'
```

Positioning information is not required for modification commands, as they all refer to existing names.

The MOD command is used to change information that is contained within a ST entry. Since the ST entry exists already, I did not want to have the user re-specify the category. Thus, the name in the MOD command is looked up to find its category, and that category dictates the syntax of the remainder of the command. The new information specified in the rest of the MOD command replaces what is already there. For example, the declaration:

```
VAR p : 0..10 <- 0 ¢ counter
```

can have its initialization and documentation changed by:

```
MOD p <- 1 ¢ new comment
```

Determining the syntax of the command by context in this way makes parsing somewhat more difficult. However, from the user's standpoint, this syntax is preferable to re-specification of the category. In a full screen editor, these operations would likely be performed by cursor positioning and this obviates the syntax problem.

Deletion of information for an attribute is accomplished by using the DELETE command. For example, to delete the comment attribute for variable "p" the following command is entered:

```
DELETE p'DOCUMENTATION
```

As well, deleting ST entries is also done by the DELETE command:

```
DELETE p
```

When a ST entity is deleted, so too are all LSTs that depend on it. To change the name of a ST entry, a RENAME command is needed:

```
RENAME p TO pcount
```

It is not possible to directly change the category for a ST entry. For example, the category for a variable cannot be changed to PROC because the meaning of the attributes is different and the internal data structure is different. However, it might be desirable to support some changes to the category. For example, a change from a variable to a parameter, or from a CLASS definition into an ACCESS definition, or from an OBJECT definition into a MEMORY definition might be allowed. However, what changes would be allowed are implementation considerations that are not part of this thesis. Currently, the only way a user could accomplish these changes is to create a new entry with the desired category, copy over any pertinent attributes, delete the old entity, and rename the new entity to the old.

3.7.2.3 Movement Operations

The movement operations cause ST information to be moved from a source location in the ST to a destination location. While movement operations are not essential since the same effect can be accomplished by deleting and re-entering information, the PE should allow the same kinds of operations as a text editor allows on conventional source languages. Most text editors allow source language to be copied or moved from one location to another in the source file. The corresponding operation on the ST is to allow copying of a ST entry from one LST to another LST, and moving a ST entry within a LST or to another LST. (Copying within a LST is not possible because this would cause two STs entries to have the same name.) As well, since a ST entry is composed of several attributes, it may be desirable to allow copying or moving of individual attributes from one ST entry to a corresponding attribute in another ST entry.

Two movement commands are required to support these operations: a command for copying and a command for moving. Each of these commands will have two forms: one for working with an entire ST entry and one for working with the ST entry attributes.

The commands for movement of an entire ST entry are discussed first. The syntax of the two commands are:

```
move-cmnd ::= 'MOVE' source-name {'AFTER'|'BEFORE'|'TO'} destination-name
copy-cmnd ::= 'COPY' source-name {'AFTER'|'BEFORE'|'TO'} destination-name
```

When either command specifies AFTER or BEFORE, the source ST entry is moved and positioned accordingly after or before the destination ST entry. When either command specifies TO, the source is moved into the ST entry specified by the destination. Here, the destination must have the same category as the source. In this case, the name of the destination can be the same as the source because the name field of the destination is not changed.

The commands for movement of ST entry attributes are:

```
moveattr-cmnd ::= 'MOVE' source-nameattr 'TO' destination-name
copyattr-cmnd ::= 'COPY' source-nameattr 'TO' destination-name
```

Since attributes cannot exist without a corresponding ST entry, both the source and the destination ST entrys must exist for movement of attribute information; thus, only TO is allowed. Also, since an attribute can only be moved to its corresponding attribute in the destination, the attribute need only be specified once.

To move either a ST entry or a ST entry attribute from one location in the ST hierarchy to another may require qualifying one or both of the source and destination names.

3.7.2.4 Effects of Moving ST entries

The movement of ST entries throughout the ST hierarchy can affect references contained in the ST and in the source language associated with the ST entry. The ST type field contains a pointer to the ST entry for the type. When a ST entry for a type is moved within the same memory, the pointer to it from other ST entries remains the same. If the ST entry is moved outside the memory the type pointer must be changed to an inter-AS pointer, but this should not affect the ability to access the type. The references to global variables in source language that is moved can subsequently become invalid. This happens when source language is moved outside the scope of the definition.

Unless the source language is stored in parsed form, the PE would not be able to detect this. This error would then be detected during the next compilation of the moved entity.

3.7.2.5 Positioning Operations

The positioning operation provides the means to establish a ST on which operations are to be done. This then defines the ST position from which the ST search begins for all subsequent PE commands. Positioning is not necessary for programming language statements because each statement is embedded at a particular position in the ST hierarchy and hence the ST search starts at this position. Conversely, the PE commands are not embedded in the ST hierarchy. Hence, the view rule for PE commands requires that the user explicitly specify the location in the ST hierarchy that is the start of the ST search.

The syntax of the positioning command is:

```
locate-cmnd ::= 'LOCATE' { cluster-defn-name | 'END' }
```

Each LOCATE establishes a LST for a cluster definition to be used as the starting point of the ST search for subsequent PE commands. One method of handling the LOCATE commands is to stack the previous cluster definition name for each new LOCATE. That is to say, each LOCATE pushes information about the cluster LST on a stack and each END pops the previous entry off the stack and repositions the user to an earlier cluster definition. For example, a user can locate to procedure definition "p" to make changes, then to a procedure definition "q" in a different cluster definition to make changes (in doing this the user may have to explicitly qualify "q"). Upon ending the LOCATE command for procedure definition "q" the user is back using procedure "p's" LST for subsequent ST searching. This technique allows the user to quickly relocate himself to another cluster definition to effect changes and then return exactly to where he left off in his prior editing.

3.7.2.6 Listing Operations

The LIST command lists all or selected portions of a ST entry, particularly including the source language and documentation if any.

```
list-cmnd ::= 'LIST' name  |  nameattr
```

The listing can be displayed on a terminal, or printed as those given in chapter 2 for the definition of "seqfile". It is also desirable to be able to list all of a cluster's ST entries as well as the descendents of any sub-cluster definitions. This can be accomplished with the command:

```
display-cmnd ::= 'DISPLAY' cluster-defn-name ['DEPTH' integer-constant]
```

which lists all the ST entries for the specified cluster definition and for each contained cluster definition down to the specified number of levels of containment given by the DEPTH option.

3.7.2.7 Comments on the ST Operations

The ST operations, described through the PE commands, should provide the user with adequate tools for manipulation of the ST. In the actual implementation more commands and command options might be included to aid the sophisticated user. As well, the ST operations could be implemented via a full screen interface as opposed to a command-line interface. Creation operations could be done by displaying a menu of possible categories and then displaying a template for the selected category. The fields in the template could be filled in to create the ST entry. Modification operations could be done by displaying an existing ST entry and allowing the user to edit the fields. Movement operations could be done by positioning to fields or entire ST entries to cut the source information, then positioning to the destination to paste it. Positioning operations could be done by zooming in on cluster definitions and by multiple windows. What is important, from the standpoint of this thesis, are the operations on the ST and what effects they have, not in how they are specified by the user.

3.7.3 Manipulating Source Code and Documentation

As has been stated, almost all ST entries will have both source language statements and documentation associated with them. Up to now the examples for entering information in these fields has been done using a short form, as in:

```
VAR x <- a * b ¢ product
```

Clearly, a more sophisticated technique must exist for entering multiple programming language statements for cluster definitions, and multiple lines of documentation. This is accomplished by having the source language and documentation field each point to an entity called a <u>packet</u>. Internally, a packet may be subdivided into lines (records) as in a conventional text file or organized in a more structured manner.

3.7.3.1 Packet Commands

Like the ST operations, the packet operations of the PE can be implemented in many ways. However, the PE must provide the basic operations for adding, changing, moving, copying, deleting, and listing packet information. While this may appear like the same set of operations required for ST manipulation, the operations are applied to an entirely different internal data structure and so the details of the operation are different. In one case the data structure is the LST hierarchy and in the other it is the internal packet organization.

The packet commands can refer to packets by the ST names and the 'INITIALIZATION and 'DOCUMENTATION attributes. Thus, initialization and documentation can be entered in two ways: using <- and ¢, or using the attribute and packet commands. Because editing of packets does not affect the ST, details of packet commands will not be discussed.

3.7.4 <u>Using the PE</u>

The following example will describe, briefly, how a program is developed. The example illustrates the entering of the definition of the OBJECT definition "seqfile". It is assumed that the user is already in the PE (how the user enters the PE is discussed in a following section), and the PE prompts the user for a command with an "E:".

When the user enters the PE from SEM, he is positioned to the ST for the environment. The PE allows the definition of a cluster to be entered in the environment ST. By locating to this cluster definition, the user enters PDM. For example, the definition:

```
E: OBJECT seqfile
```

entered into the environment ST would create an entry in the environment ST as well as create a LST for the composite entity "seqfile". The user enters the program definition by locating to "seqfile":

```
E: LOCATE seqfile
```

This operation positions a user to the cluster definition for "seqfile" and implies that subsequent declarations appear within the OBJECT definition "seqfile". The user then enters all local definitions and declarations via the PE. Thus, the editor treats declarations entered into an environment differently from those entered into a program definition. Declarations for the environment ST modify the LST for the environment and cause allocation of space in the environment DA, whereas declarations in a definition only modify the LST.

The user can now declare the parameters and the local variables for "seqfile". The declaration of the parameters might be done as follows:

```
E: PARAMETER record_type : TYPE
E: PARAMETER initial_size : INT AFTER record_type
```

The category of PARAMETER is specified to distinguish it from a local variable. The first declaration needs no positional information as there are no declarations initially. The second parameter has positional information indicating that it is after parameter "record_type"; thus, it is the second positional parameter. This positional information could have been omitted since the default is to add to the end of the LST.

The declaration of the CREATES variable can now be added to the definition of "seqfile". This is done as follows:

```
E: CREATES sm : seqmem(record_type, record_type'SIZE * initial_size)
```

Here, "seqmem" must be defined before this addition of ST information, so that the type field of "sm" can point to the ST entry of "seqmem". The values of the parameters are stored as a packet for subsequent processing by the compiler.

Next, the declaration of the local variables may be done:

```
E: VAR first_record, last_record : REF(sm) record_type
E: ACCESS seqacc
E: PROC write
E: PROC recreate
    :
```

Positional information is not necessary as these are the first local variables in the cluster definition, and hence they are added in the order they are given unless otherwise specified.

Finally, the initialization code for "seqfile" is entered into the source packet "seqfile'INITIALIZATION".

The user can then list the entire OBJECT definition by:

```
E: LIST seqfile
```

or list the source language or documentation:

```
E: LIST seqfile'INITIALIZATION
E: LIST seqfile'DOCUMENTATION
```

or information on a specific entity:

```
E: LIST seqfile.first_record'DOCUMENTATION
```

which lists the documentation of the local variable "first_record".

Before "seqfile" can be compiled, the user must complete the definition of "seqacc", "write", and "recreate". This is done by locating to the appropriate definition and declaring the local variables and source language. Several of the local entities of "seqacc" are procedure definitions and these too must be completed. Again the user can LOCATE to each and declare parameters, local variables, and input source language and documentation.

The user is not obliged to enter all the information at once; that is, the PE allows incremental program development. However, entering a program with the PE requires that some information be entered in a particular order. Both the PE and the compiler require definition before use, but unlike conventional programming languages this is temporal rather than physical. Once, an entity is entered into the ST, it can be found (with proper qualification) regardless of its ST position.

Thus, program definition must be entered top-down as far as the basic program structure. However, this program structure need only be a skeleton structure. Once, the program structure is in place, development can occur in any order by any number of users as far as entering source language and testing.

As well, the complete definition of an entity is not necessary before references to the entity can be used in compiled statements. Once the interface for a cluster definition or the type for a simple entity is entered, references to it can be compiled. Thus, declarations to "seqacc" could be made and compiled, and calls to the access components could be made and compiled before the component definitions were completed in "seqacc".

3.7.5 Effects of ST operations on the Environmental DA

Normally, when a user modifies a ST for a DA, any outstanding instances of the old ST are now inconsistent with the new ST. Attempting to update all DAs to correspond to the modified ST is complicated by the fact that there will frequently be many instances of a particular ST. To update these DAs would require pointers from the ST to all outstanding DAs. Even if these DAs could be found, the task of updating the list of outstanding DAs would be difficult and expensive. First, the list of ST pointers would have to be updated dynamically as each DA was allocated and as each DA was deleted. Secondly, in general, it may not be possible to expand a DA without invalidating the references in object code that operates on the DA.

When a user modifies the ST for an environment, the environment ST becomes inconsistent with the environment DA. As the user re-enters SEM from PDM and hence makes the environment DA accessible, the inconsistency between ST and DA is detected. At the point of detection the system must either prevent the access or update the DA to reflect the new description of the ST. In the case of an environment, this updating is possible because there is a one-to-one correspondence between the environment ST and DA. As well, this DA is maintained in a DS which isolates effects of modifications.

To be able to update the environment DA the system must know what has changed from the previous DA. This information is provided by the PE by marking the environment ST entries that are changed while in the PE. As the user re-enters SEM the compiler is invoked. The compiler examines the environment ST for changed ST entries, and generates the necessary object code to update the environment DA (deleted ST entries can now be removed from the ST). This object code is executed immediately upon entry into SEM to update the environment DA. This object code can expand the DA for new ST entries, delete storage for deleted ST entries (including entities pointed at by pointers in this storage), and re-allocate storage for ST entries that have been modified. Notice that the compiler is performing a non-conventional operation: processing a modified ST and generating object code to update a DA.

In general, determining which data items are affected by a ST modification can be difficult. This is particularly true when types are modified. When a type is modified all variables of that type must be updated, and as well, any variables whose type is dependent on the modified type must also be updated. In this latter case, multiple passes may have to be made over the ST to discover all affected types and variables.

After determining the effects of ST modifications, the compiler must generate object code to update the environment DA. This too can be difficult as the deletion or modification of a variable must not invalidate data items whose ST entries remained unchanged. As well, freeing space for deleted data items without affecting other data items and subsequent prevention of access to these items is also difficult. How this is accomplished is discussed in detail in chapter 4.

These changes to the environment DA can cause further inconsistences to arise. Any program that previously referred to a deleted or modified data item is now inconsistent with the DA. In some cases re-compilation of the inconsistent program may correct the inconsistency; in other cases changes to the program will be needed.

3.8 PREPARING A PROGRAM FOR EXECUTION

All object-code-producing cluster definitions (i.e. any cluster except RECORD) must be compiled before they can be used and are called modules. The resulting object code must be placed in an entity from which the hardware can execute it.

3.8.1 Compiling Programs

Programs can be compiled by using the PE command:

```
compile-cmnd ::= 'COMPILE' module-name
```

The compiler is then able to access the source text of the module, its LST, and its containing LSTs which form the module's environment. During compilation, the compiler still has to create auxiliary tables to do compilation, such as a parse stack, but no alteration of the ST is necessary. As well, any compilation errors are reported to the user.

To resolve references in other environment STs the compiler has to search them to extract the information needed to compile a module. Because the compiler can find all ST entries for variables by traversing the ST hierarchy, the compiler can guarantee type consistency for all references, and because the location of all names are resolved at compile time, generated object code can be frequently executed without a separate link-editing operation.

3.8.2 Segment

The executable instructions generated by the compiler must be stored so that they can be subsequently executed when a reference is made to them. The object code for a module is stored in a memory specifically for object code called a segment. Since all references in the object code have been resolved, the segment need only be made accessible to the hardware and the object code contained within it can then be immediately executed. Thus, like the Multics system, there is no explicit load phase when accessing a module.

The segment may contain object code for several modules, and usually will. For example, a procedure and all its subprocedures:

segment

```
PROC p
  PROC q
    PROC r
      :
```

can be compiled together and placed in the same segment. It is also possible to compile into a segment procedures that are not nested, for example:

segment

```
PROC p
  :
END PROC
PROC q
  :
END PROC
```

The decision as to which modules are placed in a segment is made by the user. Frequently, modules that are related to one another will be placed in the same segment but this need not be so. It is important to note that just because modules are compiled together in the same segment and call one another does not mean that modules from other segments cannot make calls to these modules as well. The only restriction on calls is that imposed by the block structure rules. This is significantly different from conventional systems where modules containing object code usually have only one entry point callable from SEM.

3.8.2.1 Creating a Segment

Each segment is explicitly declared by the user. For example:

```
SEGMENT 1
```

declares a segment "1" into which object code for a module can be compiled. The type SEGMENT is built into the system; it likely has no explicit definition.

Segment declarations can only appear in an OBJECT definition. This is because the accessing of the object code in a segment is tied to the explicit accessing of an object. This is discussed in detail in chapter 4. Multiple segments can be defined in an OBJECT definition and a module defined within the OBJECT definition can have its object code compiled into any of these segments.

It is important to note that while segments are declared with other variables of an OBJECT definition, they are instantiated not at execution time but immediately upon their definition. Since the object code for the OBJECT definition and its contained entities is generated at compile time, the segment which contains this object code must exist by this time. Thus, the PE manages segment declarations.

3.8.3 Compilation Type

When a module is compiled, the user can specify one of several options that control: when the object code is generated, where the object code is physically located, and what type of object code is generated. The compile type describes this information and is specified as part of a module definition. Thus, each module has a unique compile type, which is used by the compiler. The compiler must know this information not only when compiling the module but also for compiling references to the module from other places in the system. Three different compile types are introduced.

3.8.3.1 Multiple Copies of Program Code

This compile type specifies that the compiler generate one base copy of the object code for a module, and that this object code is copied into each segment that uses it. This is the scheme used in many conventional systems.

In conventional systems, the object code for source language statements generated by the compiler is not directly executable. This object code is referred to as an object module. The object module contains: the object code generated from the compilation, a relocation dictionary (RLD) for addresses that must be relocated when the module is loaded, and an external symbol dictionary (ESD) which names all modules referred to by this module and all modules defined by the object module. All object modules that refer to one another are copied together into an entity called a load module. This combining together of object modules is necessary in conventional systems because calls between load modules are not usually supported via the programming language call mechanism, only via a system executive mechanism. Thus, the object module itself is not directly executable and need not be. It is necessary to retain object modules to facilitate this copying operation. Thus, the object module provides a base copy of the object code for a module that can be copied into different load modules. In this system this corresponds to object modules being duplicated into different segments resulting in multiple copies of the object code throughout the system.

In this system an object module is stored in a specific segment. Any reference to this object module causes an executable copy of the object code to be placed in the referencing segment. There are two cases that constitute a reference:

1. A direct reference via a call or declaration.
2. An indirect reference which results from a copied module being referred to within another module.

Like the conventional object-module load-module systems, this system assumes that only one copy of such a module is placed in a particular segment. Thus, for those routines that are copied it is necessary to maintain a conventional object module form. This multiple copy form of object code requires extra information to be maintained (eg. RLD, ESD); an <u>object</u> <u>module</u> (<u>OM</u>) in this system will be slightly different from conventional object modules as described later.

The purpose of this compilation type is to reduce the number of inter-segment calls. Inter-segment calls involve making another segment addressable, and transferring control from one segment to another; as a result they might seriously degrade performance.

The disadvantage of COPIED routines is that copies are not updated automatically if the OM changes. It is the user's responsibility to do this. It is not until the module containing the reference is re-compiled that the inconsistency between the reference and the OM is detected. As a result, COPIED object code can persist in an out-of-date form in the system. However, this is not a problem since the out-of-date object code is referred to by equally out-of-date references in the same segment.

3.8.3.2 Single Copy of Program Code

This compile type specifies that the compiler should generate only a single copy of the module's object code and that all references access this code. The object code produced in this case is directly executable throughout the system, and so a single copy of the module code is sufficient. References to it may require an inter-segment call, if the routine is called from another segment.

The advantages of having a single copy of a module's object code are:

1. There is less space used by having only a single copy. This is especially important for system modules (such as "seqfile") that are used by many users.
2. When updates are made to a module, these updates will be reflected immediately throughout the system as there is only one copy of the object code.

The disadvantages of a single copy of a module's object code is that inter-segment calls will increase execution time.

3.8.3.3 Inline Copies of Program Code

This compile type specifies that the compiler bind the arguments to the parameters at compile time and generate inline object code for the routine based on this binding. Thus, new object code is generated for each reference to the entity, and the object code is physically placed inline at the call site. This is the conventional macro-substitution process with the added benefit of type checking.

Supporting INLINE compilation in this system is facilitated by the fact that the source language for a module is stored as a separate logical entity. That is, the source language packet for an inline module is not embedded in other source language as might be true if the text is stored using a conventional text file scheme. Thus, it is easy to locate the source language to perform the macro-substitutions and subsequent object code generation. The purpose of this type of compilation is to achieve maximum execution speed at the expense of using extra storage for the repeated object code. The disadvantages are the same as for COPIED as there are multiple copies of the INLINE module throughout the system which do not get updated automatically.

3.8.4 Separate Compilation

Only modules with separate compilation type are allowed to be compiled independently. A module that can be separately compiled is referred to as a separate compilation unit (SCU). A module without a compilation type is compiled as part of the SCU in which it is contained. The reason for not having every module separately compileable is that it may make possible some optimizations by the compiler (in respect to addressing modules).

3.8.5 Specifying a Compilation Type

The type of compilation is specified as part of a modules definition, as in:

```
module-defn ::= cluster-defn compile-type
compile-type ::= 'COPIED' segment-spec | 'SINGLE' segment-spec | 'INLINE'
segment-spec ::= segment-name | '*'
```

where the different compile types have the obvious meaning. The SINGLE and COPIED forms of the compile type explicitly indicate the segment that stores the object code, as in the following example:

```
OBJECT p SINGLE 1
```

This specifies that the object code generated, when "p" is compiled, is physically contained in segment "1".

To eliminate duplicate writing of a segment name for nested modules that are to be separately compileable in the same segment as the containing module, the segment name can be entered as a "*". For example:

```
OBJECT p SINGLE 1
  PROC q SINGLE *
    :
  PROC r SINGLE *
    :
END OBJECT
```

implies that procedure "q" and "r" are SCUs and their object code is contained in the same segment as the containing module "p", namely "l". Notice if a user wants to be able to separately compile a module he must specify a compile type, as in "q" and "r". Without this specification "q" and "r" would automatically be compiled when the containing SCU ("p") is compiled.

The initialization code for the OBJECT definition can also be specified to appear in one of the segments defined within the OBJECT definition. This allows the OBJECT definition to be defined completely, that is, its function and the location of its generated object code. Thus, when looking up a segment name, the search starts in the current module and not outside, for example:

```
OBJECT seqfile ... SINGLE seqseg
  SEGMENT seqseg
    :
```

Although segment "seqseg" is defined inside of OBJECT definition "seqfile", the object code for "seqfile" appears in segment "seqseg". As well, any other modules contained in "seqfile" can have their object code placed in "seqseg".

3.8.6 Example of a Compilation Type

The following nonsense example illustrates some of the different combinations possible with compile types and their effects:

```
OBJECT p SINGLE l
  PROC q SINGLE m
    :
  PROC r COPIED n
    :
  PROC s INLINE
    :
END OBJECT
```

The SINGLE type on OBJECT definition "p" specifies that the object code for "p" and any entities in "p" that do not have a compile type are compiled together and placed in segment "l".

The SINGLE type on procedure definition "q" specifies that the object code for "q" is compiled and placed in segment "record_type". Having "q's" object code in a separate segment might be useful during the debugging phase of "q". During debugging many compilations are done, each recompilation may causes reorganization on the segment into which the entity is compiled. For efficiency reasons (discussed later) it might be advantageous to have this reorganization done on a separate segment containing only procedure "q".

The COPIED type on procedure definition "r" specifies that the OM for "r" is compiled into segment "n". This OM contains the object code and any other information necessary to facilitate copying of the object code into other segments. A copy of the OM for "r" is automatically copied into any segment which has an instance of object "p" that calls "r".

The INLINE type on procedure definition "s" specifies that the object code for "s" is generated whenever a reference to "s" is made. The arguments are bound to the parameters in the expanded code and the resulting object code is placed in the segment where the reference occurs.

3.8.7 **Compile Type of a Prototype**

A prototype may also have a compile type. This is necessary for those situations where there are actual definitions in the prototype. For example,

