ASPECTS OF COMMAND LANGUAGE PORTABILITY INCORPORATING A MACHINE-INDEPENDENT FILESTORE CONCEPT

by

N.S. FITZHUGH, M.Sc., A.F.I.M.A.

A Doctoral Thesis submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of the Loughborough University of Technology.

May 1977.

Supervisor: Dr. I.A. Newman
Department of Computer Studies.

ABSTRACT

A brief summary of job control language development precedes a general discussion of possible improvements in command language practice. The user requirements of a command language are considered with special reference to a machine independent basis. "Primitive" functions are defined from this viewpoint.

To meet the proposed objective of portability it is suggested that an appreciation of the user interaction with the computer operating system is necessary. This provides the definition of the user profile model based on the user requirements of a command language. A second model is then developed to represent the structure of the operating system.

These two models are coupled by an intermediate abstract level which is independent of both the user and operating system, yet allows items of one model to be mapped onto the other model. It is this abstract level that later provides the primitive functions for the portable command language.

It is postulated that the file is a common feature of the user profile, the intermediate abstract level, and the operating system. The meaning and properties of files are expounded as abstractions to provide a clearer understanding of their use.

The principles of a machine independent filestore are developed. It is postulated that such a filestore would be based on an arbitrary collection of physical devices and be linked to an indefinite number of

processors. The concept is, therefore, applicable to any multiprocessor environment, including a network.

The method of obtaining the primitives and their properties is explained. The primitives required for filestore operations are syntactically and semantically defined.

These definitions are shown to be viable by a demonstration system which employed the principles elucidated for both the filestore and the intermediate abstract level.

Finally, suggestions of how the work could be used and indications where extensions would be feasible are made.

Keywords:

Computers

Job Control

Command Language

Machine Independence

Portability.

ACKNOWLEDGEMENTS.

I should like to express my appreciation and thanks to those from whom I have received assistance.

I am particularly grateful to my supervisor, Dr. I.A. Newman, for his interest throughout the period of my research and the astute and thoughtful guidance that he has supplied during the preparation of this thesis. I also wish to thank Professor D.J. Evans who has provided continual encouragement and advice.

I am grateful to Ms. A.J. Cook for reading the original manuscript and her support, Mr. M.T.R. Blackwell for proofreading the typescript; both of whom found much to criticise. Any mistakes which remain are my own.

It is a pleasure to acknowledge Mrs. J. Godber's patience and diligence in typing the thesis.

Finally, I am indebted to Nottingham University Mathematics Department for permitting me to be their guest for two years; Cripps Computing Centre, Nottingham University for the use of computer facilities; and S.R.C. for the award of a maintenance grant.

I hereby declare that the work contained in this thesis is my own unless otherwise stated.

N.S. Fitzhugh.

CONTENTS

CHAPTER I

INTRODUCTION

CHAPTER II

ADVANCES IN COMMAND LANGUAGES: A REVIEW

CHAPTER III

BUILDING CLARITY AND PORTABILITY INTO COMMAND LANGUAGE.

CHAPTER IV

COMMAND LANGUAGE MODELS.

CHAPTER V

THE MACHINE INDEPENDENT FILESTORE.

CHAPTER VI

THE FORMAL DESCRIPTION OF THE FILESTORE SUBSYSTEM

CHAPTER VII

IMPLEMENTATION OF THE FILESTORE SUBSYSTEM

CHAPTER VIII

APPLICATIONS

CHAPTER I

INTRODUCTION.

§1. Introduction.

It is estimated that 1.45 billion dollars are wasted per annum due to command language errors [16]. It is also estimated that in the year 1985 only 2% of the total programmer population will possess computer science degrees [16].

Although many existing command language errors can be attributed to mispunching and miskeying [23], a large proportion must be caused by the user unwittingly misusing the command language. However, currently as operating systems and command languages increase in flexibility and sophistication, the user needs to become increasingly experienced to use them [3]. Therefore, unless the user interface to the computer is significantly improved it can be reasonably assumed that the number of command language errors will increase as the user population becomes less computer orientated. Inevitably, therefore, there is an increasing interest in the previously neglected topic of command languages. This has manifested itself in the proliferation of standardisation groups, working parties and individual research in this area in recent years.

Most of these workers have started from the existing command languages produced by the manufacturers for the mainframe machines since these are invariably used in practice. However, every manufacturer has issued at least one operating system for each range of machine that they produce and every one presents a different command language interface to the user. Furthermore, the command languages are described in voluminous reference documents which are difficult to read, often misleading and sometimes incorrect.

Other literature in this field has, until recently, been scarce. Barron [2] has given an overview of the main manufacturers' operating systems with descriptions

of specific features. Barron and Jackson [5] and Enslow [17] have given historic accounts of the evolution of job control languages, and Cox [12] has compared a representative sample of command languages. A review of existing control languages and possible future developments were the topics discussed at a B.C.S. symposium [52] and a conference specifically on command languages organised by IFIP [53] allowed individual researchers to express their ideas. Several committees are working in the field (CODASYL OSCL Task Group, Dutch Job Control Language Committee and B.C.S. Group 5 Working Party) but these have as yet only produced interim reports.

The evolutionary process that has given rise to the current activity has been taking place over the past fifteen years. It is therefore, instructive to examine briefly the sequence of developments that have produced the existing job control languages.

§2. The Requirement for Operating Systems.

The very early computer systems were no more than the hardware components. Each programmer, by necessity, was a proficient machine operator capable of running and debugging his own programs. As the procedure for using the machine standardised it became feasible to employ a permanent operator. His job was to supervise the machine, look after the peripherals and organise the program runs. Delays in setting up programs for execution were not significant because the machine was not particularly fast. However, as the machines became larger and faster, efficient use of the system became a progressively more important objective for economic and management considerations.

Initially this resulted in the operator and a rudimentary system monitor sharing control of the machine and later to control for most functions being transferred to the system, the operator being reduced to peripheral management and responding to requests from the operating system. Parallel with these developments the number of facilities included in the system was inceasing. These

were intended to save programmer time, make better use of the hardware and produce a more attractive system for the purchaser.

Present day operating systems have grown into complex monoliths of code often costing more to produce than the computer hardware. The system assumes ever more responsibility for the organisation of the installation and provides yet more facilities.

The computer needs an operating system because:

1) a large machine provides more resources than a single user can hope to use,

thus

2) they must be shared amongst several users,
3) multiuser environments have to be controlled to prevent interference between users,

and

4) the computer is so fast and complex that decisions are required to be made more rapidly than the human operator is capable of responding for efficient use to be made of expensive resources.

§3. Job Control Development.

When the operator or programmer was in control of the machine the requirements of a program, or a complete user interaction involving several programs, could be expressed as a verbal or a written sequence of instructions. As operating systems developed it became necessary for the programmer to provide two sets of information before his programs could be executed. The first set was for the operator informing him of the expected resources required enabling him to schedule the run, set up the job - order of the paper tape reels or card decks, find the required magnetic tapes etc. The second set was either a paper tape or card deck of instructions for the operating system specifying the machine resources and software requirements of the job. The information for the system had to be written in a language decipherable

by the system itself. Job control languages were developed to fulfil this function.

§