```
CLASS PROTOTYPE c( ... )
  PROC p( ...)
    :
  END PROC
  :
END CLASS
```

In this example, the procedure definition "p" has been specified in the prototype and as a result cannot be added to by a type definition derived from this prototype. Depending on the compile-type of prototype "c" the object code for "c" may be separate or included into the segment for each type definition. If "c" has a SINGLE compilation type then the compilation of "c" will place the generated object code for "p" in the specified segment. Then, all types dependent on "c" will all refer to the single version of "p's" object code. If the compilation type of "c" is COPIED or INLINE the object code for "p" will be treated appropriately and will ultimately be associated with the segment for any type definition that depends on "c".

3.8.8 Environment Segment

So that beginning users do not have to define segments for their modules, a segment is defined in the environment ST for the user. Hence, if no compile type is specified for a module it will be compiled into the default user segment. The default segment can be added in the definition of "user" in the following way:

```
        ENVIR user( ... ) : usertype CREATES umem : user_memory
                                     SINGLE user_segment
          SEGMENT user_segment
          :
```

Thus, all modules without a compile clause, entered into an instance of type "user", will be compiled into the segment of the containing module, which is "user_segment". However, these modules are not SCUs, and hence the environment must be recompiled to cause recompilation of them. This could be done automatically when returning to SEM.

3.9 ORGANIZATION OF THE SEGMENT

The segment is organized to allow both the entering of COPIED and SINGLE object code by the compiler and the subsequent access of SINGLE object code by other modules through the call mechanism. This is further complicated by the need to support separate compilation of each SCU in the segment.

To allow separate compilation it is necessary to be able to move existing SCUs around in the segment. Movement is necessary because recompiled object code will, in general, occupy a different amount of space than it did previously. However, movement of modules affects the addresses compiled as part of a routine reference, any absolute addresses in the object code itself, and may cause other SCUs to be moved as well. A technique for solving these problems is presented.

3.9.1 Address Table

The address of an SCU in the segment must remain fixed from the time a module is declared because from that point on calls can be compiled to it. These calls require addresses which in turn must not change as a result of recompilation. To allow both fixed addresses for a module reference and movement of the object code within a segment, a level of indirection is introduced. A table of pointers, called the address table (AT) is allocated at the beginning of each segment, one entry for each SCU in the segment. Each AT entry contains the address of the corresponding SCU, as well as other information about the SCU. The position of the AT entry corresponding to a particular SCU constitutes the "address" of the routine. The entries in the AT are never moved and hence the position of an AT entry never changes.

When a definition occurs with a compile type of SINGLE or COPIED, the PE obtains an AT entry from the specified segment, and places the position of this entry in the address field of the ST entry for the entity. It is this position (along with the segment address, if it is an inter-segment call) that is used for any references to the routine.

When a segment is declared, an AT of some predefined size is allocated. However, the size of the AT can be increased dynamically if necessary. This is done by moving all routines in the segment creating more space for the AT. Of course this necessitates adjusting the SCU addresses in the AT entries and all addresses in the object code. Hence, increasing the AT size is an expensive operation; however, it will probably be done infrequently. The AT can only be made smaller if entries are freed at the end of the AT; however, in general, this will require that all SCUs in the segment be given new AT addresses, and hence, that all references to these SCUs must be recompiled. This is mitigated by the fact that AT entries for deleted SCU can be reused by other SCUs (this is further discussed in chapter 4).

3.9.2 Management of the Segment

When the compiler first compiles a module, the object code is placed at the end of the specified segment; the compiler also fills in the AT entry with the address of the code in the segment. When the compiler is about to recompile a module, the old version is destroyed. The compiler may move routines after the old version forward, recovering the space occupied by the old routine. This is called compressing the segment. The AT entry will be flagged to indicate that the old version is deleted in event of a system crash. The new object code is then added to the end of the segment. This movement of object code requires that the addresses for all modules be known, so that they can be appropriately updated to reflect the new postions.

As each routine is moved forward the address in its corresponding AT entry is modified to reflect the new position of the object code in the segment. Since SCUs are called via their corresponding AT entry and not directly by their address in the segment, no outstanding references are affected by the movement of object code. However, the object code itself may contain addresses within the segment. For example, the address of a constant table, or the address of a contained routine that is not separately compileable and hence does not have an AT entry. These addresses must also be modified when the object code is moved.

With the conventional load modules all of the addresses within the load module must be known so that they can be relocated when the load module is loaded into real memory for execution. The location of these addresses is maintained in the RLD table of the load module. Similarly, in this system addresses must be known to facilitate compression of the object code in the segment. Therefore, it is necessary to maintain an analogue of the RLD table so that addresses can be modified during compression. The RLD table contains the addresses of routines in the segment and is stored as part of the object code for an SCU, instead of containing the locations of the addresses as in conventional systems.

The RLD table is located at the end of each SCU. Each entry of the RLD table is the actual address of a referenced item (module or constant). To access these addresses in the object code, the RLD table can be made accessible by loading a base register with its starting address. The addresses within the RLD table can then be used to reference constant tables or invoke local routines.

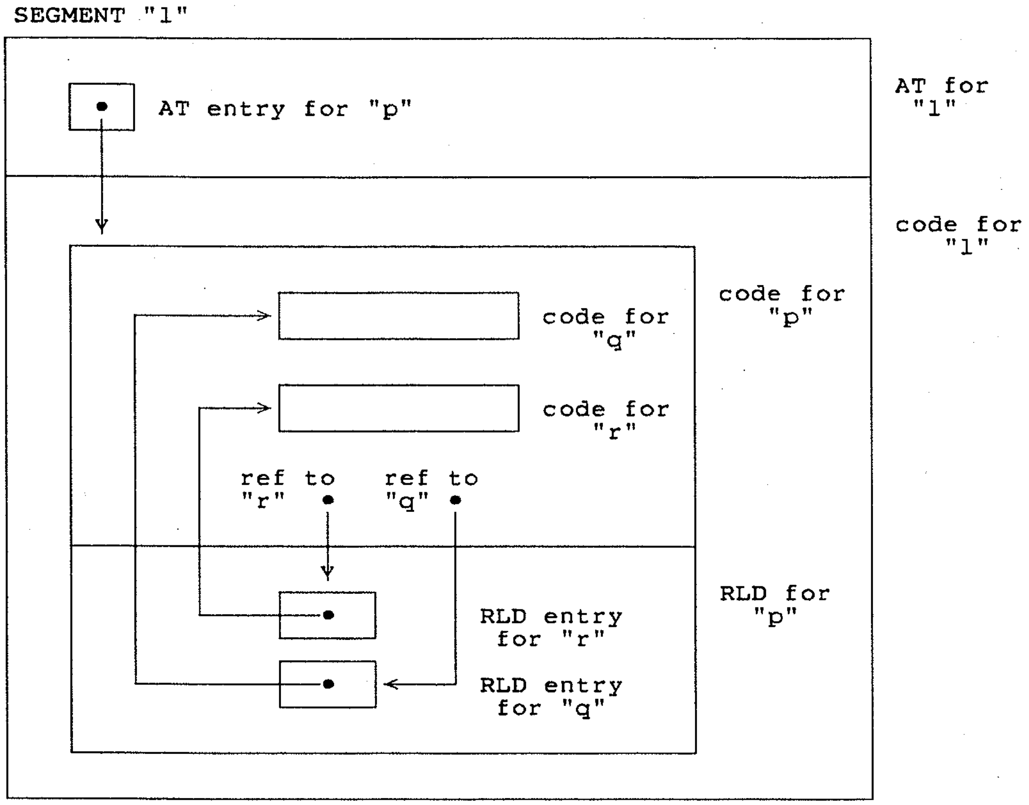
For example, the following declarations:

```
PROC p SINGLE 1
  PROC q( ... )
    :
  END PROC

  PROC r( ... )
    :
  END PROC

  q( ... )
  r( ... )
END PROC
```

would produce the following structure in segment "l":



Since procedures "q" and "r" are not separately compileable, they do not have AT entries. All references to "q" and "r" in the object code for "p" (and in the object code for "q" and "r" if they reference each other) actually refer to the RLD entries for "q" and "r" respectively. The RLD entry contains the address of the object code for procedures "q" and "r". It is these addresses in the RLD entries (along with the addresses in the AT) that must be updated during compression. This problem may be mitigated on machines that have program-relative addresses; for them, there would be few or no entries in the RLD table.

In the case of OM's there must also be an ESD to indicate all other COPIED routines referenced by this SCU. This information is used to extract other COPIED modules from the same or different segments as needed. When the OM is copied into a segment, an AT entry is created for it if none exists already. The compiler then fills in the appropriate RLD entries to inter-connect the copied routines. The ESD indicates which RLD entry is used for any particular entity. This linking is necessary because the address of the COPIED module cannot be known at compile time since there are multiple copies of it.

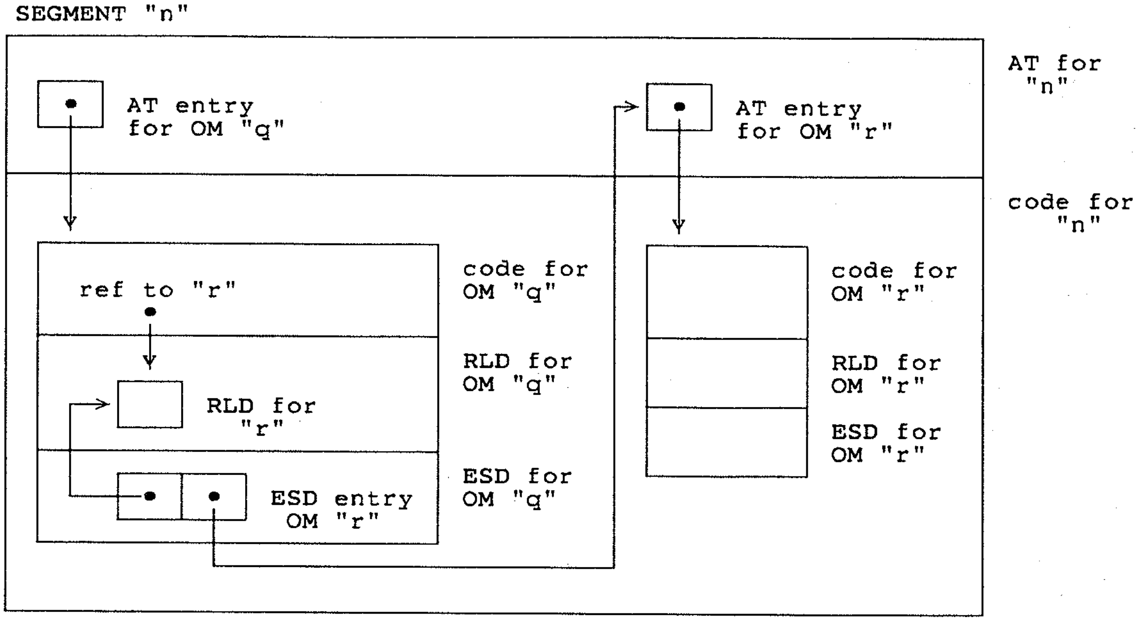
A COPIED module is unusual because it is never called from a different segment since any segment that references the module has its own copy. However, an AT entry is necessary for the executable copy of the OM so that the compiler is able to determine if a copied module already appears in the segment. So in many regards the copied module is like a contained routine that is not separately compileable. In particular, RLD references can point directly at the COPIED module, and hence it is referenced in the object code in the same manner. However, because a copied module is not logically contained in the referencing module and because it is an SCU, it must be handled differently during compression. Since the copied module is not contained in a referencing module, it may or may not move during compression. Thus, a RLD reference to a copied module may or may not have to be modified during compression, unlike RLD references to contained routines which always are modified.

If the COPIED module precedes the deleted module then during compression it is not moved and hence any RLD pointers need not be modified. If the COPIED module follows the deleted module then all RLD pointers to it must be modified. Therefore, by comparing the old RLD pointer value with the position of the deleted module it is possible to tell if it must be modified. This algorithm works for the simple case of RLD's pointing to constant tables or containing routines, as well. If the SCU they are associated with precedes the deleted module, the module is not moved; hence, the RLD values remain the same. If the module follows the deleted module, the module is moved and the RLD values must be modified.

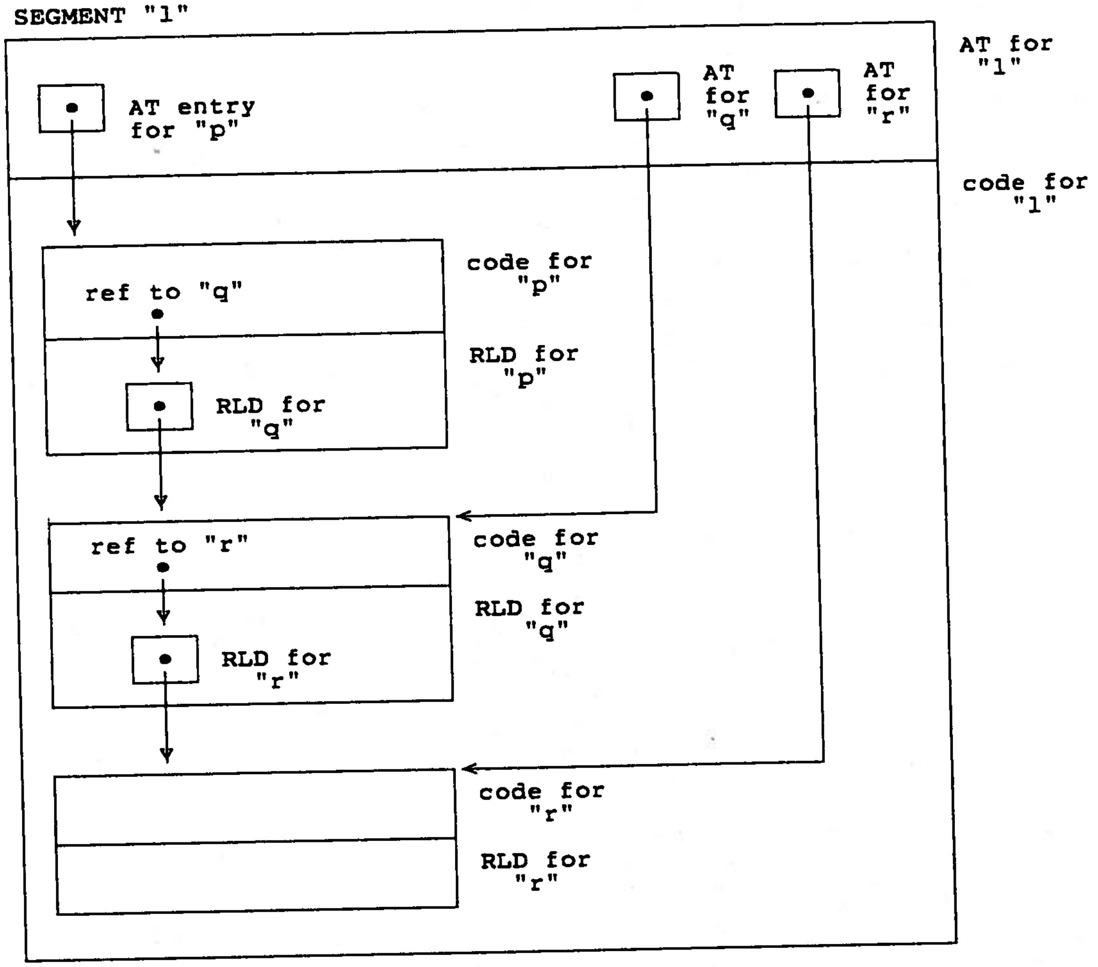
For example, the following declarations:

```
PROC p SINGLE 1
  PROC q COPIED n
    PROC r COPIED *
      :
    END PROC
    r()
  END PROC
  q()
END PROC
```

would produce the following in segments "n" and "l" when the routines are compiled:



The AT entries for OM's are distinguished from other AT entries. Thus, it is possible to allow the OM to be copied into the same segment and yet be able to distinguish the OM from its executable copy. The reference in "q" to "r" causes an ESD entry to be associated with the object code for "q" so that "r" will be copied when "q" is copied. The RLD entry for "r" is not filled in until "q" and "r" are copied into a referencing segment. "q's" ESD entry for "r" points at the RLD entry so that it can be filled in when "r" is copied along with "q". As well, the ESD entry points to the OM "r". This pointer can point to an OM in the same or a different segment.



Here there are AT entries created for modules "p", "q", and "r". The entry for "p" is created when the compile type is defined for "p". The entries for "q" and "r" are created when "p" is initially compiled. The RLD entries for the copied modules are filled in during compilation when modules "q" and "r" are copied from segment "n" into segment "1". Notice that if module "p" was recompiled, modules "q" and "r" would be moved up and the new object code for "p" would be placed at the end of the segment; in this case, the RLD address from "q" to "r" and from "p" to "q" would be modified. However, if "r" was recompiled, "p" and "q" are not moved, and hence the RLD addresses need not be modified.

3.9.3 Structure of the AT entries

Each AT entry contains the following information for each SCU:

1. address and length of the object code in the segment
2. address and length of the RLD table
3. address of the ST entry for this SCU

   This is necessary for the debugger to allow symbolic debugging, for the compiler to check if an OM is already contained in a segment, and for deferred object resolution.
4. Flags - SINGLE or COPIED or OM

Several more fields are needed to facilitate consistency checking; these fields will be discussed in chapter 4.

3.9.4 Transfer Table

To implement prototypes requires that all components in the type definitions based on the prototype be at the same fixed locations. This allows references to be compiled and these references will be the same for any of the particular type definitions derived from the prototype. Variables can be laid out in the same location of the local DA for all type definitions. Other variables that are added in the type definitions are assigned locations after these. This is done by having the compiler allocate displacements for the variables in the prototype by compiling the prototype. When the prototype ST information is used in a type definition, these displacements are retained. However, in the case of modules, this is a problem since the size of the object code for each module will vary from type definition to type definition based on the same prototype.

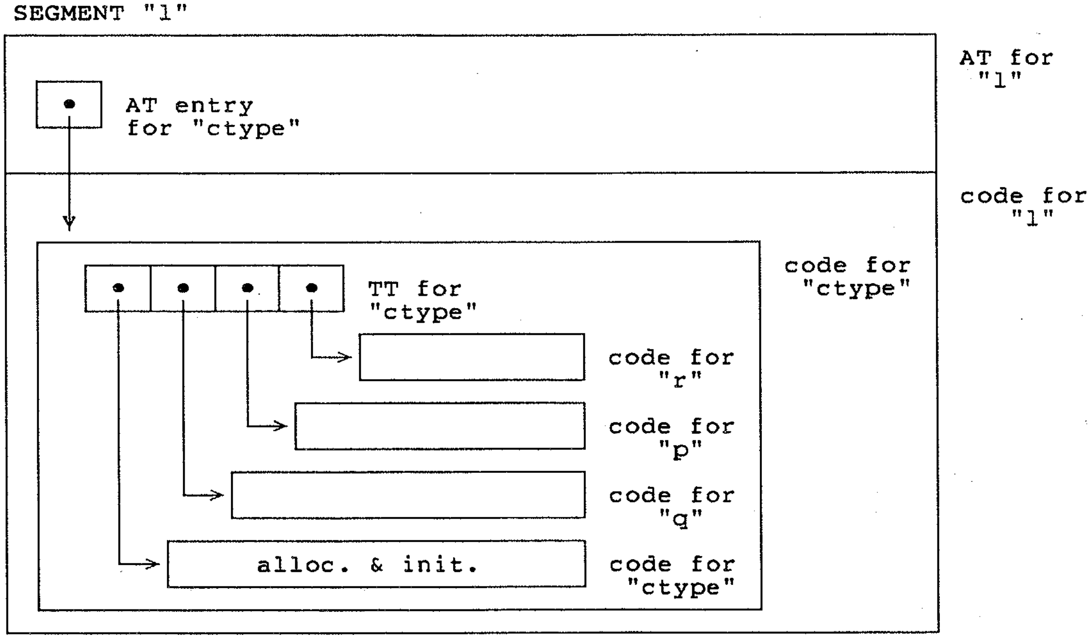
To make it possible to assign fixed locations to procedures within a prototype, a table called a transfer table (TT) must be introduced for each type that is dependent on a prototype. The TT is included with the object code for the type definition, and not in the AT. The TT like the AT provides a level of indirection so that modules of the type definition can be assigned fixed addresses and yet may vary in position from type definition to type definition.

The TT has an entry for each module in the prototype. That is, each entry points to the module's object code in the SCU. As with the variables in a prototype, the compiler can assign displacements to all the modules in the prototype, and once assigned, they are not changed. These displacements are the positions of the TT entries for each routine. Each type definition uses the same TT layout as the type definition's prototype. For example,

```
CLASS PROTOTYPE cproto( ... )
  PROC PROTOTYPE p( ... )
  PROC PROTOTYPE q( ... )
END CLASS PROTOTYPE

CLASS ctype( ... ) : cproto SINGLE l
  PROC p( ... )
    :
  PROC q( ... )
    :
  PROC r( ... )
    :
END CLASS
```

produces the following data structure in segment "l":



The first 3 entries (for "ctype", "p", and "q") in the TT are controlled by the prototype. The last TT entry (for "r") is an addition for the superset type "ctype". Because the TT cannot be made to conform to two different layouts, it is not possible to have a type dependent on more than one prototype (or one prototype dependent on more than one prototype). It would not be possible to organize the TT so that it could be passed to a context for either prototype as both situations would expect their TT elements to be at the beginning of the TT table.

In the case where a routine such as "p" is an SCU, the TT entry for "p" still points at object code in "ctype". However, this object code merely calls "p" wherever it is. This handles the situation where one type definition may have routine "p" as an SCU and another may not. Hence, the reference is independent of the number of secondary segment references. For example,

```
OBJECT PROTOTYPE cproto SINGLE l
  PROC PROTOTYPE p( ... )
  PROC q( ... )
    p( ... )
  END PROC
END OBJECT

OBJECT ctype : cproto SINGLE m
  PROC p( ... )
    :
  END PROC
END OBJECT
```

In this example, "cproto" is compiled and the object code for the actual routine "q" is entered into segment "l". However, the compiler cannot resolve the reference to routine "p" in "q" as the object code for "p" will appear in an unknown segment. This situation must be dealt with in a special way.

To make a call from a prototype routine to a routine in an object (or memory) that is derived from it, it is necessary that the segment containing the object code for the OBJECT definition and its address in the segment be readily accessible at execution. This could be done by passing the address of the current object to the prototype to be used for calls to actual modules. With the object segment known, the call from the prototype back to the object can occur.

As well, if a call is made to routine "q" from an instance of "ctype" or from within the definition of "ctype", then the compiler must use the inter-segment address of "q" to perform the call instead of the TT entry in "ctype". To avoid having to compile the inter-segment address of "q" into all such calls, the compiler will create a dummy routine for "q" in "ctype" and it is this routine that is referred to by the TT entry for "q". This dummy routine performs the actual inter-segment call to "q". Thus, a call to "q" simply uses "q's" TT entry in "ctype" to invoke the dummy routine for "q".

This mechanism is used to handle calls from the capsule of a memory definition to actual routines in the prototype for the disk memory that uses the capsule. For example,

```
MEMORY PROTOTYPE dirmemtype
  OBJECT PROTOTYPE dircap ...
    :
  MEMORY seqmem ... CREATES dircap
    :
  OBJECT seqfile ... CREATES seqmem
    :
END OBJECT

MEMORY vtocmem : dirmemtype
  OBJECT dircap
    :
```

forces the same definition for "seqmem" and "seqfile" no matter what type of devices they are stored on. Here, the allocation of "seqfile" in an instance of "vtocmem" causes a call via the CREATES clauses to "seqfile", which then calls "seqmem", which then calls "dircap" in "vtocmem". Thus, there is a call from the prototype routine "seqmem" to the object instance routine "dircap" in "vtocmem".

3.9.5 Externally Callable Programs

Programs that are able to be called from another segment are called externally callable (XC). The calling segment contains the address of the module and this address must be independent of all compilation activity in the called segment. This is why all SCUs are accessed indirectly through AT entries. However, some modules are not SCUs but are still externally callable. For example:

```
OBJECT p SINGLE 1
  PROC q
    :
  PROC r
    :
END OBJECT
```

procedures "q" and "r" are both available to be called from outside the segment via the inter-segment call mechanism, or from within the segment via the standard call mechanism. However, "q" and "r" are not SCUs, and hence would not have an AT entry. To handle this situation these modules will have entries in the TT for "p" instead of RLD entries in the RLD table of "p". This allows the address to remain fixed for references and yet allows the object code to move with the code for the containing SCU.

Clearly, the compiler must be able to determine all XC modules, because the object code that is generated for a reference is different. The procedure to determine if a module is XC and hence requires a TT entry is as follows. Start at the module and proceed to each containing cluster definition in turn until:

1. the father is a procedure definition or has compile-type INLINE

   Since it is not possible to refer to internal procedures of a procedure except from within the procedure, any module contained in a procedure cannot be XC. Conversely, objects and classes allow access to inner levels through instances of containing entities; hence, it is necessary to check outside of these entities.

2. the father has a compile type of SINGLE or COPIED.

   If the father is not a procedure definition then the module is XC. That is, the entity is accessible via an environment or object or class instance. In this case, the segment specified by the compile type indicates where the AT entry for the module will appear.

This procedure is done by the PE as part of the creation of a new module in the ST and it marks the ST entry appropriately for the compiler.

3.10 AT FOR OBJECTS IN MEMORIES

For memories that contain multiple objects it will be necessary to maintain a list of pointers to all the objects allocated in the memory. This list is the equivalent of the AT for a segment and is necessary for the same reasons that the AT is necessary for SCUs in a segment. In particular, it allows the objects in the memory to be moved for storage management reasons, and yet still maintain fixed addresses. As well, the memory writer may want to associate other information, such as security information, with the object; since this information is determined dynamically it cannot be stored in the object's DA. This information would be kept in the AT entry for that object.

Unlike AT's for segments which must all have the same structure, each memory containing objects can define its own AT structure. For example, in the definition of "vtocmem" the AT was a simple array that pointed to all capsules allocated in a VTOC. In the case of "seqfile" there is only one object in the memory and hence the AT can be very simple.

3.11 WORKING IN SEM

The environment provides the basis for interaction with the machine. The ability to define new variables and cluster definitions within the environment allows a user to create persistent data items and modules. The ability to dynamically execute programming language statements in the environment allows the user to test newly written modules and use existing modules to perform operations on environment variables.

3.11.1 Signing on to the System

It is imagined that a process is started for all input ports to the machine, after the system is loaded. If a port is inactive the task associated with it is quiescent. When a terminal is connected to the port, the task is activated. This is the same scheme used in UNIX[3] The terminal task has associated with it a segment that is declared inside the task. It is into this segment that the interactive programming language statements will be compiled. On activation of the terminal task, the terminal task invokes the compiler which reads statements from the terminal. The user types in a call to a logon routine and passes his user instance as a parameter:

```
logon(buhr)
```

The call is compiled at a particular location in the ST hierarchy (i.e. where the terminal tasks are located) and each user is probably declared in this environment; thus, the user name need not be qualified. The object code for the call is compiled into the segment associated with the terminal and subsequently executed. By having only this one module (i.e. "logon") in the ST for the terminal task and not allowing the user to enter the PE at this point, there is little that can be done here except sign-on.

---

[3] UNIX is a Trademark of AT&T Bell Laboratories.

The logon routine creates an access class for the user, establishes accounting information, and calls a routine called "signon" in the access class. Because the "user" is probably a process, the calling of the "signon" routine re-activates the task that was created when the user was declared. This routine allocates local variables and executes any code in the procedure. Normally there will be a statement at the end of his initialization code to call the compiler, at this point passing the terminal as input, the current ST context for compiling interactive statements, and the terminal segment in which the compiled object code is put. The compiler then reads statements from the terminal and compiles them as if they were entered in the context of the "signon" procedure in the user's environment.

At sign off, the compiler returns and the remainder of the "signon" routine is executed. The routine "signon" then returns, logon finishes and releases all the object code from the terminal segment, and the compiler at the terminal level is ready to sign-on another user.

3.11.2 Interactive Invocation of the Compiler

The need to invoke the compiler interactively from the "signon" routine is not an isolated situation or done just to achieve SEM. Invoking the compiler interactively and passing it an input file, a ST context, and a segment is how modules are compiled in PDM. The only difference between that and SEM is that the compiler expects individual statements in SEM and that the compiled object code is executed immediately. This facility can be used for many different purposes. Invoking the compiler interactively can be used as a method of module debugging. Rather than putting print statements at strategic points in a module to determine what is going on, it would be possible to invoke the compiler at these points. When the compiler is invoked, the user would be able to enter interactive statements that are compiled in that context. This allows printing out and/or changing variables that are visible in the current static context. When the interactive invocation of the compiler is terminated, the module will resume after the call to the compiler.

As well, this facility is needed by the debugger, which will allow dynamic setting of breakpoints in a module. When these breakpoints are encountered during execution, the debugger can invoke the compiler to interrogate and/or change the module DA.

3.11.3 Transient DA for Access Variables

A declaration such as:

VAR pz : ^sys.zarnke.useracc

made at the top level of user "buhr" would result in the storage for "pz" being allocated in buhr's environment DA, along with the creation of an access class to access environment "zarnke". When the user signs off, the access class is deleted when the DA it appears in disappears. Since the environment does not disappear, the access variable would remain outstanding. Hence, the access variable "pz" would only be valid during the terminal session in which it is created. Attempting to use "pz" during another terminal session would result in an error.

If the user always wants to access environment "zarnke", then it is necessary to declare "pz" in a context that will create "pz" at each sign-on. This will cause the creation of an access class at that time. A context associated with the environment that can provide this type of context is the environment access class. The user can make declarations in the environment access class by locating to the environment access class definition using the PE and then making the appropriate declaration. These declarations will be executed when the user re-enters SEM and references can be made to them in executable statements. Note that this is possible because each environment has its own ST and hence any declarations in the access class definition of an environment are unique to that particular user.

When the user signs off or ceases accessing the environment, the access class is released and so are all of its locally declared variables such as "pz". When the access class is subsequently re-created such as at sign-on, all the locally declared variables are re-created. This includes simple variables as well as access variables. Hence, the user can continue to directly access entities in environment "zarnke" through "pz" until the declaration of "pz" is removed from the environment access class definition.

While this scheme seems to provide the necessary facilities for variables that must be allocated in transient storage, there is one problem. Users who wish to access another user's environment (which contains all his files) do not want to have all of these extraneous access variables created when an access class for the environment is created. Hence, the environment access class definition is not an appropriate place for these transient variables.

To solve this problem the user can declare all transient variables in the "signon" routine instead of the "user" access class. This works because the "signon" routine is called when the user signs on, hence any transient variables declared within it will be created and be accessible from SEM. However, "signon" is not called when another user accesses the environment.

3.11.4 Statement Execution Mode

When a user signs on, he is placed in SEM for his particular environment. In SEM the user may enter executable statements, and these statements are compiled into the terminal segment, and executed immediately. The statements entered may refer to the variables declared in the active sign-on environment or to other environments via appropriate qualification.

The following is an example of an interactive session:

```
I: i <- 1
I: output.int(i)
     1
I: LOOP
   UNTIL i > 5 EXIT
     IF i ¬= 3 THEN
        output.int(i)
     END IF
     i +<- 1
   END LOOP
     1
     2
     4
     5
I:
```

The compiler accepts one executable statement at a time after the prompt "I:". A statement may be a control structure, as the loop above illustrates; it must be entered in its entirety before it is executed. If a complex series of operation are desired, the code can be made into a procedure in the user's environment which can be edited and compiled. This procedure can then be invoked dynamically.

3.11.5 Entering the PE

The PE can be considered as a procedure in the definition of a user. Thus, the PE has access to the environment ST. Since the PE is a routine in the environment, it can be invoked in SEM by typing:

```
I: edit
```

but this may be handled specially because edit is really a procedure associated with the ST. The PE then reads from the terminal until it is ended by the user. This operation might also be done by switching windows or creating a new window on a full screen editing system. In either case it involves a mode switch between SEM and PDM.

3.11.6 Moving Around in the Environment

A user must be able to position himself to other environments in the _environment hierarchy_ (a subset of the DS hierarchy). This would allow a user to enter executable statements in other environments and to enter the PE to modify these environments. Thus, some technique is necessary to allow moving from one environment to another.

To move in the environment hierarchy requires changing the current environment in which executable statements are compiled and executed. To do this the user can invoke a "locate" routine in the another user's environment:

```
I: ^sys.zarnke.useracc.locate
```

This would create an access class for "zarnke" and invoke routine "locate" in the access class. The "locate" routine would simply invoke the compiler interactively, so that subsequent interactive statements are compiled using the new environment ST context, and so that subsequent execution uses the new environment DA. Terminating the compiler would terminate the "locate" routine and the user would return to his original environment.

3.11.7 Signing Off the System

A user should be able to sign off the system in 3 different ways. A user can terminate or suspend or continue a session at sign off.

1. Termination means that all transient active tasks are terminated and all transient DAs are released.
2. Suspension means that all tasks are suspended until the user logs back on to the system, but the transient DAs for the suspended tasks are retained. The retension of the transient DAs allows the tasks to resume exactly where they left off when the user suspended his session.
3. Continue means that all tasks continue to run even after the user has disconnected his terminal from the system.

Termination can be handled simply by signalling end-of-file to the compiler. However, both suspension and continuation will likely be done by calls to routines in the terminal task. For example:

```
^systemem.termtask.suspension
^systemem.termtask.continue
```

This deferred reference calls the appropriate routine in the terminal task to cause suspension or continuation and then disconnects the terminal.

3.11.8 Example User Session

The following user session demonstrates the use of the PE and SEM. In this session the definition of "seqfile" is augmented by introducing a new routine in "seqacc" which returns the number of records currently in the file. In the process of adding the new routine, the existing definition of "seqfile" must be augmented as well.

```
I: logon(buhr)                  ¢ sign on
   LOCATED TO BUHR @ 11:15 WED. 14, 1984
I: edit                         ¢ enter the PE
E: locate seqfile               -- position to LST for object "seqfile"
E: var no_of_rec : INT          -- declare record counter
E: list seqfile'initialization
   first_record <- last_record <- NIL
E: append seqfile'initialization
   no_of_rec <- 0               ¢ set counter to zero at creation
E: locate write                 -- position to LST for "write" in "seqfile"
E: list write'initialization
   last_record <- sm.write(r)
   IF first_record = NIL THEN
     first_record <- last_record
   END IF
E: append write'initialization
   no_of_rec +<- 1              ¢ count each new record
E: locate end                   -- end locate to "write"
E: locate seqacc                -- position to LST for "seqacc"
E: proc size                    -- declare new procedure
E: locate size                  -- position to LST for new proc "size"
E: returns norec : INT          -- declare return value
E: append size'initialization   -- insert source text
   norec <- no_of_rec           ¢ return number of records
E: locate end                   -- end locate to "size"
E: compile seqfile              -- re-compile "seqfile"
   NO ERRORS DETECTED
E: locate end                   -- end locate to "seqfile"
E: var f : seqfile(INT)         -- declare test file in environment "buhr"
E: var pf : f.seqacc            -- declare access class for test file
E: var i, nr : INT              -- declare sundry variables
E: locate end                   -- go back to SEM
I: nr <- pf.size                ¢ check current size of test file
I: output.int(nr)               ¢ no records just after declaration
       0
I: FOR i <- 1 TO 100            ¢ load file with 100 records
   LOOP
     pf.write(i)
   END LOOP
I: nr <- pf.size                ¢ check current size of test file
I: output.int(nr)               ¢ 100 records after loading
     100
I: signal end_of_file           ¢ sign off
   LOCATE END TO BUHR @ 11:45 WED. 14, 1984
```

While this method of modification may appear tedious for a command line system, facilities like pop-up menus, a mouse, and full-screen editing would remove much of the tedium.

3.12 SUMMARY

The introduction of the programming language construct the ENVIR allows the programming language to be used interactively. This brings all constructs of the programming language, including control structures and data types, to the interactive level. Hence, the conventional command language has been eliminated.

As well, maintaining the ST information for all declarations allows type checking of all references throughout the system at both the user and the system level. The maintenance of the ST by the PE guarantees the integrity of this information and introduces a new style of programming that is significantly different from that introduced using a parse-tree editor.

Finally, file declaration and use is now available throughout the system both in SEM and in PDM. Thus, files that the user wishes to persist can be declared as variables in the environment.

Chapter IV

ADDRESSING

A processor/CPU spends most of its time addressing data: both in the hardware when instruction addresses are evaluated, and in the software where a large proportion of calculations and movement of data has to do with calculating or finding the address of data to be processed. These calculations involve many different types of addresses: for example, manipulating real memory addresses, or virtual memory addresses, or I/O device addresses, or the address of data on the I/O device, or network addresses. Traditionally, all of these addresses have significantly different forms and must be handled by different hardware or software routines. Normally, the user deals strictly with symbolic names, and hence is not aware of these different forms and their manipulation because the translation from symbolic name to internal address is performed for him by the system.

Some of the different address forms are the result of the physical characteristics of the hardware devices, and some are the result of the software structure that is imposed by the programming system (for example, the DS hierarchy). While the hardware is cast-in-stone to a large degree, the software address forms can be designed to provide a consistent approach to describing where data is located. This approach implies the ability to process an address without having to treat any part of the address in a way substantially different from the other parts. This type of addressing is called a <u>uniform addressing scheme</u>. This chapter will present a particular design for such a uniform scheme given the considerations necessary to implement the ideas presented in chapters 2 and 3.

It is desirable to have this consistent approach to addressing in order to cover the broad spectrum of addressing needs. Work has been done using a large uniform address to assign unique addresses to all objects in the system [MYERS80], thus providing a simple consistent method of accessing information no matter where it is physically located in the hardware structure of the machine. While this eliminates many different address forms, it produces a number of undesirable effects. First, the uniform address space cannot expand to handle increased addressing capacity. Secondly, the length of the uniform address must be large enough to handle the largest possible file/data area. A consequence of selecting this maximum size is the fixing of the size of the address used in pointers. Lastly, there is the problem, with a uniform addressing scheme, of trying to keep track of unused addresses so that they can be subsequently used.

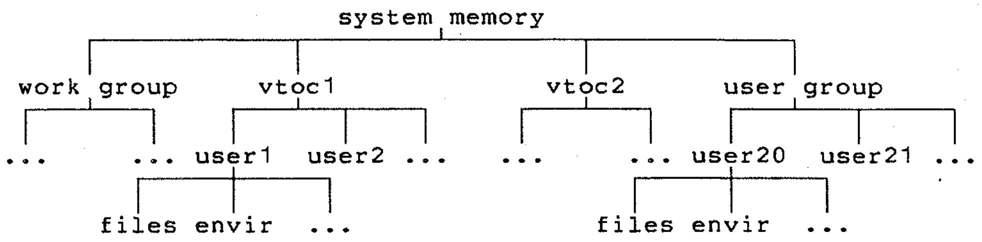
4.1 STRUCTURED ADDRESSING SCHEME

An addressing scheme is proposed that will provide a consistent addressing structure for the system, and yet not have the drawbacks of a uniform addressing scheme. Such a scheme is called a structured addressing scheme. A structured addressing scheme will provide a consistent addressing method that covers addressing all of the diverse parts of the programming system. It must be able to accommodate system growth and yet this expansion must not cause existing addresses to become invalid. This structured address is like a qualified name in a programming language (eg. X.Y.Z); the structured address is the internal analogue.

4.2 MEMORY HIERARCHY

The memory hierarchy (MH) is formed from the persistent declarations in the system. The root of the hierarchy is the system memory or system DS (or nucleus) in which the declarations for all persisting storage areas are made. These persisting storage areas are further subdivided. A subdivision does not have to correspond with physical storage; it may be a logical division used for administrative reasons. A memory such as "seqmem" is a leaf memory in the MH as it is not designed to accommodate other memories.

For example, a particular system's MH might look like this:



where a group is a logical subdivision of the memory. For example, a particular group may be one or more physical volumes which collectively define a single logical memory. The installation controls the declaration of the entities that form the upper portions of the structure. The user may or may not be able to add new levels to the lower portion of the structure.

Notice that the memory hierarchy does not have a large number of levels, and that it is independent of the ST and DS hierarchies that show the logical structure of cluster definitions and instances. It is this structure that forms the basis for formation of the addresses that reference data throughout the system. These addresses are called <u>MH addresses</u>.

4.2.1 <u>MH Definition</u>

The kind of MH structure that can be constructed by an installation depends on the definitions that are created for the structure by the system designer. For example, does the system need the ability to allocate a "seqfile" or a user in the "systemem", or are only disks allocated in "systemem"? Can a disk contain a "seqfile" in it that does not belong to a user? Can a user allocate another user or a disk memory within his own memory? These decisions must be made by the system designer and the definitions for the MH must be organized to provide the type of MH structure that is desired.

The MH definition can be structured in several ways, two of which are the following:

1. At each level an explicit definition is made to allow each particular type of entity to be declared at that level. For example:

```
MEMORY systemem
  :
  MEMORY PROTOTYPE dirmemtype
    MEMORY seqmem ...
      :
    OBJECT seqfile ...
      :
    other file types
      :
  ENVIR PROTOTYPE usertype : dirmemtype
    :

  ENVIR user : usertype
    :
  MEMORY vtocmem : dirmemtype
    MEMORY user : user_volumetype
      :
```

If the system "sys" is of type "systemem", this would allow the declaration of a "vtocmem" and "user" in "systemem":

```
VAR vtoc1 : sys.vtocmem
VAR buhr : sys.user
```

and "seqfile" and other file types in a "user":

```
VAR f : buhr.seqfile( ... )
```

No other combinations are allowed by this definitional structure, such as declaring a "seqfile" in "systemem".

2. A definition which allows all types of memories to be allocated within other memories (such as a volume). Such a definition is necessarily recursive relative to the types. For example,

```
MEMORY PROTOTYPE dirmemtype
  ENVIR user : dirmemtype
    :
  MEMORY vtocmem : dirmemtype
    :
  MEMORY seqmem
    :
  OBJECT seqfile
    :
  other file types
    :

MEMORY systemem : dirmemtype
  :
```

this would allow the declaration of a "vtocmem", a "user", a "seqfile" and all other file types in "systemem":

```
VAR vtoc1 : sys.vtocmem
VAR buhr : sys.user
VAR f : sys.seqfile( ... )
```

and all other types in a "user" and "vtocmem", as well, hence, allowing the declaration of any object at any level.

Only the form of the MH is chosen by the system designer; each installation is free to structure the MH structure following this format.

4.3 FORMATION OF A MH ADDRESS

All entities in the system from simple integers to memories must have a MH address. However, most entities in the system are addressed relative to a memory that is known to be already accessible. This is reflected in the software by the need to explicitly access objects and memories, and in the hardware where all instructions refer to data relative to the beginning of a paged memory. As a result, most entities are in fact accessed with an address that is not absolute within the programming system but relative to a specific memory.

However, there are times when the address of an entity must be passed outside of the containing memory. In this case, an address must be formed that is relative to some higher-level entity. For example, a MH address must be formed for the declaration of an object such as:

```
VAR f : vtoc1.seqfile( ... )
```

Here, a "seqfile" is allocated in memory "vtoc1" and the MH address of the new file is placed in variable "f". Since the storage for variable "f" is not necessarily in the same memory as the storage for the file itself, the address of the file cannot be relative to the current memory.

4.3.1 Component Value

As was seen in chapter 2, a memory might accommodate only a single object (sequential file) or several objects (variable length strings or multi-precise values) or it may accommodate other memories (volume). Thus, there exists a nesting of objects and memories. Each of these nested items is addressed relative to the beginning of its containing memory. The series of displacements that locate an item in the nested storage areas serve as an address by which that item can be uniquely located. These displacements are called component values.

A component value is generated from a call to the allocation routine in a memory. The declaration of an object causes the compilation of a call to the "allocate" routine in the memory that is to contain the object. The value returned from "allocate" is the "address" of that object in the containing memory.

The value returned from "allocate" does not have to be the actual address of the item in the memory. For example, in the case of allocating a capsule in a VTOC, the value returned may be the subscript of the VTOC directory entry allocated for a capsule (i.e. the AT entry allocated for the capsule). Thus, only this smaller amount of information, which is an encoded form of the location of the object, must be stored in the address. This is an important point, as it means that a component value need not depend on the size of the object's memory that it addresses.

The address that is created for "f" is a structured address. The components of the structured address correspond to the levels of the MH. Each component value identifies the next lower-level object in the address, and the last component identifies the actual object to be referenced. (In the case of a reference to a module, there are two final components: one for the AT entry and the other for the TT entry.) The declaration of "f" involves calling the "allocate" routine for several memories. In particular, notice that "seqmem" returns a component value (i.e. a single bit) even though there is only one object contained within it. This value is necessary as its presence indicates a level in the MH.

4.3.2 Component Size

As was discussed in chapter 3, the AT can be expanded dynamically by moving the object code within the segment to make room for a larger AT. This storage management technique also applies to the capsules in the VTOC if it should fill up. The blocks for the object, immediately after the VTOC, can be moved to allow expansion of the VTOC.

If the structure of the segment and VTOC allow the size to be dynamically increase, the component size may have to increase correspondingly. If the component size for a segment or VTOC is fixed in size, then that fixes the maximum number of objects in the memory. Hence, it does little good to be able to expand an AT or VTOC if the new space cannot be addressed.

One solution is to fix the component size at a large value, but this makes largely unnecessary the encoding of the address to obtain a smaller MH address. For this reason, a length field is associated with each component making the component fields variable length. Thus, as the DA expands, this allows the component values of a new address to be larger than corresponding components of older addresses. Clearly, this approach has simply transferred the fixing of the size of an entity to fixing the size of the component length field. Now, the length field is the limiting factor on the size of an addressed entity. The maximum number of bits that can be specified in the length field limits the maximum size of a component.

However, choosing a length field of 4 or 5 bits allows a 16 to 32 bit component size. Since these numbers are large, the fixed-sized length field does not impose as much of a restriction as a fixed-sized component. And, if the average size of a component plus the length field is less than that of a fixed-sized component, then this approach will achieve a reduction in the average size of a MH address.

Once this scheme is selected, it is necessary for memories to supply a routine to decode the encoded address returned by "allocate". Thus, each memory must contain a routine like the following:

```
PROC adrtrans(component : COMPVAL) RETURNS a : ADDRESS
  a <- decode component to the virtual address of the object
END PROC
```

This routine is passed the component value and returns a virtual address of the desired object within the memory. This routine would be called as part of de-referencing to actually locate an object or memory in a memory (eg. a capsule entry in a segment or VTOC). To enforce the inclusion of the routines "allocate", "free", and "adrtrans", a memory prototype is defined for the system. All memories that can accommodate other memories must use this memory prototype in their definition.

```
MEMORY PROTOTYPE basememtype
  CONST maxcomp : INT <- 16
  TYPE  COMPVAL = BIT(1..maxcomp) VARYING

  PROC allocate(s : POSINT) RETURNS c : COMPVAL
  PROC free(component : COMPVAL)
  PROC adrtrans(component : COMPVAL) RETURNS a : ADDRESS
END MEMORY
```

4.3.3 MH Address Size

Because of the above, the address for an entity in the MH is of varying length, both in the number of components and the size of each component. This presents a problem because the compiler cannot determine the amount of space to be allocated for a declaration, such as:

```
VAR f : ^user.seqfile( ... )
```

Here, the compiler does not know how much storage to allocate for (the pointer) "f". This would also be true if the programming language allowed a file variable that could refer to different "seqfiles", as in:

```
VAR f : REF seqfile( ... )
```

Here "f" could point at any "seqfile" file, and the MH address of each one may have a different size as they may be located at different nodes in the MH. Hence, the compiler would not know how much storage to allocate for a file pointer.

An acceptable solution to this difficulty is to fix a maximum size for a MH address. This maximum amount of storage is allocated for all variables that hold MH addresses (i.e. MH pointers). This maximum amount could be an installation-dependent value, depending on the number of levels in the MH and the size of the components in the address. I feel that a 64 to 80 bit maximum size for a MH address would provide an adequate size to accommodate a large MH without having unacceptably large pointers.

This fixed-sized address only constrains the MH address indirectly as the number of components or the size of each component is not restricted, only the overall size of the MH address. For example, a MH address could contain 6 components each of length 8, or 2 components each of length 32, etc.

4.4 RELATIVE ADDRESSES

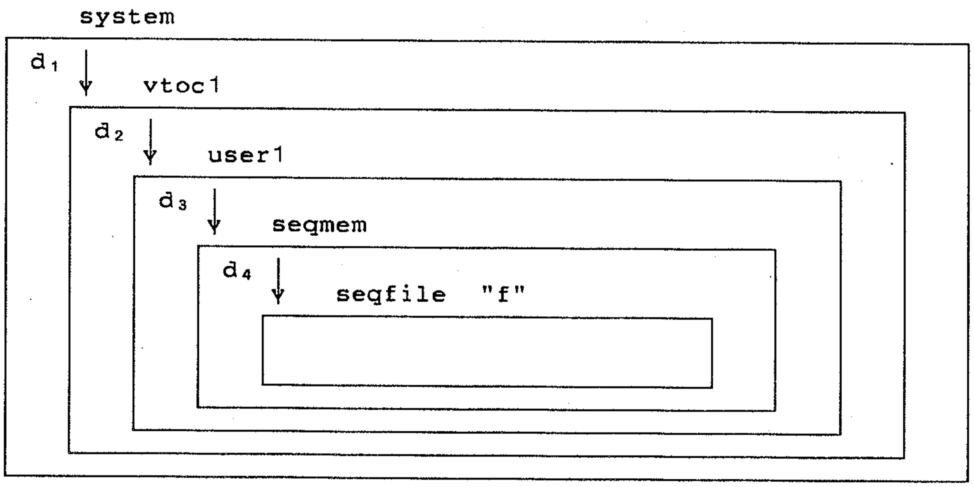
While the values returned by "allocate" form the downward path in the MH, there are several alternative starting points relative to which MH addresses may be formed. Two different schemes for the starting point of a MH address are discussed.

4.4.1 Absolute MH Address

In this scheme the MH address starts at the root of the MH, that is, the system DA. Hence, components that form the MH address extend from the root down to the particular entity. For example, given the MH above and the following declarations:

VAR f : user1.seqfile

the address of "f" will use the address of "user1" as the first part and to this will be added those components needed to address a "seqfile". The address will start at the MH root. The first component of the address for "f" is the address of "vtoc1". The second component is the address of "user1" in "vtoc1". The remaining components are the results of calls to the allocation routines of the VTOC and of "seqmem". These 4 components specify the address of "f" in the MH.

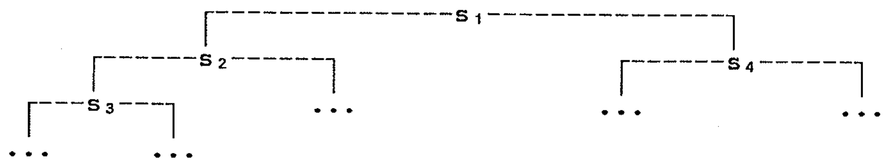


The picture is diagrammatical because d_1, d_2, d_3 and d_4 actually point at the AT entries which then point at the actual data area. As well, this diagram does not necessarily show the relation between the actual/physical storage devices and the memories.

An absolute MH address allows the MH to expand downwards. New levels can be added onto the leaves of the MH without affecting existing addresses. Since absolute MH addresses cannot refer to anything above the root, this prevents the MH from being embedded in a larger hierarchy later on. For example, one MH could not be embedded in a network of MHs.

4.4.2 <u>Subsystem-Relative MH Address</u>

An alternative approach to having one point in the MH from which all addresses start is to designate a few selected points in the MH as reference points for address construction purposes. This implies that all MH addresses generated under one of these reference points would start at this reference point. Hence, all the addresses and references under a reference point would be local and all addresses in this subbranch of the MH tree would be self-contained. This subbranch is called a <u>subsystem</u> and each subsystem is identified by a unique subsystem code. For example, in the following MH:



the dashed line denotes paths through several possible levels. An address formed at any particular point is made relative to the closest containing reference point (i.e. s_1, s_2, s_3, or s_4).

Subsystem addressing is implemented by having the unique subsystem code form the first component of every MH address. When the MH address is de-referenced, this code is used to identify a <u>subsystem entry</u> in the system that contains the downward pathway from the root to the subsystem. This downward pathway plus the remainder of the MH address forms the complete MH address.

There are several advantages of a subsystem-relative address. First, the subsystem addresses are physically smaller than those that begin from the root. Secondly, it is possible to restructure the MH by moving subsystems without affecting addresses in it or to it by simply modifying the address in the subsystem entry. Lastly, it is possible to expand the MH by adding new subsystems along any MH pathway by fixing up the appropriate subsystem entries for any subsystems below the new subsystem.

However, there are also several disadvantages in adopting this addressing scheme. First is the problem of specifying in the programming language that a particular declaration introduces a subsystem. Secondly, all the subsystem entries must be maintained in a directly accessible location so that the subsystem code can be used to find the start of the downward path. Lastly, there is the problem of assigning the unique system-wide codes for each subsystem.

As a result of these disadvantages the simpler absolute MH addresses are used in the remainder of the discussion on addressing. However, even with the problems inherent in the subsystem-relative address it may become necessary in the future to adopt them to deal with an expanding system.

4.5 MH ADDRESS RESOLUTION

MH address resolution is the process of using the MH address to traverse the MH to find a particular entity, and then make its memory addressable. Each component in the MH address identifies an object in a higher-level memory. Thus, in traversing the MH, objects in several memories may have to be made addressable, and hence several representatives may have to be created, one for each object. Its “address” in the memory is given by the appropriate component value in the MH address. The compiler knows which data items contain MH addresses and the context in which a MH address must be resolved before data in the referenced memory can be accessed. To this end, the compiler generates a call to a system routine to resolve the MH address. This routine takes the MH address and traverses the MH to the desired entity. This system routine is called the address resolution routine (ARR).

The first step in de-referencing a MH address is to start at the root of the MH. The ARR takes the first component of the MH address and decodes it by passing it to the "adrtrans" routine for system memory "sys". This is possible because both the segment and the DS for "sys" are always accessible (eg. are in real memory). The routine "adrtrans" returns the virtual address in the system DA of the object specified by the encoded component. The object is underlying the next memory that contains the AT entry. Hence, the above process can be repeated on the next component with the newly located memory. However, this memory may or may not be accessible (i.e. have a pager), and if not accessible it must be made accessible by the ARR to decode the next MH-address component.

The process of making the next entity accessible for the ARR involves creating a representative for an object or a representative for the underlying object of a memory. For components other than the last (or the last two components for a MH address of a segment) the representative will be a pager and is used to make the next memory accessible for the ARR. This representative must be allocated and accessed before proceeding to the next component. This is possible because object code for the capsule and its representative as well as the DA for the capsule are all accessible because the containing memory is accessible (the issue of how all the object code becomes accessible is discussed shortly).

Notice that the higher levels of the MH will normally remain active and hence they will already have representatives created. Thus, the first part of the MH address will likely be able to be resolved quickly. Only for the later components of the MH address will new representatives need to be created. All intermediate memories accessed along the MH pathway must remain accessible until the access is finished. This is because each memory is dependent on the one in which it is contained to be able to directly access its capsule and to allow access to the blocks of the overlying memory.

4.5.1 Active Object Tree

As has been mentioned already, only one representative is created for an object no matter how many accesses are made to it. To do this the system must have some way, during address resolution, of determining if an object is already being accessed. Thus, if a representative already exists, the resolution must use the existing representative.

This is accomplished by having an active object tree (AOT) maintained in the system DA. The AOT is a tree structure which contains the pathways to all active objects. The structure of this tree corresponds to the structure of the MH, but it is the small subset of the MH that is currently active. Each level in the tree represents a level in the MH. At each level, there is a list of all objects that are active and their associated representatives. The objects are identified by their component values. The size of the AOT is a function of the number of processes running in the system, the number of inter-AS calls made, and the number of accessed objects. While this may be in the hundreds, the amount of space required for the AOT is small in comparison to the space required for the representatives.

The resolution of a MH address may produce several entries in the AOT. However, at the module level, only the leaf entry will be necessary to access the data or module. Hence, only a single pointer is needed in a module to store the address of the representative.

An analogy can be drawn between the AOT, and the KST (Known Segment Table) and AST (Active Segment Table) in Multics. However, the symbolic names in the KST correspond with the components of a MH address.

4.5.2 Accessing Segments

Each segment is a memory, and hence must be made accessible before it can be accessed. This accessing will create a representative to page the object code in the segment that is accessed by the hardware. A segment is made accessible when the context in which it is declared is entered. In this object-oriented system, each object must be made accessible before any items in it can be accessed. This access makes accessible the DS for the object. It must also make accessible the segment that contains the object code to manipulate the DA. Thus, the segment is made available as a result of the access to the object. This is different from the Multics system which waits until the first reference to the object code occurs before it makes the segment containing the object code addressable.

To allow the accessing of the DA to implicitly access the object code for the DA, the MH address of the object code is associated with the DA (probably as a field in the capsule for the DA). Thus, the address for an instance of an object is just a pointer to its DA and this then points to its object code.

4.5.2.1 Secondary Segment Accessing

An object may contain several modules that are compiled in separate segments. Thus, when an object is accessed, not only must the segment containing the object's initialization code be made accessible, but also the segments for all nested modules whose code is in separate segments. To make possible this secondary segment accessing, the object code for the object must point to all secondary segments. As well, these segments must be made accessible implicitly as part of accessing the object.

4.6 DETAILS OF MH ADDRESS RESOLUTION, CREATION AND TERMINATION

The previous discussion outlined, in general, the steps involved in resolving a MH address. However, there are several details that were ignored in this introduction to MH address resolution, in particular, the detailed order in which resolution occurs and entities are allocated.

4.6.1 MH Address Resolution

A MH address can be de-referenced in two contexts: implicit and explicit access.

4.6.1.1 Implicit Access

Implicit access occurs when an object is referenced without using an access class. For example, if the visibility restrictions (discussed in chapter 5) allowed the following reference:

```
p <- f.first_record
```

where "f" is a "seqfile", then it would be necessary to de-reference the MH address for "f" creating all the appropriate representatives to allow access. After the access, these representatives would likely be deleted. Notice that an access class is not necessary here as the representatives make the DS accessible and are independent of the access class. Here the duration of such an access is short and the overhead for making the object accessible is relatively expensive, if many such accesses are performed. In some situations this type of access is necessary, such as accessing a variable in another user's environment. Creating an access class in this situation would allow references to entities only in the environment access class and not in the environment DA.

4.6.1.2 Explicit Access

Explicit access occurs when a MH address is de-referenced as a result of the creation of an access class, as in:

```
VAR pf : f.seqacc
```

Here the MH address of "f" is de-referenced, creating the appropriate representatives. This must be done before the initialization code for the access class is executed because the initialization code can refer to the variables in the file. As well, each access class for an object must point to its representative so that the representative remains allocated and hence the object is accessible for the duration of the access class. For example, "seqfile's" access class points at representative "seqrep" and the access class for "dircap" points at the representative "seqpager". Hence, the storage for the access classes must be allocated before de-referencing "f" so that the pointers to the representative can be placed in these data areas. This is not a problem since the storage for the access class is allocated on the heap. As well, all the access classes in the access chain are linked together according to the CREATES clauses. Hence, it is possible to access the pointer to the pager indirectly by the access class for "seqfile".

4.6.2 MH Address Creation and Termination

The following discussion details the operations that occur when a MH address is constructed for a declaration, such as:

```
VAR f : ^sys.vtoc1.seqfile( ... )
```

At compile time, the compiler looks up "vtoc1" in the containing ST called "sys". The compiler must then generate object code to traverse the containing scopes at execution time in order to find the DA corresponding to "sys", and using the displacement from the ST entry of "vtoc1", extract the MH address for "vtoc1" from its DA. As well, the compiler must generate object code to call the necessary routines to declare a "seqfile" and to store the resulting MH address in the DA containing "f". This involves using the displacement of the object code for "seqfile" (i.e. which TT entry) which is determined from the type of "vtoc1". This displacement will be used at execution time to call "seqfile".

At execution time, the MH address for "vtoc1" is located. This MH address is really the address of the capsule underlying "vtoc1". Since "vtoc1" is allocated in "sys", it would only contain a single component. That component is the value returned by "allocate" when "vtoc1" was declared in "sys". This MH address is de-referenced and a representative is found to exist for the volume that underlies "vtoc1" because the system created an access class for the volume when the operating system started up. Hence, the segment containing the volume's object code and the segment containing the representatives for the volume and any of its contained segments are accessible.

Once the object code that manipulates "vtoc1" is accessible, it is possible to call the object code for "seqfile" to begin allocation of the file. However, the object code for "seqfile" may reside in a segment different from that containing the object code for "vtocmem" (which is the type of "vtoc1"). Consequently, the segment containing the object code of "seqfile" must be made accessible. Yet, when processing the declaration above, the compiler generated only the displacement of the TT entry for "seqfile" in "vtocmem" and not the MH address of "seqfile's" object code. This displacement is used to call a dummy routine in "vtocmem" which in turn calls "seqfile". This dummy routine was generated because "vtocmem" is an instance of the prototype "dirmemtype" which contains the definition of "seqfile". It is the dummy routine for "seqfile" that contains the MH address of the object code for "seqfile". The dummy routine causes the MH address of "seqfile" to be resolved, and hence, causes the segment to become addressable.

The displacement of "seqfile" is now used to call the initialization code for "seqfile". This code calls the CREATES code in "seqmem", which calls the CREATES code in the appropriate capsule defined in "vtoc1". At the end of the CREATES chain, the "allocate" routine in "vtoc1" is called to get the storage for a capsule. The value returned from "allocate" will form part of the MH address for "f", and must be passed back up the CREATES chain to be placed in "f".

The initialization code for the capsule is executed using the newly allocated DA in "vtoc1". The last operation of the initialization code for the capsule is to create the representative associated with the capsule and its access class. The address of this representative must also be passed up the CREATES chain. The initialization code for the capsule then returns to the call from "seqmem".

Using the newly created representative to address the memory for "seqmem", the initialization code for "seqmem" is executed, and the "allocate" routine in "seqmem" is called to allocate a "seqfile". The value returned from "allocate" is concatenated with the MH address returned from the capsule and this new MH address is passed back to the caller. The initialization code for "seqfile" is executed but a representative is not created for "seqfile" as this is not an access to the file. Instead, the representative which was returned from the capsule (and possibly other representatives if there were more levels in the CREATES chain) is de-allocated, since a file does not become accessible until an access declaration is executed.

On termination of the block that contains the declaration of "f", the file must be made accessible to execute the termination code. This is accomplished by de-referencing the MH address for the file which creates the appropriate representatives. As in the case of allocation, the last representative does not have to be created because the file is not being accessed. The termination code for "seqfile" is executed and then the "free" routine is called in "seqmem" to release the storage for the "seqfile". This is followed by executing the termination code for "seqmem". The termination code for the capsule is executed which begins by de-allocating the representative for the file which is now no longer needed. Finally, the "free" routine is called in the VTOC memory to remove the capsule.

4.7 ADDRESS CONSISTENCY

It is necessary to ensure that all references to an entity in the system be consistent with the definition of the entity when the reference was created. In a strongly typed system such as this, consistency means checking that the type of the entity referenced has not changed since the reference to the entity was created. An inconsistency arises when either a change is made that will affect the way a reference interpretes the entity (i.e. a change in type), and/or there is a change in the location of the entity so that the address specified in the reference is incorrect. A reference can be made to either a ST entry (pointers to type definitions throughout the ST hierarchy) or a module (a procedure call) or a data item (an instance of an object, class, record, etc.).

4.7.1 Inconsistency

Logically, in a compiled system, it is possible to imagine that any references invalidated by a change to the system could be corrected by recompiling the entire system and deleting all DAs. This would assure that all references to the ST, to modules, and to data items are consistent because the compiler checks the type of each reference and re-creates all references using the corresponding definitions. All DAs will be re-created during subsequent execution and hence reflect their new ST definition. Clearly, this would not be practical, because it does not permit individual modules to be separately compiled and it does not permit persistent DAs. As soon as these capabilities are introduced, it is necessary to check consistency between the separately-compiled parts and their persistent DAs. Thus, it is the desire to support SCUs and persistent DAs which introduces the need for consistency checks in the system.

The notion of the SCU allows a portion of the system to change independently from the rest of the system even if there are outstanding references to the SCU. This will force checks at both compile time and execution time to guarantee that any entity referenced in an SCU has not been changed since the reference was generated. Since a user will, in general, wish to alter individual declarations, it is possible to imagine each declaration and each reference as an SCU. In this case, every reference would require a check to determine if the reference is still consistent with the entity it refers to. This too is not practical. Therefore, a compromise between the two extremes is chosen.

This compromise involves letting the programmer decide when static compilation checks are made and references re-created, and when dynamic checks will occur. This is done by controlling which modules are SCUs.

4.8 SCU COMPILATION

Within an SCU the compiler can guarantee that all references are consistent with the entities they refer to. However, to do this the compiler must guarantee that none of the STs that are used during the compilation of the SCU change. In the following example:

```
PROC p( ... ) SINGLE ...
  :
  x.y
  :    ←— change to definition of "x"
  x.y
```

if "x" is declared in "p" and if a change was allowed to "p's" ST during compilation, then between the two references of "x.y", the type of "y" could change or "y" could be deleted. If "x" is external to "p", the compiler would have to create a reference to its ST entry. The ST information that the pointer is referencing cannot be allowed to change or disappear during compilation of "p". To guarantee that STs are not changed during compilation, the compiler must lock all STs referenced from the SCU against write access. Thus, all references both inside the SCU and references to other SCUs are type checked and created to reflect their current definitions.

While there is the potential for many STs to be locked during compilation of an SCU, most of these STs will belong to the owner of the

SCU. Hence, STs not belonging to the user will be locked only occasionally. As well, these STs are locked only against changes and these STs will change rarely anyway. When references are made to another user's entities, the STs containing these entities will be locked, but these users must have made their STs public to allow the references in the first place (this is discussed in chapter 5). Hence, they must be resigned to the fact that these STs may be locked during another user's compilation.

4.9 **SIGNIFICANT CHANGE**

To detect an inconsistency between a reference and the entity being referenced, it is necessary for the system to be able to determine when a change causes an inconsistency so that it can be marked as significantly changed. There are many different changes possible to an SCU but they can all be divided into the following categories:

1. modification to the documentation - Such a change would not result in a significant change in the system.
2. modifications to the initialization code - Such changes would cause recompilation of the SCU so that the source language and object code are in agreement. Since no definitions are allowed in source language, any changes here cannot affect other references and make them inconsistent.
3. modification to the name of the ST entry - Here, such a change causes the source language to become inconsistent with the object code. The question here is whether this is really a significant change. From the system standpoint it is not, because the various pieces of object code remain consistent with one another. However, from the user standpoint the source language, which refers to the old name, is now inconsistent with the ST definition. To deal with this situation a decision must be made as to whether a module must be kept consistent at the source language or object code level. In general, it is a good idea to maintain consistency at the source language level so that object code always reflects the current source. Renaming would then be considered as a significant change.

4. all other changes to ST entries, especially changing the types (including the compile type) or deletion of the ST entry - Such changes would cause any reference to the SCU ST or an instance of its DA or to object code associated with the DA to no longer be consistent with the entities referenced, and hence the ST, DA and object code are marked as significantly changed.

Thus, only changes to declarations, which are under control of the PE, can result in a significant change. As well, the PE controls changes to an environment DA by ST modifications, hence it is in the position to determine which changes potentially result in a significant change to it. Once the PE has determined that a significant change has occurred to an SCU, it must take some appropriate action so that the significant change is detectable.

4.10 DETECTING INCONSISTENCIES

Once a significant change has been made to an entity and marked by the PE, the system must be capable of detecting the inconsistency between it and a reference to the entity. In general, this detection must be done at execution time because the compiler and PE cannot determine what other SCU's will be affected by a significant change. However, the dynamic detection of a significant change has the undesirable consequence that the time of detection is in no way related to the time at which the change is made. As a result, a user does not know the full effect of a change in the system, except through subsequent dynamic references that fail. In fact, because detection does not occur until a reference to the modified item or some containing item is made, it is possible that unrecoverable changes may have been made before an inconsistency is detected.

To provide a means of reducing this problem of the unknown effect of a change, the system could inform the user of all possible SCUs that are affected by a change. Of course, this approach is only a warning mechanism as the change is still considered significant and produces dynamic reference failures. Even if the user wanted to correct all modules that were reported inconsistent after a change he may not be able to do so because of security restrictions, or because of the large number of affected modules and/or DAs.

To inform a user of all references that become inconsistent as the result of a change, it would be necessary to record with every entity all references to the entity. This could be accomplished by a list of pointers stored with each ST entry. However, the ability to discern every reference to an entity would require that source language be parsed immediately so that the list of references to an entity could be kept up to date. Thus, to implement such a scheme would require a parse-tree editor for source language.

However, this scheme would not come without some cost. When a user deleted all references to a particular entity from a module, the parse-tree editor would access and delete a reference pointer from the ST entry for that entity. That ST entry might or might not belong to the user. If the ST entry did not belong to the user, it seems fundamentally wrong to allow another user's ST to be updated without some prior authorization by the user even if the system is performing the operation. As well, the amount of space in the ST for storing pointers to references could be significant. Also, the time involved in managing the reference pointers and assuring that the pointers themselves are consistent (even after a system crash, for example) might be considerable. For these reasons and because a user may not be able to correct all inconsistencies, I have decided that this idea is not practical. However, the system design does not preclude either a parse-tree editor for source language, or the inclusion of reference pointers along with the other ST fields. Hence, this scheme is not precluded should it be deemed desirable at a later date.

4.11 DETECTION MECHANISM

To perform a consistency check two pieces of information are required. One piece is associated with the entity that is referenced and one piece is associated with the reference. The information that must be stored at each point is some indicator to determine if the current type of the SCU is the same type that was used to compile the reference. When an SCU is significantly changed, this information is updated in the SCU. The exact nature of this information will be discussed shortly. For now this information will be referred to as detection information (DI)

4.11.1 Associating DI with the Referenced Entity

Since references to an SCU can be to its ST or its object code or its DA, all three of these entities must have DI stored with them. Thus, when the SCU is declared, the PE must set aside space in the SCU ST for the DI and must update the DI value when a significant change is made. The compiler sets aside space in the segment AT-entry of the SCU for the DI and updates the DI when the SCU is recompiled by copying the ST DI to the segment DI. This now reflects the fact that the SCU ST and the object code are consistent. In the case of a DA, the situation is more complex as there is no convenient place in a DA to store the DI. This is especially true since a memory writer is not obligated to organize the data items within the memory in any particular way. However, the system requires a memory writer to maintain DI for all data items allocated within the memory that can be referenced via a MH address. It is this DI that is compared with the reference DI. This DI can be stored in the memory AT entry for an object. However, this DI is normally never updated as the DA only represents one particular instance of a ST.

4.11.2 Associating DI with References

A reference to an SCU may be created either in its ST, when a reference is made to another SCU's ST, or in its object code, when a reference is made to another SCU's object code. Such a reference must be checked when it is subsequently used. This is because both the ST and the object code can change independently of the references to it.

References from one ST to another SCU ST are constructed by the PE when declarations such as the following are made:

```
VAR x : ^sys.zarnke.t
```

This creates a reference to the type "t" which is located in another SCU, possible in a different environment. This reference may be a MH address if the type is not in the same environment as the reference. When the PE looks up the type, the DI stored with the type is extracted and stored with the reference in the type field of the ST entry. References in the executable statements to the code of another SCU are constructed by the compiler. When the compiler is processing such a reference, it must look up the ST for the SCU to compile the appropriate object code. At that point the DI is extracted and stored with the reference in the generated object code.

In theory, each address, whether a component in a MH address or an instruction address in a segment, needs DI since the potential exists for that data item to be modified. However, in practice it will be seen that the DI for many of the references in an SCU are the same, and as a result can be factored out so that only one piece of DI is needed for all references in an SCU.

4.11.3 Internal and External DI

While it is possible to imagine any change to the declarations of an SCU resulting in a significant change, it is worthwhile to divide these changes into two categories: internal and external. By discriminating between these, it is possible to reduce the number of inconsistencies that result from changes and hence the amount of recompilation that must be done. An internal change is a modification of a ST entry that is not visible outside the SCU (i.e. modification of local variables of procedures). An external change is a modification of a ST entry that is visible outside the SCU (i.e. to the interface of the SCU). This occurs for modifications to the number and type of parameters (including a return value(s), if any), and modification to routines that are externally callable or data items that are externally accessible. Since these entities are all visible inside the SCU, an external change is also an internal change. However, in an object-oriented system, there will be many external references to objects and their components throughout the system.

Therefore, it is worthwhile trying to prevent these external references from being considered invalid when only an internal change is made to an entity. To do this, the DI contains information related to both types of changes. References to externally visible entities have the external DI associated with them, and the dynamic consistency check compares the reference DI with the external DI. Internal changes only affect the internal DI and do not invalidate external references.

4.11.4 Consistency Checks

Checks must be made for each reference to an SCU to determine if the reference is consistent with the entity referenced. In the case of references to a ST entry, the PE and compiler must check each reference as they use the ST hierarchy during editing and compiling. This check determines if the type of the ST entry that is referenced has changed since the reference was created. In the case of references to an instance of an object or memory, the compiler must generate object code to check the type at execution time. For references to modules, the check is done by object code at the beginning of the SCU before transferring to the actual routine, and for references to DAs, the check is done when the DA is made accessible. This checks if the type of the item that is referenced has changed since the reference was created. All these checks involve extracting the DI from the reference and the entity referenced, and comparing them for equality. If they are equal the reference is consistent with the entity; otherwise, they are not and the reference must raise an exception.

4.11.4.1 DI Check

For both logical and security reasons the DI check should be done before the referenced entity is actually accessed (i.e. before a representative is created for it). As well, the DI is associated with the AT entry for security reasons (i.e. it is located outside the entity where it cannot be referred to). The DI check must be done at the time and place that will maintain these constraints.

One such point is when the address is being decoded. Here the component is looked up in the AT entry to get the address of the entity in the memory. At this point the DI can be extracted too. As well, the decoding of the component is done just prior to creation of the representative. Hence, the component decoding routine is the obvious place for extracting the DI from the AT to perform the DI check. Two alternative ways of performing the DI check during component decoding are the following:

1. The memory returns the DI to the system which then does the DI comparison and takes the appropriate action.

   ```
   PROC adrtrans(component : COMPVAL) RETURNS a : ADDRESS, di : DI
   ```

2. The system passes the DI of the reference to the memory, and the memory does the comparison and takes the appropriate action.

   ```
   PROC adrtrans(component : COMPVAL, di : DI) RETURNS a : ADDRESS
   ```

While both schemes provide the same effect, the latter provides for a slightly more flexible approach.

For example, if the DI is passed to "adrtrans", it is able to do a more sophisticated check than the system could. Normally, the check would be to compare the DI with that for the entity being addressed, and if these DI values are not equal, signal an exception, otherwise continue processing. However, if the memory writer wanted to implement versions of data items, for example, the memory could have a series of data items all with the same address. The memory then discriminates between the versions of the data items on the basis of the DI passed to it. Thus, the same MH address would get access to the appropriate version of the data item. This usage of the DI over and above its normal use would be impossible to implement if the system performed the check.

4.12 INCONSISTENCIES THAT REQUIRE CONSISTENCY CHECKS

The following situations discuss in detail the effects of significant changes and how the consistency check will detect the change and prevent any invalid accesses.

4.12.1 Changing the Compilation Type

When the compilation type is removed or changed, the PE must perform the necessary action to correct any inconsistencies that will result from the modification. When the compile-type is removed from a module, the module inherits the compile-type of the closest containing module that has a compile-type.

Changing the compile-type can be broken down into only two basic changes: from multi-copies to single and vice versa. This is because INLINE and COPIED produce multiple copies of the object code and hence they behave similarly when changed to SINGLE.

4.12.1.1 Effect of changing from Single to Multiple Copies

When a module's compile-type is changed from SINGLE to either INLINE or COPIED, it is necessary to force recompilation of all references to the changed routines. The recompilation of the references causes the necessary object code to be copied into the segment containing a reference. In order to prevent other references from still calling the SINGLE version, the DI for the SCU can be changed which will force any outstanding references to fail when attempted later. Their subsequent recompilation will then incorporate the necessary object code.

4.12.1.2 Effect of Changing from Multiple Copies to Single

When a module's compile-type is changed from COPIED or INLINE to SINGLE, the change cannot be accomplished as easily as above. This is because it is not possible to change all references to now refer to the single copy because these references refer to the copy in their own segment. However, this phenomenon is a property of copying; that is, references to copied routines will not be updated automatically when changes are made to the original. Only when the module that contains the reference is eventually recompiled, will the references refer to the single copy.

Unfortunately, a small problem still exists. For the segments that contain copies of the COPIED module whose compile-type has changed to SINGLE, it is difficult to know when the COPIED object code is no longer referenced in a segment and hence can be deleted. In the case of INLINE, this is automatically done when the reference is recompiled as the old object code actually contained the INLINE code (i.e. the INLINE object code is not kept separate, but is embedded in the object code where it is referenced). However, for COPIED modules, there may be several references to the copied module from other modules in the segment. The COPIED module cannot be removed until all the references are recompiled and refer to the SINGLE copy of the module.

The system checks for this situation and automatically searches and removes the COPIED module when it is no longer referenced. This is possible because the AT entries for COPIED modules are appropriately marked, and all RLD references in the segment can be checked to see if that AT entry is ever referenced. Hence, it is possible to find these dangling modules and remove them. Unfortunately, there is no way to detect when the last reference to a dangling copied module is removed, and as a result these dangling modules are checked for every Nth time the segment is compressed.

4.12.2 Consistency of References to Inline and Copied Code

Because INLINE modules are copied at the point of reference, there can never be an inconsistency between the reference and the object code as there are no actual references.

COPIED modules are also copied into segments that reference them, but not immediately, for example:

```
PROC p( ... ) COPIED l
  PROC q( ... ) COPIED m
```

Here compilation of "p" does not cause "q" to be copied into segment "l". Instead, only an ESD entry is created in "p" which points at "q". Only when "p" is ultimately linked with other modules to form an executable module will all the ESD entries be resolved and all the appropriate OMs copied together. Thus, there is a time lag between the creation of the ESD reference and the time the OMs that are referenced are bound together. Hence, it is possible for the OM to have changed since the ESD reference was created.

This situation is solved using the same technique as for SCU references to SINGLE modules. DI is associated with both the ESD and the OM. When the OMs are finally brought together to form an executable module, a consistency check is performed. If the consistency check fails then the user will be informed immediately, and he should recompile the out-of-date module. However, whether he does or not, the reference is considered invalid and will cause an error at execution time if used.

4.12.3 Changing the Environment DA

In the case of changing the environment ST, the PE knows when the environment ST is significantly changed. If a new variable or module is added to the environment ST then the environment DA or segment may be able to be appropriately modified without affecting any existing references. If this is the case, the PE does not change the DI. However, when a significant change is made to the environment ST, such as deleting a variable or a module, then DI associated with the the environment DA or the module's AT would have to be modified. Here the effect of changing the DI for the environment DA would have a greater impact than changing the DI for an SCU's AT. All references to the environment DA would now have to fail, whereas only the references to the deleted module in the segment would fail. Clearly, there is a finer granularity of control over the effect of a change in the segment as opposed to the environment DA.

These two different degrees of effect are a function of the system design. Normally, a DA is mapped by a module ST and there can be multiple DAs per module definition. Changes to the ST, even just the addition of a new local variable, cause all outstanding DAs to become inconsistent with it. In general, it is not feasible to try and find and update all outstanding DAs to correspond with the new module definition. Hence, it is not necessary to have a fine control over the effect of a change to the DA.

The effect of change on the environment should not be quite as drastic as on normal DAs (i.e. a simple change should not make all references to the environment inconsistent). This is because there is a one-to-one correspondence between the environment ST and DA, and this makes it feasible to maintain consistency between the two. However, in many ways, changes to the environment DA have the same consequences as for other DAs: that is, to invalidate any reference to data items in the environment. As a result, changes such as deleting a data item or changing a data item would cause all references to the environment DA to fail. These failing references could be corrected by modifying the modules and recompiling them or in some situations just recompiling them. However, in general, this is too severe a constraint. Hence, the design for the allocation of data items in the environment DA will be modified to allow a finer granularity of control over the effect from a change. The environment DS will be organized like a segment where access to each internal entity has a level of indirection in order to reduce the effects of modifications (i.e. an AT entry).

Because a segment is a memory like a DS, it will also have DI associated with it as a whole. This is in addition to all the DI in the AT entries contained in the segment. This segment DI is always checked when a segment is made accessible. The same check occurs as with a module and a DS. The purpose of this check is to insure that none of the AT entries have been moved in the segment. If the AT entries have been moved, all modules with references to the AT are inconsistent.

4.12.3.1 Updating STs for Existing Objects

Because of the coarse granularity of control, it is difficult to change a module definition without invalidating outstanding DAs. Yet, changes must be made and old DAs must not be lost. To accomplish a modification, such as adding a new local variable to "seqfile" without invalidating all the existing "seqfile's", the following procedure can be used.

To avoid such invalidation it is necessary to support multiple versions of object code for a particular SCU. Creating a new version must not change the address or type of the old "seqfile" and hence all old DAs still depending on it are still consistent. A new version of sequential file is created so that all subsequent declarations will access the new version. As long as the old and new versions of "seqfile" are based on the same prototype, all existing references in modules will continue to work. This is because the DA, when accessed, will chose the routines that correspond to its definition. Thus, old and new versions can run simultaneously. This selection can be based on the DI associated with a reference.

Ultimately, it will be desirable to delete the old definition of "seqfile". To do this it would be necessary to write a conversion program that converts an old "seqfile" to a new "seqfile". This program looks like the following: it accepts two parameters, the old and new "seqfile", and does a record by record copy from the old "seqfile" to the new "seqfile". It is also possible to modify the old version of "seqfile" to print a warning message to the user when the old version is accessed telling him that his file must be converted. This change can be done as a source language change, and hence is not a significant change. While such a facility will probably exist in this system, it is not within the scope of this thesis to detail the exact mechanism and syntax required to provide versions of object code.

4.13 DETECTION INFORMATION

Detection information must allow the system to determine if the type of the entity when the reference was created and the type of the entity in its definition are the same. The following presents two schemes for defining the information necessary for the consistency checks.

4.13.1 Versions Numbers

In this scheme, a version number is associated with the ST for each SCU. Here a simple integer counter is associated with the ST for each SCU. When the SCU is significantly changed, the version number is incremented. This version number is stored with each reference to the SCU and any DSs for the SCU. The dynamic consistency check determines if the reference and entity are generated from the same version.

4.13.2 Time Stamp

In this scheme, a system wide clock is used to associate a time stamp (TS) with the ST for each SCU. When the SCU is significantly changed, the TS is updated to the current time. This TS is stored with each reference to the SCU and any DSs for the SCU. The dynamic consistency check determines if the reference and the entity have the same TS [FELDM79].

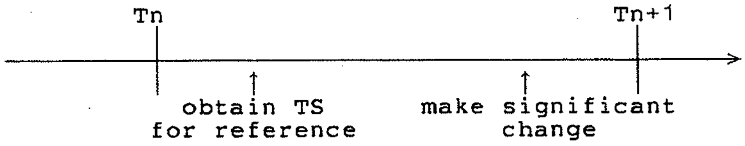
Conceptually both schemes provide identical facilities for detecting inconsistencies. Hence, the reason for choosing one over the other is based on considerations other than the checking consistency. Of the two methods, I have chosen the TS method. This is because the TS method allows several optimizations to be performed that are not possible with versions.

4.14 TS SIZE

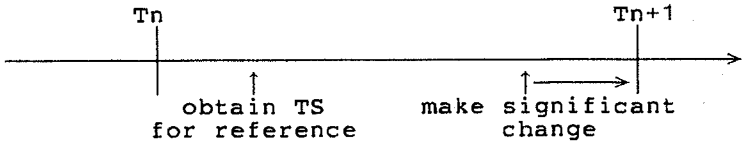
The size of the TS depends on the duration over which the system will exist and the accuracy that is necessary. A system must be expected to last tens of years; therefore, the TS must be large enough to handle this case. The accuracy indicates the speed at which the TS clock is ticking. If the accuracy of the clock must be high, then the TS will be large, and occupy a larger amount of memory.

4.14.1 TS accuracy

The problem that affects the accuracy of the clock is that once a TS has been obtained for a reference, it is necessary that the entity not change significantly before the clock ticks. Otherwise, the reference will have the same time as the changed item and hence appear consistent when in fact the reference is later than the change.



To guarantee that this does not occur, the PE, which is making the significant change, must not release access of the STs that are affected until the next clock tick.



If the change takes more than a fraction of a clock tick, then it must be treated as if it required an integral number of clock ticks. In the case of making a significant change to the environment ST, the time for the change must be the time of update to the environment DA. Otherwise, a reference could be compiled and an access attempted between the time of the ST change and the time the data item is updated or reallocated. This ensures that any reference made in the time period before the significant change will fail as the changed entity will have TS Tn+1 or greater. As well, since no references to the ST can be made when the ST is locked, any subsequent references must have TS ≥ Tn+1.

The PE can prevent all references while a user is LOCATEd to a LST and changing it. Thus, a user can make multiple changes to the LST and the internal and/or external TSs are updated only once when the user unLOCATEs from the LST. Since this prevents other users from accessing the ST, the user may be able to indicate on the LOCATE command that only read access to the LST is desired. In this case, the PE does not have to lock the LST, but must prevent any modifications.

Thus, the accuracy of the TS ticks must be such that the locked period forced by this scheme does not impose too long a wait for subsequent access to the locked ST. For example, if a clock tick were 5 minutes, then the worse case would be a 5 minute wait until the STs that were locked could be released.

In this system it is imagined that almost all significant changes will be performed by users communicating with the machine at terminals, and not by programs that are modifying other programs. As a result, it is possible to have a clock tick that is relatively large from the machine standpoint but relatively small from a people standpoint. For example, a clock tick of 0.25 seconds would impose a worse case wait of 0.25 seconds and an average delay of 0.125 seconds. These delays will probably go unnoticed by the user. And since a user's program will mostly refer to his own STs, there will be little delay in the system resulting from the extra time the STs are locked.

This slow ticking of the TS clock allows the size of the TS clock value to be relatively small. A 32 bit value would allow a system duration of 30 years before the system TS clock is reset. At the end of this period, the system can be re-created by recompiling it and updating all persisting DAs.

4.15 OPTIMIZATIONS OF CONSISTENCY CHECKS

At the moment, every reference to an entity that is in another SCU ST or SCU instance must be checked for consistency. To accomplish this, every reference must contain either internal or external DI and every entity referred to must contain both internal and external DI to allow the check to occur. In practice this could amount to a substantial amount of information that must be retained and checked. Several transformations can be made to provide a practical implementation of this scheme. These transformations preserve the ability to perform the consistency checks and yet eliminate some of the DI and some of the dynamic checks.

4.15.1 Forcing Recompilation

When a significant change is made to an entity, most references within the entity are no longer valid. The only exception would be a routine that is self-contained (i.e. has no external references). The entities that contain these references must be modified and recompiled before they are consistent with the changed entity. This is true for either a variable or type change within the entity.

4.15.1.1 Variable Change

In the following example:

```
PROC or OBJECT p( ... ) SINGLE 1
  VAR x : ... <- change or delete x

  PROC or OBJECT q( ... ) SINGLE 1
   :
```

the change to "x" has the potential to invalidate all modules contained in "p", even if the contained modules do not reference it. The reason that all modules may be affected is that a deletion or change to "x" may cause the displacements of all the local variables after "x" in the ST to be reassigned. Even if the compiler did not change the displacements for old variables this would be of little benefit without also knowing which contained routines referred to "x". The only exception is if a subordinate routine is self-contained.

Hence, if the PE wanted to do selective compilation of affected routines only, it would be difficult to determine the complete effects of a change. As well, all the contained routines would have to be searched to see if they referenced an entity that had been affected. This latter problem might be mitigated by having an IMPORT statement in the programming language (à la EUCLID) that listed all imported entities.

The PE may be able to allow new variables to be added without affecting existing ones since existing types and displacements may not be affected. This is possible because only the PE is managing the ST.

4.15.1.2 Type Definition Change

In the following example:

```
PROC or OBJECT p( ... ) SINGLE 1
  RECORD or OBJECT t  ← change external definition of "t"
  VAR v1 : t

  PROC or OBJECT q( ... ) SINGLE 1
    v1.r or VAR v2 : t
```

The effect of a change to "t" may not directly affect "p" since "p" may not reference "t" (i.e. has no variables of type "t"). However, this still leads to the same problem as with a changed variable: locating the contained routines that are affected. Rather than do this, it is reasonable to assume that since "t" is defined in "p" that a change to "t" will likely affect "p's" DA or some contained DA.

Thus, an internal change results in marking the changed entity and all its descendents as needing recompilation, unless the descendent is self-contained. By adopting this scheme it is not necessary to perform consistency checks between references to containing (i.e. global references) variables as any change will cause the reference to be recompiled and hence be made consistent. This also eliminates the need for the internal DI for those entities whose data areas are transient, such as procedures. Notice that the entities are only marked as needing recompilation and not actually recompiled. This is because further changes may be necessary before the affected routine can be recompiled successfully.

There is one exception where the internal DI must be retained, and that is if the object creates a DS. In this situation outstanding DAs may exist and these become inconsistent after an internal change. Hence the internal DI is necessary and is used by the dynamic check when the DA is accessed.

One other situation can arise that forces recompilation. However, in this situation the compilation of a lower level routine causes recompilation of a high level routine. For example, in this routine:

```
PROC or OBJECT p( ... ) SINGLE 1
  VAR o : ^sys.buhr.t  ← t is an SCU

  PROC or OBJECT q( ... ) SINGLE 1
    :
       o.r
```

"^sys.buhr.t" is a remote type and a significant change to it does not cause any references outside the cluster definition in which it is declared to be marked as needing recompilation. In this case, where "t" is an object or class, this is handled by a dynamic consistency check at the point of the declaration of "o", which detects that the reference in the declaration is no longer consistent with the object code for type "t".

However, if "t" is a RECORD there is no dynamic check as the object code to allocate an instance of "t" is compiled directly into "p". Hence, the change to "t" does not affect "p" or its descendents. When executed, "p" will allocate an old version of "t" which is consistent with all references to "o" in "p" and any descendents.

But if "q" is recompiled, the reference to the component "r" of "o" must use the new definition of "t" to determine the displacement of "r" and yet use an old instance of "o" which is created in "p". This inconsistent situation is detected when the compiler looks up "t" during the compilation of "o.r" to determine the displacement of "r". A consistency check is performed between the ST type pointer for "o" and the ST definition of "t". This check would fail since "t" has been significantly changed. If this check fails, this implies that a type has changed in "p" and "p" and its descendents must be recompiled. Thus, the compilation of "q" should probably be aborted and "p" is marked as needing recompilation. Subsequent recompilation of "p" will subsume the recompilation of "q" and all other routines potentially changed by the change in "p".

4.15.2 Automatic Compilation

Internal changes imply recompilation of all the contained entities. As well, changes to the source language of an entity implies that the entity should also be recompiled. In any system where the source language is not executed directly, there is a transformation of the source language into object code for some interpreter (usually the hardware). Both the source language and object code can be potentially manipulated independently. From the system standpoint, as long as references are consistent with definitions, the programming system is consistent. However, from the user standpoint it is not reasonable to have the source language specify one thing, and the object code another. Thus, it is desirable to have the object code always correspond to the source language.

This can be done by having a <u>recompilation indicator</u> for each SCU both in the ST and with the object code. This indicator is set by the PE after a user is finished editing the source language for an SCU. At execution time, this indicator is tested before control is transferred to an SCU as part of the TS check. If the indicator is set, execution is suspended and the SCU is recompiled. If there are no compilation errors, then execution can be resumed. If a compilation error occurs, the resulting SCU becomes invalid and results in an execution time termination. So that further attempts to recompile the erroneous SCU are not attempted, another indicator might be worth while. This <u>unsuccessful recompilation indicator</u> is set by the compiler indicating an unsuccessful compilation and is reset by the PE when it sets the recompilation indicator (i.e. when a change has been made to correct the error in the source language for this SCU). The unsuccessful recompilation indicator is tested at execution time only if the recompilation indicator indicates that recompilation is necessary.

This dynamic compilation is possible because the AT has a pointer back to its ST entry (the pointer is originally there to allow symbolic debugging and deferred object resolution). Notice that the automatic compilation does not occur until there is a reference to the entity.

Thus, the user is not forced to make changes in such a way that the source language is always compileable. The source language can be modified over a period of time as long as the entity is not referenced dynamically during that period. If a reference does occur during the modification period, while the source language is not completely changed, a recompilation will still occur and compilation error messages may be produced resulting in the failure of the reference.

If a user explicitly compiles his module, the compiler resets the recompilation indicator for that module. In this way a user can check for syntax errors before making a reference to an entity. However, if a user ever forgets to explicitly compile a module he has modified, he will always get the modified version when he invokes the module. This is particularly important to beginning programmers who often make changes and forget to recompile their modules before testing them to see if the change fixes the module.

4.15.3 Compile Time TS Checks

Since the consistency check must occur at each call to each SCU, this increases the overhead of all calls. This might be mitigated by having the hardware call instruction also perform the TS check. The TS check might be done in parallel with saving the status of the current routine. However, in some cases it is possible to eliminate the dynamic consistency check and instead perform the TS check at compile time. The situation that allows compile-time DI checks is when entities reference one another in the same segment and are not externally callable. Inter-segment calls must always do a dynamic consistency check.

When a non-externally callable module is significantly changed, it is possible at compile time to check all routines in the segment to determine if they reference the changed module. If they do reference the changed module, the RLD entries can be modified to branch to an error routine in the segment instead of the referenced routine. Hence, at run-time, the call will execute the error routine which generates a consistency error. This allows elimination of the dynamic TS check on calls and hence the extra object code to perform the check. Modification and recompilation of the module containing the reference will eliminate the consistency error.

However, compile-time TS checks increase compilation time. First, all calls to other routines in the segment must be done using the RLD table even if the call would normally be done directly using the AT. By using the RLD table in this fashion, it is possible to determine all the routines that a module references. Secondly, it is expensive to locate all the references to an inconsistent routine. All RLD tables in the segment must be examined to locate any references to the inconsistent routine. Because of this additional overhead for compile-time TS checks, the user might want to decide whether he wants compile-time TS checks or dynamic TS checks for intra-segment calls. When a user declares a segment, he could specify which type of check will be used.

4.16 REMOVING REFERENCE TS

If the type field of a ST entry points to the ST of another SCU, then any use of that pointer must be checked. However, DI for all type pointers in an SCU are updated during compilation since all the ST entries are remapped to reflect their current type. All the TSs for the type references will be less than the compile time, and because of ST lockout, any type that is subsequently changed will have a TS greater than the compile time. Thus, the compile time can be used to perform the consistency checks.

This same argument applies to references in object code for an SCU. When an SCU is compiled, all of its references to other SCUs have the TS information from the ST of the SCU associated with them. All of these TSs are less than the time of compilation since no changes are allowed during the compilation. This means that if any of the referenced entities are changed they will have TSs greater than the current compile time. As a result, it is sufficient to store the compile time of an SCU with the object code, and remove all the TSs associated with each individual reference to other SCUs. This compile-time TS is then used for checks on references between other SCUs and the entity. If the reference TS is greater than or equal to the compile time of the entity being referenced it is consistent, otherwise an exception is raised.

As a result, it is possible to have a single compile-time TS associated with both the SCU ST and the SCU object code and eliminate all the TSs associated with each external reference. This scheme would not work if version numbers were used for the DI. This is because the version numbers for different entities are not related, and hence they could not be factored out as a common version number.

4.16.1 DA Address TS

Unlike the addresses created in a module which are all created at the same time, dynamically created entities (such as objects creating new memories) can be created at any time. As a result, there is no way to factor out the TS for the pointers to DAs. This is especially true in the case of an environment where objects like files can be created at any time. Hence, each DA address must have its own TS. However, it is not necessary to have a TS for each displacement (component) in the MH address for a DA. During allocation of a new DA, the MH address is built up as each component is created. A single TS for the entire MH address is adequate to verify consistency between each component and the DA in which it refers. The value for the TS is the time at which the allocation of the entity begins. Unfortunately, including the TS with each DA reference increases the size of a MH address substantially.

4.17 REUSING AT AND EQUIVALENT ENTRIES

Eventually, modules and DAs will be deleted by the user. But after this is done, there may still be outstanding references to the AT entry of the object code in a segment or the capsule of a DA in a memory. These references must fail with a consistency error if executed.

There are several ways of handling the problem of dangling references so that consistency is maintained.

1. Mark the AT or equivalent entry as deleted and do not reuse it. Hence, any outstanding references will fail on access to the AT entry. Unfortunately, in a highly volatile memory, there will soon be a large number of unused AT entries resulting from deletions and this will force a larger component value.
2. Change the TS in the AT to the deletion time and allow the entry to be reused. This is possible because the AT entry does not need to point at the old (deleted) item. Since the time of the

entity is now past the time at which all the references were created, the TS check will detect any invalid reference to the deleted entity. When the entry is reused, the time of creation of the new entity can be stored in the AT entry and this time will always be further in the future than the deletion time, and hence all old references will continue to fail. The only restriction is that the entry must be maintained in exactly the same position in the AT.

Of the two schemes, the second provides the greater flexibility in utilizing space in the memory and still allows consistency to be maintained.

4.18 IMPLICIT AS

The way in which the programming system has been designed implies the existence of multiple active ASs (MAAS) in the hardware. The idea of MAAS was developed in the Multics system. There, a task can have access to any number of Multics' segments, some containing data, others containing object code. This is in contrast to most large computer systems which support only a single-level store (i.e. only one AS directly accessible from a task). In the case of micro and mini computer systems, MAAS were introduced because the address size on these systems severely restricted the amount of accessible memory. Hence, these systems were forced into MAAS not because of the desire to solve storage management problems but simply to be able to increase the amount of data that could be addressed.

In the single-level store scheme it is possible to mimic access to multiple ASs. This is done by mapping a portion of each accessed AS into the single AS accessible by the task. While this scheme can simulate MAAS, many of the benefits that result from MAAS are lost: in particular, to "limit or control access to information so that a computation may be self-protected from its own mishaps and so that different computations that share the same physical facilities may be mutually protected"[BENSO69]. As well, simulating MAAS introduces a difficult storage management problem in the single-level store. Therefore, it is imagined that this system would be implemented on a machine that actually provides MAAS.

The general architecture of this machine would resemble that of the GE 645 [ORGAN72] which supported Multics. However, in detail many things would be different. This is because the Multics system is fundamentally different from this system, and hence has different specific hardware requirements: in particular, implicit accessing of a segment at the first reference. To support this, many hardware features were built into the machine. Addresses have to have a special bit indicating whether they are symbolic or have been converted to an actual address and a segment created, and this bit is tested on each reference. This is linked with construction and use of the Multics' segment table which is also part of the hardware addressing mechanism. However, in this system, access is not done at first reference but is done explicitly before a reference, and hence most of the special Multics hardware features are not required.

In Multics, all the segments were conceptually all accessible, but the hardware only has 4 or 5 segments that it could access at any instance. If a different segment was to be accessed, it displaced one of the 5 active segments. Like Multics, this system requires several ASs directly accessible by the hardware:

1. segment AS - instructions are implicitly fetched from this AS
2. stack AS - storage for local variables of procedures and for parameters
3. heap AS - storage for dynamically created objects
4. data AS - storage for the DS of objects
5. temporary data AS - storage for the DS of objects that must be accessed temporarily, such as parameters in an inter-task call, or data moved from one DS into another DS

All the ASs must be explicitly selectable in the address portion of an instruction. For example, the segment AS may be accessed explicitly because it may contain constants.

These 5 active ASs will be accessed through 5 special hardware registers on the machine. These hardware registers point to the representatives for the ASs instead of to the actual data structure for a page table. This is desirable in order to permit the hardware to make direct calls to the representative, for example, on a page fault. This implies that the hardware is aware of the structure of a representative and not just of a page table.

4.18.1 Changing of Implicit ASs

During a call some or all of these ASs may be changed. The conditions that cause each AS to change are the following:

1. The segment changes when a call is made to a routine that is compiled in a different segment.
2. The stack and heap change during a call to a routine in another task. This may depend on the programming language task-communication mechanism.
3. The data area changes during a reference to a routine contained in an object that creates a DS.

For example, the declaration:

```
VAR pf : f.seqacc
```

creates an access class which refers to a new segment and DS and uses the existing stack AS. "Seqacc" will be compiled into the segment with "seqfile" because it has been defined this way. This was done because it would use too much storage if "seqacc" was copied into all users segments and there would be problems in updating these copies should "seqacc" change. In fact, all the components of "seqfile" will likely be defined together so that only one segment is needed to access all routines. The storage for the local variables of "seqacc" will be allocated on the heap of the calling module. The instantiation of "seqacc" makes accessible the AS for the sequential file "f" as the DS. Variables, such as "first_record" and "last_record", will be accessed using this data memory. Because the compiler is aware of the 5 accessible ASs during compilation of "seqacc", the compiler can determine which variables are in which ASs. Hence, in this case they can be accessed without MH references. This greatly reduces the overhead involved in executing in "seqacc".

4.18.2 Implications of Implicit ASs

It is necessary that the compiler know exactly when an AS change occurs and in which AS a variable resides (i.e. which of the above 5 ASs), because the object code generated to access variables depends on the AS in which the variable is located. Normally the compiler can determine the AS in which a variable is allocated from the context in which it is declared. For example, entities declared in an object are in its DS, and entities declared in a class or procedure are allocated on the stack or heap. When arguments are passed by value, they are copied onto the stack and the object code to access them in the calling module retrieves them from the stack. In the case of an inter-task, call the old stack is accessible via the temporary AS.

In the case of passing arguments by address or passing the value of a pointer, the called module does not know which of the 5 implicit ASs the address is relative to. And even worse, the ASs may change when the called module is entered. Therefore, to pass an argument by address may require that additional information be provided that identifies the AS in which the argument is stored. One possible way of doing this is to pass the MH address of the argument. However, constructing the MH address dynamically, and de-referencing it in the called module would be expensive. A simpler solution is to pass both the address of the data and the address of the representative that can be loaded into the temporary AS register to access the argument. This type of address which contains both the representative address and the data address is called a REPREF.

While it is conceiveable to implement all transmission of arguments by address using REPREFs (as is done in Multics where each address contains a segment number and the data address), it is undesirable for routines that operate on the same DAs. These can pass parameters by simple addresses.

To handle this situation, the compiler must be able to determine when parameters passed by address come from ASs different than the ones active when a routine is called. The situations where this can happen are:

1. When the routine is externally visible and the object creates a DS, here the argument might be in the caller's DS, or the return value might be in the callee's DS.
2. When a routine has a SINGLE type as its compile-type and is compiled into a segment different than the next containing entity with a compile clause, as in:

```
PROC or OBJECT q SINGLE Q
  :
  PROC or OBJECT p SINGLE P
```

   The segment AS will change during a call to "p" and an argument might be a constant in the calling module's segment AS.
3. When the entity being called is a task, here the stack and heap will change and an argument may be in the caller's stack or heap.

All parameters to routines in any one of these contexts must be REPREF parameters.

Theoretically, routines like the "read" routine in "seqfile" must return a REPREF pointer, because the pointer is relative to an AS different from the caller's. However, in practice, this is necessary only if the pointer is to be stored by the caller. And, even this is unnecessary if the pointer is qualified by the access class to which the pointer is relative, as was mentioned in chapter 2; through the access class, a pointer to the representative can be located, and the compiler can use this to obtain access to the AS that the pointer is relative to.

The most common situation, however, is to de-reference the pointer immediately upon returning from "read" as would be done in:

```
r <- pf.read()
```

where "r" is of the same type as the file's records. Here, the compiler knows the AS to which the pointer from "pf.read()" is relative (namely, the previous DS) and hence can extract the data value and assign it to "r". In the case where the pointer is copied, as in:

```
p <- pf.read()
```

where "p" is a REPREF to the same type as the file's records, the compiler can detect that the simple address returned by "pf.read()" must be converted to a REPREF and can construct the REPREF address dynamically. Thus, both situations can be handled without the need to declare "read" as returning a REPREF pointer.

As well, there are situations where the programmer must declare parameters to be REPREF in other contexts. For example,

```
OBJECT q ...
  PROC p(k : REPREF t)
    PROC r(w : REF t)
       :
    END PROC
       :
    r(k)
       :
  END PROC
END OBJECT
```

In this example, "p" is externally visible and the programmer has made parameter "k" a REPREF. However, when "k" is passed to "r" there is a problem. Procedure "r" is an internal routine and hence assumes that all pass-by-address references are in some accessible AS. To be able to accommodate either an argument from outside the object or a variable inside the object, requires that the parameter be a REPREF. Hence, the programmer would have to declare the parameter "w" to be REPREF, as well.

4.18.2.1 Structure of a REPREF

Besides pointing at the object being addressed, the REPREF must also identify the AS in which this object is allocated. It does this by pointing at the representative for that AS. In addition, in order to check for dangling references, the REPREF must include a TS. For example, a dangling REPREF can occur if an access variable is declared in an environment DA. Here, at sign-off from the environment, the representative is deleted, and hence the access class pointer to the representative is no longer valid. This is because the access variable points to the representative with a REPREF pointer. Thus, the REPREF is just a MH address. The first component forms a pointer to the representative, and the last component is a displacement within the memory corresponding to the representative. As well, a TS is associated with this MH address.

4.19 PASSING ARGUMENTS USING EXPLICIT QUALIFICATION

The previous discussion explained what is done when a caller passes an argument by address that may not be directly accessible to the called module. The next case to consider is when an argument is not directly accessible in the caller. That is, the argument is not in any of the caller's implicit ASs and in fact does not necessarily have a pager currently available through which it can be accessed. Consider the call:

```
p(^sys.zarnke.f)
```

where "p" is defined as:

```
PROC p(f : seqfile( ... ))
```

In the call to "p", the argument is a "seqfile" in environment "zarnke". The value that must be passed to routine "p" is the MH address that is stored in environment "zarnke" for "f". Within the body of "p" this address can be used to declare an access class for "f".

However, to get the address of "f", several other DAs must be made accessible implicitly. To compile the call, the compiler must generate object code to extract the MH address for each component in the qualified reference. The steps involved are:

1. Look through containing objects for an object "sys", which must be accessible since it is statically above the entity that calls "p".
2. Fetch the MH address for "zarnke" from "sys" and de-reference that address. The location of the MH address for "zarnke" in "sys" is the displacement that is compiled into the object code from the ST entry for "zarnke".
3. Fetch the MH address for "f" from "zarnke".

In this case, once the MH address for "f" is found, the objects made accessible to locate "f" may be de-accessed. If "f" is not self-contained, then any ASs it depends on will have to be made accessible. For example, if "seqfile" accessed a data item in "sys", "sys" would have to remain accessible during the access of "f".

It is important to note that the implicit accessing of the qualifiers is independent of the implicit accessing of components in a MH address. Each of the three components in the qualified name has a MH address which itself is composed of several components. Resolving both addresses will result in the accessing and hence creation of representatives for the necessary DAs. However, in both cases most of the higher-order components will already be accessed and hence will require little overhead to find their representatives.

4.20 SUMMARY

MH addresses reflect the fact that objects (including capsules) are allocated dynamically in nested memories. The component values which make up the MH address and identify an object can be encoded. The form of encoding is not mandated by the system, but instead depends on each MEMORY definition; each MEMORY definition determines the structure and meaning of the component that it provides in the MH address. The encoding of a component value has the benefit of producing a smaller address and of allowing a level of indirection between the component values and the actual object. This indirection permits the object to be moved around within the memory for storage management reasons and yet still maintain a fixed address for referring to an object. The resolution of a MH address involves traversing the MH, accessing an object for each component in the address. This object access causes the creation of a representative for each object.

TSs provide a mechanism for checking that addresses still refer to the proper type and location of entities. This check allows the system to maintain integrity when entity definitions can be changed independently of entity references; such changes are possible because of separate compilation. This check is a dynamic check and in theory must occur for all references. However, by the judicious choice of the entities that are given TSs based on the compilation units that the user defines, many of the dynamic checks can be eliminated or at least reduced.

In general, it is desirable that hardware assistance be given in the tasks of storage management and access. One way in which this is accomplished is by having ASs paged. And, in order to be able to access several objects simultaneously in a module, the hardware must provide a MAAS capability. In the author's opinion not enough effort has been spent in hardware design to aid the software developer in dealing with addressing and storage management problems. I find it difficult to understand why most of the ideas present in the Multics hardware were not accepted, refined and included in all current hardware designs.

Chapter V

PERSPECTIVES

Objects provide a service for a user. The service is a high level abstraction of a complex operation. The service provided by the abstraction can be refined by logical partitioning into components to implement the service. This is the top-down successive refinement method applied to the definition of an object. What is interesting about this decomposition is that there usually is a natural grouping of the components based on the functional subset of the abstraction. This grouping forms an intermediate level of understanding during implementation of the abstraction, and arises concurrently with the development of the components. As components are defined the operations they implement indicate which group they belong in, and as the intermediate level develops so does an understanding of the functions required to be implemented by the components. This feedback during program analysis is what normally occurs in developing software.

For example, the OBJECT definition "seqfile" provides a storage service. The components for "seqfile" define the operations that support the storage service and the sequential nature of the storage management. These components fall into the following groups: reading from the file, updating the file, extending the file, and recreating the file. Each group may be implemented by several components.

These inherent groupings can also be used in the implementation of other facilities not directly related to the operation of the abstraction. Two of these facilities are security and arbitration. Security is the prevention of unauthorized access of an object by a user, and arbitration is the determination of how to treat multiple simultaneous requests for an object. Neither security nor arbitration is essential for the definition of an object.

For example, in a "seqfile" the protection mechanism can use the inherent groupings as a way of directly controlling what components can be accessed by a user. By allowing a user to access only the components that are in one or more of the groups, it is possible to establish a security scheme for controlling the access of the records of the "seqfile". Also, by separating the groups into those that contain components that perform reads and those that perform stores, it is possible to determine when concurrent access of the file can occur. These facilities could be implemented directly by enumerating those components that belong to each type of security or arbitration, but the groups provide a concise indirect method of specifying lists of components.

5.1 DEFINING PERSPECTIVES

The groupings associated with an object are usually lost when an object is implemented in a programming language. Normally, programming languages do not support a means of indicating the groupings and associating the components that implement each grouping. This is because the steps necessary in deriving the components are considered as documentation at the programming language level. But as suggested above, these groups can be used to support special features in the system, thus there is a need to be able to define the groupings in the programming language.

A perspective is the name given to a group of components that are logically related by the function that they jointly perform. Perspectives are similar to ALPHARD operation lists [SHAW81]. Usually there are several perspectives defined for an object and this set of perspectives is called the perspective set of the object. Perspectives are ultimately sets of OBJECT definition component names. However, it is frequently convenient to consider one perspective to contain other perspectives; that is, the components in the first perspective are a subset of those in the second. These relationships are established in the perspective set definition. The perspective set is defined along with the OBJECT definition.

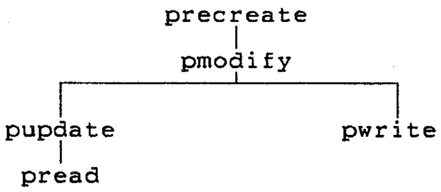
For example, there are perhaps 5 different perspectives associated with "seqfile". (The exact number depends on the implementer's intuitive notion of what really constitutes a "seqfile".) These are:

read : allows a user to read records from the file
write : allows a user to add new records
update : allows a user to read, and change existing records
modify : allows a user to update, and add new records
recreate : allows a user to modify the file or delete all records

These perspectives can be declared and their relationships established by the following declaration entered in "seqfile":

```
PERSPECTIVE p : (pread,
                 pwrite,
                 pupdate > pread,
                 pmodify > pupdate & pwrite,
                 precreate > pmodify)
```

Here, "p" is the perspective variable for "seqfile" through which the actual perspective in a "seqfile" instance can be obtained. A perspective declaration is like an enumerated type declaration in that it declares a set of named constants. As well, the perspective definition provides the ability to indicate hierarchical relationships among the perspectives (i.e. a partially ordered set). The symbol '>' is used to specify the ordering among the perspectives. The form "pupdate > pread" means that perspective "pupdate" implies perspective "pread" (i.e. "pread" is a subset of "pupdate"). A graphical picture of the relationship among these perspectives is:

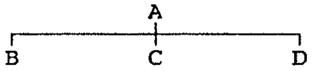


Thus, a perspective forms a partially ordered set; a partially ordered set is a special form of enumerated type and can be named and used to declare variables other than perspectives. However, the type perspective has its own unique properties (as will be discussed) and its type can be any of the enumerated types.

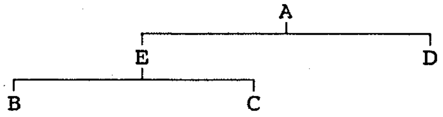
These perspective names can then be associated with the appropriate components of the OBJECT definition that provide the function appropriate for the perspective. For example, to complete the perspective specification for a "seqfile", the following perspectives are associated with components in the body of "seqfile".

```
ACCESS seqacc {pread} ...
  ESCAPE end_of_file {pread}
  PROC read() {pread} ...
  PROC reset() {pread} ...
  PROC write {pwrite} ...
  PROC update {pupdate} ...
  PROC recreate() {precreate} ...
```

Only one perspective is allowed to appear in the perspective set for a component. In general, this should not cause any problems. Situations where more than one perspective needs to be associated with one component are rare, because that would imply that that component belongs in two functionally different groups of components. If this is the case, then it is most likely that there is some intermediate functional level that could contain the two groups. For example, if there are perspectives defining the relationship:



and a component must belong to perspectives "B" and "C", then this can be accomplished by introducing a new perspective "E", such that:



and the component can belong to perspective "E". All other components in "seqfile" that have no perspectives associated with them are local components necessary to support the implementation, and are not part of the functional operations provided by the object to the user.

The definition of perspectives and their association with components are thus far not essential in the writing of the implementation in the programming language, although they serve to define the implementation structure.

5.1.1 Using Perspectives to Restrict Access

The perspective is used to provide information, to both the compiler and the object instance, that only a subset of the actual OBJECT definition is going to be used. This additional information, which is supplied by the programmer, is used to check that only this restricted subset of components is used. As well, at execution time this information about which functional aspects (i.e. which components) of the object are being used allows certain operations to occur which could not occur if unrestricted access to the object were allowed. While it might be argued that the perspective is just another parameter to the object, it is a special parameter because its information is used at compile time as well as execution time. This information is important because it describes at the perspective level how an object will be accessed.

Perspectives can be associated with object references as well as with components in the OBJECT definition. The actual perspective to be used is specified along with an object or object-prototype reference using the following syntax:

```
object-name{perspective-name}
```

The following four examples illustrate some of the situations where a perspective is associated with an object reference to specify a restriction of usage of the object.

1. An object can have a perspective associated with an access variable to restrict what can be accessed through the access variable. For example, in the declaration of the following access variable:

   ```
   VAR pf : f{pread}.seqacc
   ```

   only components in perspective "pread" can be accessed by "pf". This restriction on the operations that can be performed on the file can be used at execution time by the object to both protect itself and schedule concurrent accesses to it.
2. An object parameter can have a perspective associated with it to specify what operations can be performed on the parameter in the body of a procedure. For example,

   ```
   PROC p(VAR g{pread} : seqfile(INT))
   ```

   indicates to the compiler that only components in perspective "pread" will be used in conjunction with parameter "g". This the compiler can check statically in the body of "p". If no perspective is specified for a parameter, the compiler assumes that access may be made to all user visible components.

3. An object can have a perspective associated with it at a call site to specifically indicate a restriction on the type of operations that can be performed on the arguments. For example, in the call:

   ```
   p(f{pread})
   ```

   the caller is indicating that only components in perspective "pread" can be used in conjunction with "f". This allows a user to protect his data when calling a program that he is not sure will modify the argument. This can be checked statically by comparing the call site perspective with the parameter perspective. If the parameter perspective is not a subset of the argument's perspective, a compilation error occurs.

4. An OBJECT definition can have a perspective associated with it to indicate that only a subset of the OBJECT definition is to be implemented. For example, an object which can be used in the same context as a "seqfile" but only provides the components specified in the "pread" perspective of "seqfile" is defined as follows:

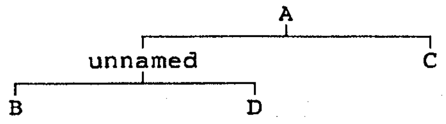
   ```
   OBJECT readfile : seqfiletype{pread}
   ```

   Instances of type "readfile" can only be used in contexts that use the perspective "pread" components on a "seqfile".

As with the perspective for components, only one perspective can be specified to restrict access. This stems from the notion that allowing a list of perspectives would, in essence, be defining a new (although unnamed) perspective with a new relationship. Using the perspective relationship given above a usage such as:

```
VAR pg : g{B,D}.access
```

is in effect creating a new perspective:



Clearly, this is not a combination that the file writer has included in the original definition. As a result, it does not seem restrictive to force a user of a definition to use only those relationships that are defined via the original perspectives for an object.

5.1.2 Implementation of Perspectives

The compilation of the PERSPECTIVE declaration places information on each perspective and on the relationships that exist between the perspectives in the ST entry for the perspective variable.

The compilation of the perspective for components associates information about the perspective with its ST entry. This extra ST field is called the perspective field, and its value is called the perspective value. For components that have no perspective set, the perspective value is set to the special perspective "NONE" which indicates the component belongs to none of the perspectives defined for this OBJECT definition.

The perspective value is the unique value assigned to the names in the partially ordered enumerated set. Like other enumerated types, the perspective constants are numbered from 0 to N-1. Thus, a single byte is probably enough to store the perspective value as that would allow 255 user-defined perspectives plus the special perspective "NONE".

5.1.3 Using the Perspective at Execution Time

As has been shown in the usages of the perspective, the perspective is always associated with a reference to an object. This is because the object is the entity that provides the resource that the perspectives describe. This is expressed by specifying the perspective with the object usage and the perspective is essentially passed as a parameter to the object. However, unlike other parameters that are passed to the object, the perspective is not fixed for the duration of the object but instead changes with each user's specified access. This is another way in which the perspective is special.

The perspective associated with an object is passed implicitly as a parameter to the object when it is made accessible. Since the perspective for each accessor is different, the perspective must be implicitly stored in the data area of the accessor's access class. While the scope rules allow access to the perspective in the object and the access class, the perspective variable has no value during either initialization or termination of the object. This is the same as for the representative and is due to the fact that both perspective and representative are related to access of the object and not to the object itself.

5.1.4 Modifying Perspective Sets

Adding or deleting of a perspective for components is a significant modification. As will be seen, the perspective value for a component is compiled into modules, and hence changing it invalidates the value in a module which results in a significant change. However, adding a perspective to a component that previously had no perspectives is not a significant modification. Since no module could reference the component without an associated perspective, there are no outstanding perspective values to be invalidated.

5.2 ARBITRATION

Scheduling, like paging, is within the domain of the system. Only the system is in the global position needed to assess when resoures can be allocated and used. However, although the system can best decide how the physical resources of the machine can be managed, the system is not able to deal with the logical situations when an object may be accessed by several users. The decision concerning concurrent access of an object is best specified by the object. The decision about when to grant access to and when to delay access to an object and other data is called arbitration. For example, a "seqfile" object may have multiple readers, but only one writer, and an "indexseq" may be the same but on a record basis. Implementing arbitration techniques to support these two types of concurrent access for "seqfile" and "indexseq" requires a different set of arbitration routines to implement each. These arbitration routines must be aware of the logical nature of the object they are arbitrating, and they must be used by the system to determine if an object can or cannot be accessed.

5.2.1 Arbitrator

The arbitrator decides which user will receive access to a shared resource. This is done through the representative which provides a place in which data items for serializing current access can be created, and exist for multiple accessors.

The representative code for arbitration depends on the type of file and the level of arbitration required in the file. For example, in the case of a "seqfile" object, a user is granted or denied access to the whole object. Concurrent read access is allowed, but write access forces queueing of subsequent users. In an index-sequential file, the arbitrator may provide routines for arbitration on individual records as well as the whole object. A user may want access to the whole index-sequential file during loading or wish to have exclusive or shared access to a particular record.

The following example illustrates the additions to "seqfile" to use its arbitrator to handle concurrent access.

```
TYPE seqpersp = (pread,
                 pwrite,
                 pupdate > pread,
                 pmodify > pupdate & pwrite,
                 precreate > pmodify)

OBJECT seqfile( ... ) REPRESENTATIVE arb : seqrep ...
  PERSPECTIVE p : seqpersp

  ACCESS seqacc ...
    VAR pa : arb.seqrepacc(p)
      :
      :
  END ACCESS
END OBJECT
```

The partially ordered set for "seqfile's" perspective is removed from "seqfile" so that it can be used as the parameter type to the representative's access class. When a "seqfile" is accessed, the system implicitly creates a representative and passes the object the specified perspective. "Seqfile" creates an access class for the representative and passes it the access perspective.

5.2.2 Complete Definition of "Seqarb"

The following is a complete definition of "seqrep" which performs arbitration of concurrent accessors for "seqfile".

```
OBJECT seqrep
  RECORD task_info
    t : TASKID,
    p : seqpersp
  END RECORD

  VAR q : ^sys.queue(task_info)
  VAR read_count : POSINT
  VAR write_count : 0..1

  ACCESS seqrepacc(p : seqpersp)
    startarb(p)
  TERMINATE
    endarb(p)
  END ACCESS

  PROC startarb(p : seqpersp)
    IF p = pread THEN
      IF write_count = 1 THEN
        q.rear(task_info(CURTASK,p))
        ^sys.sleep()
      END IF
      read_count +<- 1
    ELSE
      IF read_count ¬= 0 THEN
        q.rear(task_info(CURTASK,p))
        ^sys.sleep()
      END IF
      write_count <- 1
    END IF
  END PROC

  PROC endarb(p : seqpersp)
    VAR t : task_info

    IF p = pread THEN
      read_count -<- 1
    ELSE
      write_count <- 0
    END IF

    IF ¬q.empty THEN
      t <- q.front
      IF t.p > pread THEN
        IF read_count = 0 THEN
          ^sys.wakeup(t.t)
        END IF
      ELSE
        LOOP
          ^sys.wakeup(t.t)
          q.remove
        UNTIL q.empty EXIT
          t <- q.front
        UNTIL t.p > pread EXIT
        END LOOP
      END IF
    END IF
  END PROC

  read_count <- 0
  write_count <- 0
END OBJECT
```

When the representative is created, it declares a queue which will contain the users that must wait while writing occurs. The elements of the queue contain the task identifier of the process that made the request (obtained via the special name CURTASK), and the type of access requested. Each user accessing the object gets his own representative access class which is created as part of the declaration of the access class for the object. The representative's access class is passed the type of access that the user's perspective implies. This perspective is then passed on to "seqrep". The representative will not return control back to the access class and hence back to the user's access declaration until the type of arbitration requested can be granted. If the user must wait, the system routine "sleep" is called to suspend the current process. Two counters, "read_count" and "write_count", count the number of users reading and the number writing. A user wanting to write must wait, until "read_count" is 0 (i.e. no readers).

The termination code is called at the end of the block containing the access variable just before the access variable is deleted as part of normal block termination. The user's perspective is then passed to "endarb". The appropriate counter is modified depending on the type of access indicating that this user's access is over, and the queue is checked to see if there are any users waiting to use the file. If there are users waiting, the first user is checked to see what type of access he had requested initially. If the user had requested other than "pread" access, then he must wait until all users currently sharing the object are done. Only when the last shared accessor finishes accessing the file will a non-shared access process be restarted . This is done by calling the system routine "wakeup" which marks the process as dispatchable and the system may then schedule it for execution. If the user at the front of the queue requested shared access then he and all users following him that requested shared access are dispatched. Hence, multiple users can access the file simultaneously, but only one non-shared access is allowed. Notice that both "startarb" and "endarb" are defined in the object. This is done to force serial access to them during the arbitration process.

The perspective simply supplies information from the access to the arbitrator. It is up to the file writer to ensure that things work according to their stated definition. For example, if a user requests "pread" access to a "seqfile" this does not mean that the system checks that the "seqfile" is not modified. The file writer may wish to increment a counter inside the file to maintain a count of the number of records read. This can be accomplished safely because access to variables in the object are serialized. Thus, perspective "pread" access does not necessarily imply physical read access to an object.

5.2.3 Arbitration of the contents of Objects

In the case of an index-sequential file, concurrent update of the file is allowed and updating a record should not lock the entire file. But individual records may have to be accessed in a non-shared fashion. To perform this type of access to the records requires a queue of users for each record that is being operated on. These queues must be different from the queue used for access to the whole file. Hence, they must be allocated dynamically in the arbitrator.

To handle this situation involves augmenting the representative access class with explicit routines that can be called from the access class routines of the OBJECT definition "indexseq". For example, the representative may supply a lock record routine (and an unlock record routine) that is passed a record key. It would then check to see if that particular record has a wait queue created for it in the representative. If it does, the write is queued and the caller waits, otherwise a queue is created and execution continues allowing the access to the specified record.

5.3 SECURITY

Security is necessary within all systems, not only to protect sensitive information, but to protect against accidental or incorrect use. Security should not be an optional mechanism, where entities are normally unprotected and may be protected from users by explicitly invoking the security mechanism. Instead, security should be provided automatically, where entities are protected at creation and must be explicitly unprotected or exposed for each user wanting to access the entity. This latter scheme requires more effort to allow sharing by others, but in general provides a safer system.

Other general requirements essential for security are:

1. Exposure must be able to be done on a selective basis. A user may wish to have finer control over exposure then just total exposure or no exposure. Complex entities provide many components and the user may wish to expose only selective groups of components to another user.
2. The definition of a user must be reasonably flexible so as not to be too restrictive as to what sort of entities can perform access.
3. The ability to rescind access from a user must be supported, and this rescinding must be reflected at the next access.

To achieve these goals a dynamic security check will be used. Although attempts have been made to perform security checks at compile time [JONES78], the nature of security demands that security will be updated and changed. This would imply recompilation whenever security on an OBJECT definition changed. These security modifications are independent of the definition of an entity, and therefore as it is difficult to justify re-compilation of an entity or its usage to reflect security modifications.

5.3.1 **Using Perspectives to Expose Components**

Perspectives provide an ideal mechanism for selectively exposing components of a cluster definition. This is done by the creator of a cluster granting a particular user one or more of the perspectives defined in the cluster definition. This allows the user to access the components defined by the perspective for that instance of the cluster. Using the perspectives to expose components allows a precise control over how each user of the entity can access the entity via its components. It is important to note that because perspectives are defined to be associated with a cluster definition, simple data items cannot be protected directly, but are protected by the cluster definition that contains them.

5.3.1.1 Users

In conventional systems a user is an entity that is “in effect” for some duration (usually a terminal session) and while in effect can access entities authorized for that user. As well, each user must be unique so that one user cannot impersonate another. In this system, the software entity that most resembles a conventional user is an environment; however I do not wish to base security solely on environments. In this system, security may be based on other DS's besides those corresponding to users. These DS's are called security DSs. It is the MH address for a DS that serves as the unique identifier for it, and when this DS is accessed it determines whether access is allowed or not and as long as it is accessible it is the one on which all security decisions are based. A DS used for security checking comes “into effect” when it is accessed. When access to an object is attempted, the MH address of the DS is used to determine whether the access should be allowed or not. As long as the DS remains accessible, it is used in this way for security checks unless it is superceded by another such DS. Thus, objects can be granted access to other objects. Notice that an object DS is used implicitly for security checks like a conventional user; however, it is not necessarily constant throughout a terminal session. By changing environments, the DS used for security checks can change dynamically.

5.3.1.2 Passwords

Passwords provide a secondary mechanism to associate security with cluster instances. Passwords provide the same protection as users but they are explicitly specified on a reference. As with a security DS, a password and perspective are associated with a cluster. The password must be specified explicitly wherever the instance is accessed and it is used for the security check instead of the current security DS.

5.3.1.3 GRANT Statement

The GRANT statement is a programming language statement used to associate a perspective of a cluster definition with a user or password. It is not a PE command because the grant statement must refer to the cluster instance to associate the user or password with it. Since the cluster instance is only available in SEM, the grant statement must be used in interactive mode and hence must be a programming language statement. As well, to be able to associate security with dynamically created clusters would require that the grant statement be a programming language statement.

The syntax of the two forms of the grant statement are:

1. GRANT cluster-instance{perspective} TO user-list

   Which gives to the users in "user-list" the right to access the components of "cluster-instance" defined by the "perspective".
2. GRANT cluster-instance{perspective} WITH password-list

   Which gives to the user of a password from "password-list" the right to access components of "cluster-instance" given by the "perspective".

For example, suppose user "buhr" wishes to declare a sequential file "f" in his environment and make this file available to user "zarnke" to read. The file "f" is declared in "buhr" and the following grant is executed by "buhr" to expose file "f" with read access:

```
GRANT f{pread} TO ^sys.zarnke
```

Thus, when "f" is accessed in environment "zarnke", the components "seqacc", "read", and "reset" may be used.

5.3.2 Using Perspectives to Expose Environments

Granting a user a perspective from a cluster means that the user can access those components to manipulate a particular cluster, but only if the user can gain access to the environment containing the instance. In the case where the instance is located in the user's environment, he can directly address the instance because of the scope rules. However, to gain access to entities in other environments requires obtaining their addresses from that environment, and this, in turn, requires that the user be authorized to access the necessary environment components to extract and use the address.

In fact, before a user can access any entity, he must be able to access all environments along the logical path between his environment and that containing the entity. Hence, multiple grants may be necessary to allow a user to access an instance in another environment. However, these grants need only be done once to establish an access pathway between the environments. Once the pathway is established, grants are only necessary to make new entities inside the environment visible to the user.

5.3.2.1 Environment Perspectives

The environment is considered to be an object, albeit a rather special object. The specification for an environment allows a perspective set to be defined. The perspectives defined in the perspective set can then be used to expose components defined within that environment. In this way, components can be made accessible to other users by granting access to them via the environment perspectives.

Because an environment collects together many unrelated components, its perspectives do not necessarily describe functional subsets of these components. Instead, the perspectives are used merely to partition the components declared within the environment into different groups for the purpose of controlling exposure of these components to other users.

Because each environment instance is unique and hence has its own ST, each user can declare his own perspective for it. For example, if the user edits the environment perspective and adds a new enumerated constant, called "extern", that would create a perspective for the environment. Thereafter, any component declared in the environment can have perspective "extrn" associated with it. This perspective may be used to allow components to be exposed to another environment.

For example, to expose sequential file "f" outside environment "buhr", perspective "extrn" is associated with "f" at the declaration:

```
E: VAR f{extrn} : seqfile(INT)
```

To grant to user "zarnke" the right to access the components defined by perspective "extrn" use the command:

```
I: GRANT buhr{extrn} TO zarnke
```

This exposes to user "zarnke" all components in environment "buhr" that have a perspective that is a subset of "extrn". It is up to user "buhr" to ensure that only the entities that he wishes "zarnke" to see are associated with perspective "extrn" or its subsets. This grant statement is specified only once. Any components that are associated with "extrn" in the future automatically are exposed to user "zarnke".

Even further grants will be required if more environments exist along the environment pathway between environments "buhr" and "zarnke". If there is a common father environment in which environment "buhr" and "zarnke" are declared (most likely in "sys"), then any references from one to the other must “pass through” the father environment. To pass through the father environment means that information about the locations of instances must be extracted from the father environment, which implies that components of the father environment must be accessed. Since all components are by default not exposed, any user wishing to address components must be granted appropriate access.

The ability to address one sub-environment from another within the father environment is accomplished through a particular perspective associated with "sys". For example, to expose "buhr" and "zarnke" to each other the environment perspective "puser" of "sys" is associated with "buhr" and "zarnke" at their declaration:

```
E: VAR buhr{puser} : user
E: VAR zarnke{puser} : user
```

The perspective "puser" can now be granted to environment "buhr" and "zarnke" so that they can refer to each other:

```
I: GRANT sys{puser} TO buhr
I: GRANT sys{puser} TO zarnke
```

This exposes all components in environment "sys" with perspective "puser" to users "buhr" and "zarnke". These grant statements are specified only once to associate sub-environments with perspective "puser". Any new sub-environments that are associated with "puser" in the future automatically have an access pathway to all previously associated sub-environments. These grants make it possible for user "zarnke" to dynamically access "buhr.f". However, although user "zarnke" can address "buhr.f" he as yet cannot compile references to entities in environment "buhr" because "buhr" has not granted appropriate access.

5.3.2.2 ST Perspectives

The ST is a complex entity manipulated and maintained by the PE. The ST is considered an object with a data area containing the ST information, and routines performing operations like:

1. searching and extracting information from the ST
2. adding entries into the ST
3. changing entries in the ST
4. deleting entries from the ST

The OBJECT definition for a ST allows a perspective set to be defined and these ST perspectives can be used to grant access to the ST operators. For example,

```
GRANT buhr'ST{search} TO zarnke
```

exposes environment "buhr's" ST to user "zarnke" for searching and extracting information. Individual entries in the ST cannot be associated with perspectives because they are not components of the ST object. They are just data items within one of the data area components of the ST. Thus, protection of the ST information is on a gross scale compared to the protection of environments. Another user can see or not see, add or not add, delete or not delete information from the ST. This gross type of protection normally does not pose a problem as access to a ST does not give the right to access the data instance whose names appear in the ST.

Like environment perspectives, appropriate grants of access to the STs must be made before a user can reference entities at compile time in another user's environment ST. However, a user does not necessarily have to refer to an entity by name to access the entity. He might obtain a pointer to the object from another user and thus access it without having to refer to it by name.

Once all the grants are performed, user "zarnke" can compile and access "f" from his own environment. For example, user "zarnke" can pass "f" as an argument to a routine:

```
I: p(^sys.buhr.f)
```

or use "f" in a declaration:

```
E: VAR pf : ^sys.buhr.f.seqacc
```

When these statements are compiled and executed, a dynamic security check occurs to verify that environment "zarnke" is authorized to access the specified components in the father environment and environment "buhr", and that only components associated with perspective "pread" are used with sequential file "f".

5.3.3 Using Perspectives to Expose Objects with Passwords

The previous example where user "buhr" wished to allow user "zarnke" to read from sequential file "f" will be used to demonstrate how passwords are used. In this example, all the users will simply be replaced with passwords.

First, user "buhr" must define a password for reading from "f":

```
GRANT f{pread} WITH rdpwd
```

Secondly, "f" must be able to be seen outside of "buhr", so "f" is the environment perspective "extrn" when it is declared and the following password is created to expose it:

```
GRANT buhr{extrn} WITH extrnpwd
```

These passwords can then be communicated to user "zarnke". (The father password is omitted here for simplicity.)

Once user "zarnke" knows the necessary passwords, he can access "f" from his own environment by explicitly specifying them, the syntax of which appears in the following example. For example, "f" can be passed as an argument to a routine as follows:

```
p(^sys.buhr:extrnpwd.f:rdpwd)
```

Execution of this routine call causes a dynamic security check to be done; it extracts the perspective corresponding to the password and uses it for checking legitimacy.

Passwords have the advantage that they can be used with any entity. They have the disadvantage that they can be stolen and misused. However, passwords must be used in situations when there is no current user yet, for example at sign-on. Here, the user must specify a password to obtain access to his environment:

```
I: logon(buhr:password)
```

5.3.4 Rescinding

Rescinding means to narrow the access a user has to an instance. This must be done without forcing re-compilation of pre-compiled modules. Since all security checks occur at run-time, changes to security are reflected immediately even in pre-compiled modules. To rescind visibility, the GRANT statement can be used by specifying a new perspective that allows access to fewer components or by specifying the special perspective "NONE". The special perspective "NONE" would remove the user from the list of granted users so that he is no longer entitled to access the instance.

There is one potential problem that can occur if a user is granted access to an instance and then the user is deleted. The grant statement to remove this user from the capsule will be incorrect because the user's name is undefined. Hence, it would not be possible to remove this entry. This does not represent a serious problem because the user is gone and so are all of his modules; therefore, not removing the user's name causes no problem other than the space used by the entry in the list of users.

However, there is another esoteric problem that can arise. Since the storage is freed for the deleted user, it might be used in the creation of a new user. Thus, the new user might have the same address as the old, and there is the potential that this new user could now impersonate the old user. However, this is prevented because the TS of the new user is always further in the future then all references to the old user and hence all old references will fail.

5.3.5 Implementation of Security

Because the security check is dynamic, it will be performed frequently. Hence, it is desireable to make the security check as efficient as possible.

5.3.5.1 Using AT Entries to Store Security Information

For each cluster definition that has perspectives, there can be a list of users granted access to an instance of the cluster definition. This list must be stored so that it is accessible at execution time to perform the security check. As well, this list must be protected so that it cannot be modified except through the proper security mechanism.

To satisfy these requirements the granted users or passwords are stored in the capsule that maps the memory which contains the cluster. This list is then accessible at execution time because the memory that contains the capsule is made accessible incidentally to make the object accessible. As well, the list that is being accessed is not part of the cluster or even the underlying memory which contains the cluster. Hence, the user cannot get access to the granted user or password lists.

The capsule for a memory must hold and identify user and password information about all instances contained in the overlying memory. For example,

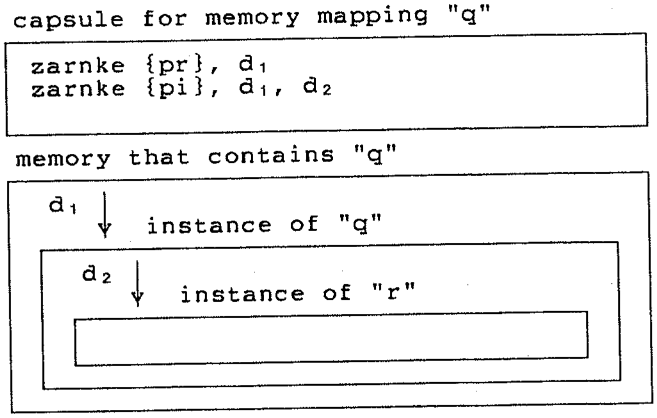
```
OBJECT Q {extern}
  PERSPECTIVE pQ : (pr)

  RECORD R
    PERSPECTIVE pR : (pi)
    VAR i {pi} : INT
      :
  END RECORD

  VAR r {pr} : R
    :
END OBJECT

VAR q : Q
GRANT q{pr} to zarnke
GRANT q.r{pi} to zarnke
```

Here, the capsule underlying the memory for object "q" would contain security information about both "q" and "r". A diagram of this situation looks like the following:



Notice that extra information is needed in the capsule to identify which cluster instance the perspective is actually associated with. This extra information is just the displacement of the cluster within the memory that contains it.

5.3.5.2 Granting

The grant statement adds information into the capsule entry which underlies the memory that contains the cluster. As described so far, the only user that is allowed to grant access to a cluster is its creator. When an object is allocated, the address of the current security DS is entered into the capsule as the owner. Part of the execution of a grant statement is to verify that the grant statement is executed by the owner of the object.

If it is desirable to have more than one user grant access to an object it would be necessary to introduce a new predefined perspective (similar to "NONE"), such as:

I: GRANT f{ALL} to zarnke

Adding another owner would be implemented by adding another entry to the owner field in the capsule. Currently, the grant statement does not allow this type of modification. At the moment, I am not convinced that multiple owners of an object are necessary, and so I will defer this issue until more knowledge about security can be acquired from the system implementation.

The value for the perspective entered into the capsule by the GRANT statement is not the perspective value, but rather a SET value derived from the perspective. This SET value is a bit string that is constructed by the compiler during compilation of the GRANT statement. The construction of the bit string begins by assigning a unique bit for each perspective in the perspective set. For perspectives Pi this value is:

$$B_i = (0 \ldots 0 \; 1_j \; 0 \ldots 0)$$

where "j" is the perspective value. The bit string is defined as the logical OR of all the Bi which are descendents of the perspective in the GRANT statement including the perspective in the GRANT statement. This bit string describes all the perspectives and hence all the components accessible by this user or with this password.

Compilation of the grant statement generates the necessary object code to enter the constructed bit string and either the password(s) or the address of the user(s) plus any additional displacements into the capsule that underlies the memory containing the cluster. This means that the grant statement is performing an operation at the capsule level. In fact, each capsule could be required to provide a routine that is called, possibly only by the system, to perform this updating of the security information in the capsule. Execution of the grant statement causes the security information in the capsule to be searched for the specified user. If the user is not found an entry is created and the user and perspective are inserted. If the user is located, his corresponding perspective is updated to the perspective specified in the GRANT statement. This handles rescinding and augmenting of access. The special case of associating perspective "NONE" with a user causes that entry to be removed from the capsule security information.

5.3.5.3 Security Check

The dynamic check to verify that a user accesses only those components that his granted perspective allows is called the <u>security check</u>. This check is performed implicitly, and is composed of two parts: obtaining the necessary information so that the check can be performed, called <u>security extraction</u>, and using this information to validate references to components, called <u>security comparison</u>. The information necessary for the security check are the <u>user's perspective</u> and the <u>component perspective value</u>. The user's perspective is extracted from the capsule underlying the memory containing the cluster to be accessed and the component perspective value is obtained from the module's object code, which in turn is obtained from the perspective field in the ST for the component.

As instances are accessed during MH address resolution, the security check is performed. During MH address resolution, the ARR performs the security check as part of making each object accessible. As well, the compiler may have to generate some further final checks for those instances that are contained in the last AS of the MH address. These are the instances whose security information is in the capsule for the memory of the last object accessed.

5.3.5.4 Security Extraction

Security extraction is done before each access. For an object like a file this means only once per access. For simple variables the extraction must occur for each reference. Hence, if a user's level of security changes which updates the information in the capsule it will not become effective until the next access to the entity.

Security extraction involves searching the capsule for all entries that contain either a specified password or the user that is currently making the reference. If found, the displacement of the capsule instance in the overlying memory must be checked with the next displacement in the MH address of the reference. If they are the same, the corresponding granted perspective is extracted from the capsule for use in the security comparison.

The extraction of the user's perspective from the capsule occurs at the point where the address is obtained. In the case of making an instance accessible, such as:

```
E: VAR pf : f.seqacc
```

extraction of the perspective occurs as a normal part of the referencing of "f". However, when an object is passed as an argument, such as:

I: p(f)

the extraction of the user's perspective must occur at the call site. Here the MH address of "f" is extracted from its containing DA at the call site and passed to routine "p". This address is normally not used until the file is accessed in the routine "p". However, it is necessary to use it here to extract the perspective for the user from the capsule underlying "f". The perspective must be extracted at the call site since the security DS may change by the time "f" is actually accessed. Also, if the routine is recursive the user at the original call should be the same as the user at subsequent calls. Thus, to insure that the user that is making the request is the one that is used in the security check, the user's perspective must be extracted at the call site. Although passwords are not sensitive to changes in active ASs and could be passed along with the MH address they are instead looked up at the call site to be consistent with what is done for users.

5.3.5.5 Security Comparison

Whenever a data item is declared with a perspective, all subsequent references to it will have security checks done. The security check must be done for every access to an instance. For simple variables, where access duration is not explicitly specified, the access check occurs for every reference. For instances whose access duration is explicitly specified, the security check occurs at the beginning of access and that level of security persists until that access terminates. Thus, the compiler must generate security checks at access points. In the case of a simple instances this may substantially increase execution time as a security check must occur for every reference. However, I feel that perspectives will normally not be associated with simple instances. Security will normally be associated with objects and objects must be made accessible. Since the security check is done during MH address resolution it will not cause a substantial increase in execution time to the process that must already occur.

Once the two parts needed for the security comparison are extracted, the security comparison is performed. This check is simply indexing into the user's perspective for the bit specified by the component's perspective. If the bit is set, the component is a subset of the user's perspective and hence execution can continue; otherwise an exception is raised. Thus, the security check can be done efficiently.

The security check can now be illustrated with an example. In the following example user "zarnke" is attempting to use file "buhr.f" by passing it to an already compiled procedure "p" in environment "zarnke".

```
I: p(^sys.buhr.f)
```

In compiling this routine call, the compiler must traverse the ST tree structure verifying the existence and correct type usage of all the fields in the qualified variable. Once "f" is located in the ST of "buhr" and the type is compatible with the type of the parameter of "p", the compiler generates the object code necessary to reference "f" from "zarnke". The compiler generates as part of the code the perspective value for each appropriate field of the qualified variable.

At run-time, the MH address of "sys" is extracted from the containing pointer in "zarnke" and the capsule for it is accessed. The security check routine looks up the active DS ("zarnke") in the capsule of "sys", finds it, and extracts perspective "puser". This is compared to the perspective value in the object code from the perspective field of "buhr". In this case, it is the granted perspective "puser", so the MH address of "buhr" is now extracted from "sys" and the capsule for "buhr" is accessed. The security check routine looks through the capsule of "buhr" for "zarnke". The MH address for environment "zarnke" is found there and the perspective "extrn" is extracted. This is compared with the perspective field of "f" which is "extrn". The MH address for "f" is extracted from "buhr" and the security check looks up "zarnke" in the capsule for "f" and extracts perspective "pread". Finally, the MH address for "f" and the perspective "pread" are passed to routine "p".

5.3.5.6 Specified Perspectives

Perspectives may be explicitly specified on a reference to indicate the type of operations that will be performed on an instance. This extra information is interpreted as the minimum exposure necessary for execution to proceed normally. For example,

```
f{pupdate}
```

means that only the update operations of "f" will be used. This involves comparing the explicitly specified perspective with the user's or password's perspective to determine if the specified perspective is a refinement of the user's perspective. This relationship is determined from the hierarchical structure of perspectives defined in the perspective set. If the user does not have at least the specified access, then an exception is raised.

When the compiler cannot check perspectives at compile time, it inserts object code at the point of the specified perspective to verify that the user of the object has at least the exposure indicated. For example, in the declaration:

```
E: VAR pf : f{pupdate}.seqacc
```

the user's perspective is extracted from the capsule of "f", or if "f" is a parameter then the user's perspective was extracted at the call site and passed as part of the address of "f". As part of the declaration, the granted perspective is compared with perspective "pupdate" to verify that it is a subset of the granted perspective. As well, the specified perspective is now used as the granted perspective, instead of the user's perspective. This is because all code that uses "pf" may be compiled under the assumption that the user has "pupdate" access or one of its descendents. If "pf" was passed as a parameter then the perspective "pupdate" would be passed along with it instead of the granted perspective.

The comparison of the granted perspective and the specified perspective involves checking if the specified perspective is a refinement of the perspective associated with the user. This comparison is done in the same way as for the security check.

5.4 SUMMARY

Perspective definitions provide a powerful mechanism for grouping together components of an object and forming relationships among the groups. Since these groupings are stated in the OBJECT definition, they provide a precise mechanism for specifying what operations can be performed by each group. The perspective is then available for use by the compiler and by modules at execution time in order to check access legitimacy. This information can be used in many ways by the compiler and modules not all of which were enumerated in this chapter.

Representatives use the perspective to infer the type of modifications that will be made to an object. This information is then used to arbitrate among multiple concurrent accessors to an object. Arbitration is done in the object representative because only it exists for the proper duration and because its data area is transiently allocated.

The security mechanism uses perspectives to control which users can access an object and what type of access they are allowed. This is done indirectly by controlling the groups of components to which a user has access. Since perspectives are peculiar to each OBJECT definition, they provide a specific and precise way of controlling user access. This is in contrast to security mechanisms provided by most operating systems that define one general perspective for use with all entities.

Chapter VI

CONCLUSION

> The most critical reader of all, myself, now finds many defects, minor and major, but being fortunately under no obligation either to review the book or to write it again, he will pass over these in silence, except one that has been noted by others: the book is too short.
>
> — J. R. R. Tolkien, The Lord of the Rings, Forward

While this concludes the written discussion of the programming system, this simply marks the end of the beginning. The most important phase of the thesis is about to begin: implementing the ideas. I know full well that ideas are cheap, but ideas written down and thought out may have some value. Yet, as always, the proof is in the pudding, not in the idea for a pudding or the recipe for a pudding.

Due to the lack of implementation it is not possible to derive any quantitative conclusions. Figures on the efficiency and usability of the system will have to wait until after the forth-coming implementation. Even then it will be difficult to judge because of the magnitude of the comparison. How does one compare two entire programming systems, and on what criteria does one decide whether one is superior to another? Therefore the conclusions that are drawn are subjective ones.

As well, during implementation I am sure that some of the ideas presented will require some modification. Some problems are difficult to anticipate no matter how much forethought is given. Only during the formalization of these ideas in a programming language may some of the more subtle and intricate problems surface.

Finally, the implementation of the system will likely have to be done on hardware that does not provide the necessary addressing capabilities. To the author's knowledge, no existing hardware system provides all hardware requirements specified in chapter 4. Existing hardware systems provide only some subset of the hardware requirements, and as a result some important part of the addressing mechanism may have to be simulated by software, in particular, the ability to have multiple accessible ASs. This software simulation may impose a limitation on the usefulness of the system because of the overhead involved.

6.1 REVIEWING THE OBJECTIVES

Providing file definition in the programming language was one of the key issues that allowed much of the integration of the programming system. File definition allows movement of the conventionally predefined file organizations and file hierarchy out of the operating system. These facilities are no longer cast-in-stone, but instead are accessible via the programming language. It is not imagined that all users will want to define new file organizations, but large scale application development may need to tailor their own file organizations. This can be accomplished and the new tool that is produced will be immediately accessible via the programming language.

This accessibility is multi-fold. The new file is accessible through the ST hierarchy so that all programmers can refer to it. The programming language allows the declaration of the new file in both PDM and in SEM. Finally, if the new file is based on an existing file prototype, the new file can be used with any existing modules that worked with that prototype.

The programming language used interactively provides a high-level, powerful, and consistent interface between the machine and the user. Coupled with the concept of an environment, the user is provided with a powerful interactive type-safe mechanism for accomplishing work on the machine.

The PE supplies a controlled and consistent mechanism for program development. It provides a controlled interface between the system and the user that allows a structured method of module development and detection of changes that affect consistency.

The PE and the segment allow separate compilation of modules in the programming system. The user is allowed to choose the compilation entity (down to the level of the procedure), and hence control the scope of compile time detection of changes. Because the programming language is used as the interactive language all parts of the system can be compiled.

The retention of all STs and the ST hierarchy provides the means to allow type checking of all references in the system. This includes references to variables, programs and files in the user's environment, in another user's environment and in the system.

The MH address provides a consistent approach to addressing data in the system. By checking the TSs associated with the addresses it is possible to detect any address that is no longer consistent with the entity it is referencing. This assures that type safety can never be violated when separately compiled changes are made within a constantly changing programming environment.

6.2 ADDITIONAL WORK

There are several areas in the programming system that were not specifically addressed due to the time constraints imposed in writing a thesis. Two of the more important topics, which were briefly mentioned, were versions of modules and the debugger. Both of these area are not essential for the programming system to work, but are necessary to make the programming system a truly comfortable place in which to do development.

6.2.1 Versions

There are many different kinds of versions possible [TICHY82]: versions of source language for a module on the same or different machines or versions of object code for the same or different machines. Providing tools to manage versions and to allow their specification in commands to edit or compile particular versions is complex. However, this programming system provides an ideal place to implement versions: the PE. Because the PE handles all access to STs and has some control over segment usage, the PE should be able to be augmented to provide versions of source language and object code. The extra commands needed to handle versions and the extra information necessary in the ST and the segments for them is best handled by the PE.

6.2.2 Debugger

The debugger is expected to be an interactive tool that can be used to aid module development by allowing a programmer to step through a module and examine data areas, or used after an error has occurred to examine data areas to determine the cause of the error. With the availability of the entire ST structure and data areas pointing to STs for which they are instances, the debugger can dynamically map data areas that are currently active. This allows symbolic access to the data areas in exactly the same way as in normal programming language statements. The debugger would allow stepping through the executable statements by allowing breakpoints to be specified at the source language level by having a statement map created by the compiler. At the breakpoint, the user could examine data items for all active modules, and the user may be allowed to change the values of data items which can be done safely because the debugger has access to the type of each data item. As well, it will be necessary to alter control flow in a structured way by escaping from control structures or modules. This would allow normal block termination code to be executed and hence maintain the integrity of the executing module.

In the case of a module that is executing and encounters an error, it is possible to imagine the system suspending the module but retaining all the active representatives associated with the module. This in effect saves the complete environment that existed when the error occurred. (This might cause problems if a user had write access to a public file, as no further accesses to the file would be allowed until module termination.) The debugger is then called automatically. The user can then examine and modify both data values and control flow, and subsequently allow the module to run until completion. In the case of large production runs, this might allow skipping a data value that caused an error and continue processing without having to restart the module.

6.3 CONTRIBUTION TO COMPUTER SCIENCE

The programming system presented in this thesis is a forerunner of programming system development to come. This programming system is aimed at a more general type of application than other currently available programming systems. LISP [TEITE75] systems provide a single programming system for AI development. APL systems provide an environment for mathematical development. While all of these systems can perform equivalent tasks, each has its own forte.

This programming system is designed to handle large, complex applications (including the programming system itself) such as specially designed systems to handle particular application problems (i.e. accounting system, medical systems, etc.), programs developed to execute quickly and yet that do not change dramatically over short periods of time. These programs are usually characterized by large development time, and a long period of maintenance. This programming system addresses some of the problems associated with both developing a large application and maintaining it.

However, this system is intended not only for developing large applications by an end user, but also for the subsequent use of these applications by an end user. This type of user may never use any of the program development facilities (programming language, program editor, etc.) described in this thesis. He might simply sign on and invoke a single application. However, he may soon wish to use several applications that interact with one another. This system should allow such integration to be done simply and safely (as in UNIX[4]). This would then lead him to simple uses of the program development system in order to create programs in which several applications interact. The same level of integration that was achieved by the program development facilities should be achievable for the application programs that are ultimately created on the system.

Specifically, this programming system is intended to replace the programming language(s), operating system software, and interactive command language provided in a conventional computer system. To

---

[4] UNIX is a Trademark of AT&T Bell Laboratories.

accomplish this, many of the ideas presented have analogues in conventional systems. This only makes sense as many of the ideas necessary to achieve an operational system are the same in both cases. These ideas have developed over many years as a result of solving difficult problems. So in this regard some of the algorithms presented are widely known and used.

What is new is the way in which these ideas have been integrated. The integration of these ideas required a substantial amount of engineering to ultimately produce the designs presented in the thesis. Many approaches were investigated and were rejected. This point was also made about the UNIX[5] system:

> The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system [RITCH74].

It would be nice to say that the approach presented in this thesis is the best approach, however this remains to be seen. Whether it is the best or not, what it does do is to open up a new approach in integrating the many components necessary to form a programming system.

---

[5] UNIX is a Trademark of AT&T Bell Laboratories.

Appendix A

PROGRAMMING LANGUAGE SYNTAX

In this system the programming language is processed by two different programs. The PE is used to enter all declaration and to modify these declarations. These declaration are parsed and executed to create or modify ST entries immediately. Hence, the PE has a grammar for these declarations. While the PE is also used to enter the source language statements, the PE does not parse these. This is done by the compiler. Hence, the compiler has a grammar for the executable statements in the programming language. Thus, there are two grammars: one for the PE and one for the compiler.

When syntax is specified, the notation used is the following:

- quotes (') enclose a terminal symbol
- [x] means that "x" is optional
- x | y indicates that "x" and "y" are alternatives (i.e. one of "x" or "y" is chosen)
- { } is used to group alternatives within a single definition

The following is the grammar used by the PE.

```
module-defn ::= cluster-defn compile-type

compile-type ::= 'COPIED' segment-spec | 'SINGLE' segment-spec | 'INLINE'

segment-spec ::= segment-name | '*'

creation-cmnd ::= cluster-defn [ positioning-information ] |
                  type-defn [ positioning-information ] |
                  declaration [ positioning-information ]

cluster-defn ::= 'RECORD' [ 'PROTOTYPE' ] name |
                 'PROC' [ 'PROTOTYPE' ] name |
                 'CLASS' [ 'PROTOTYPE' ] [ 'PROCESS' ] name |
                 'ACCESS' [ 'PROTOTYPE' ] [ 'PROCESS' ] name |
                 'OBJECT' [ 'PROTOTYPE' ] [ 'PROCESS' ] name |
                 'ENVIR' [ 'PROTOTYPE' ] [ 'PROCESS' ] name |
                 'MEMORY' [ 'PROTOTYPE' ] name

type-defn ::= 'TYPE' name '=' type-constructor

type-constructor ::= type-name |
                     '(' ordered-set ')' |
                     subrange |
                     '[' type-list ']' type-name

type ::= type-constructor | cluster-type

type-list ::= type-list ',' type | type

cluster-type ::= 'RECORD' | 'CLASS' | 'OBJECT' | 'ENVIR' | 'MEMORY'
```

```
declaration ::= 'CONST' name-list [ attribute ] |
                'VAR' name-list [ attribute ] |
                'ESCAPE' name-list |
                'PARAMETER' name-list [ { attribute | ':' 'TYPE' } ] |
                'KEYWORD' name-list [ { attribute | ':' 'TYPE' } ] |
                'RETURNS' name-list [ attribute ] |
                'CREATES' name [ attribute ] |
                'REPRESENTATIVE' name [ attribute ] |
                'SEGMENT' name-list |
                'PERSPECTIVE' name [ attribute ]

attribute ::= [ '{' perspective-name '}' ]
              [ ':' type ]
              [ '<-' initialization ]
              [ '¢' comment-text ]

positioning-information ::= 'AFTER' name | 'BEFORE' name
```

The executable statements that the compiler processes are entered as text during program editing and the PE places them in the initialization code associated with the relevant ST definition. Thus, the PE treats the "initialization" simply as a character string(s); but during compilation the compiler parses them according to the following grammar:

```
initialization ::= statement-list

statement-list ::= statement-list ';' statement |
                   statement-list EOLN-symbol statement |
                   statement

statement ::= assign-statement |
              'IF' if 'END IF' |
              'CASE' case 'END CASE' |
              loop-prefix 'LOOP' loop 'END LOOP' |
              'EXCEPTION' exception 'END EXCEPTION'
              'WITH' primary 'DO' statement-list 'END WITH'
              'SIGNAL' escape-name

assign-statement ::= primary [ binary-operator ] '<-' assignment-expr |
                     '(' primary-list ')' '<-' expression
assignment-expr ::= assignment-expr |
                    expression binary-operator assignment-expr |
                    expression

expression ::= expression binary-operator primary |
               unary-operator expression |
               '(' expression ')'
               primary
               cast

expression-list ::= expression-list ',' expression | expression

primary ::= identifier |
            identifier call |
            identifier subscript |
            identifier qualification |
            literal

primary-list ::= primary-list ',' primary | primary

identifier ::= const-name | var-name | parameter-name | keyword-name |
               return-name | creates-name | representative-name |
               procedure-name

call ::= '(' [ expression-list ] ')'
```

```
subscript ::= subscript '[' expression-list ']' | '[' expression-list ']'

qualification ::= qualification '.' primary | '.' primary

cast ::= [ 'REF' ] type-name '(' expression ')' |
         type-cluster '(' primary ')' |
         record-name '(' expression-list ')'

type-cluster ::= class-name | access-name | object-name | envir-name |
                 memory-name

if ::= expression 'THEN' statement-list |
       expression 'THEN' statement-list 'ELSE' statement-list |
       expression 'THEN' statement-list 'ELIF' if

case ::= primary case-list 'OTHERWISE' statement-list

case-list ::= case-list 'WHEN' chooser-list 'DO' statement-list |
              'WHEN' chooser-list 'DO' statement-list

chooser-list ::= chooser-list ',' chooser | chooser

chooser ::= expression '..' expression | expression

loop-prefix ::= [ 'FOR' identifier [ '<-' expression ] ]
                [ 'BY' expression ]
                [ 'TO' expression [ 'FINI' statement-list ] ]

loop ::= loop ';' loop-statement |
         loop EOLN-symbol loop-statement |
         loop-statement

loop-statement ::= statement | until

until ::= 'UNTIL' expression [ 'THEN' statement-list ] 'EXIT'

exception ::= statement-list escape-list

escape-list ::= escape-list 'ESCAPE' escape-name statement-list |
                'ESCAPE' escape-name statement-list
```

REFERENCES

ANDRE83 Andrews, G.R., Schneider, F.B. "Concepts and Notations for Concurrent Programming". *ACM Computing Surveys*, vol. 15, no. 1, March 1983, pp. 3-43

APPEL84 Appelbe, W.F., Rann, A.P. "Encapsulation Constructs in Systems Programming Languages". *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, April 1984, pp. 129-158

BEECH79 Beech, D. "Command Language Directions". *Proceedings of the IFIP TC 2.7 Working Conference on Command Languages*, Lund, Sweden, September 1979

BENSO69 Bensoussan, A., Clinger, C.T., Daley, R.C. "The Multics Virtual Memory". *Second Symposium on Operating Systems Principals*, Princeton University, October 1969, pp. 30-42

BOURN78 Bourne, S.R. "The UNIX Shell". *The Bell System Technical Journal*, vol. 57, no. 57, part 2, July-August 1978, pp. 1971-1990

DAHL72 Dahl, O.J., Hoare, C.A.R. "Hierarchical Program Structure". *Structured Programming*, Academic Press, 1972, pp. 175-220

FELDM79 Feldman, S.I. "Make – A program for Maintaining Computer Programs". *Software-Practice and Experience*, vol. 9, April 1979, pp. 255-265

FRASE83 Fraser, C.W., Hanson, D.R. "A High-Level Programming and Command Language". *SIGPLAN Notices*, vol. 18, no. 6, June 1983, pp. 212-219

GOLDB83 Goldberg, A., Robson, D. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, May 1983

HANSE73 Hansen, P.B. *Operating System Principles*, Prentice-Hall, 1973, pp. 133-153

HANSE76 Hansen, P.B. "The Solo Operating System: A Concurrent Pascal Program". *Software-Practice and Experience*, vol. 6, July 1976, pp. 141-149

HOARE74 Hoare, C.A.R. "Monitors: An Operating System Structuring Concept". *Communications of the ACM*, vol. 17, no. 10, October 1974, pp. 549-557

IBM78 IBM. *OS/VS2 TSO Terminal User's Guide*, Manual GC28-0645-4, 1978

IBM79a IBM. *OS/VS2 MVS JCL*, Manual GC28-0692-4, 1979

IBM79b IBM. There is no reference manual for PL/S, for examples see: *OS/VS2 Rel. 3.8 Base Control Program*, microfiche SJD2-6217-00, 1979

IBM81 IBM. *OS/VS2 System Programming Library : Data Management*, Manual GC26-3830-4, October 1981, pp. 147-153

IVERS62 Iverson, K.E. *A Programming Language*, Wiley, New York, 1962.

JONES77 Jones, A.K. "The Narrowing Gap Between Language Systems and Operating Systems". *Proceedings IFIPS 77*, Montreal, Canada, 1977, pp. 869-873

JONES78 Jones, A.K., Liskov, B.H. "A language Extension for Expressing Constraints on Data Access". *Communications of the ACM*, vol. 21, no. 5, May 1978, pp. 358-367

KERNI78 Kernighan, B.W., Ritchie, D.M. *The C Programming Language*, Prentice-Hall, 1978

LAMPS77 Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., Popek, G.L. "Report on the Programming Language Euclid". *SIGPLAN Notices*, vol. 12, no. 2, February 1977, pp. 1-79

LISKO79 Liskov, B.H., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C., Scheifler, R., Snyder, A. *CLU Reference Manual*, Springer-Verlag, New York, 1981

McCAR62 McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I. *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, Massachusetts, 1962

MASHE76 Mashey, J.R. "Using a Command Language as a High-Level Programming Language". *Proceedings 2nd International Conference on Software Engineering*, San Francisco, CA., October 1976, pp.169-176

MITCH70 Mitchell, J.G. "The Design and Construction of Flexible and Efficient Interactive Programming Systems". *Ph.D. Thesis*, University Microfilms, Inc., Ann Arbor, Michigan, 1970

MITCH79 Mitchell, J.G., Maybury, W., Sweet, R. *Mesa Language Manual*, Xerox Palo Alto Research Center, Technical Report CSL-79-3, April 1979

MYERS80 Myers, G.J., Buckingham, B.R.S. "A Hardware Implementation of Capability-Based Addressing". *Operating System Review*, vol. 14, no. 4, October 1980, pp. 13-25

ORGAN72 Organick, E.I. *The Multics System*, The MIT Press, Cambridge, Massachusetts, 1972

RITCH74 Ritchie, D.M., Thompson K. "The UNIX time-sharing system". *Communications of the ACM*, vol. 17, no. 7, July 1974, pp. 365-375

RUDMI82 Rudmik, A., Moore, B.G. "An Efficient Separate Compilation Strategy for Very Large Programs". *SIGPLAN Notices*, vol. 17, no. 6, June 1982, pp. 301-307

SCHWA84 Schwartz, M.D., Delisle, N.M., Begwani, V.S. "Incremental Compilation in Magpie". *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, *SIGPLAN Notices*, vol. 19, no. 6, June 1984, pp. 122-131

SHAW81 Shaw, M. *ALPHARD: Form and Content*, Springer-Verlag, New York, 1981

TEITE75 Teitelman, W., Goodwin, J.W., Hartley, A.K., Lewis, D.C., Vittal, J.J., Yonke, M.D., Bobrow, D.G., Kaplan, R.M., Masinter, L.M., Sheil, B.A. *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978

TEITE84 Teitelman, W. "A Tour Through Cedar". *IEEE SOFTWARE*, April 1984, pp. 44-73

TICHY82 Tichy, W.F. "Adabase: A Data Base for Ada Programs". *Proceedings of the AdaTEC Conference on Ada*, Arlington, VA., October 1982, pp. 57-65

USDOD80a United States Department of Defense. "Requirements for Ada Programming Support Environments". United States Department of Defense, February 1980

USDOD80b United States Department of Defense. "Reference Manual for the Ada Programming Language". United States Department of Defense, 1980

WEGBR71 Wegbreit, B. "The ECL programming system". *Proceedings of AFIPS 1971 FJCC*, vol. 39, AFIPS Press, Montvale, N.J., pp. 253-262

WEGBR74 Wegbreit, B. "The Treatment of Data Types in EL1". *Communications of the ACM*, vol. 17, no. 5, May 1974, pp. 251-264

WHEEL81 Wheeler, J.G. "Improved Sharing of APL Workspaces and Libraries". *APL Quote Quad*, vol. 12, no. 1, September 1981, pp. 327-334

WIJNG77 van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., Fisker, R.G. *Revised Report on the Algorithmic Language Algol 68*, SIGPLAN Notices, vol. 12, no. 5, May 1977, pp. 1-70

WIRTH77 Wirth, N. "Modula: A Language for Modular Multiprogramming". *Software—Practice and Experience*, vol. 7, January 1977, pp. 3-35

WULF71 Wulf, W.A., Russell, D.B., Habermann, A.N. "Bliss: A Language for Systems Programming". *Communications of the ACM*, vol. 14, no. 12, December 1971, pp. 780-790

INDEX OF DEFINITIONS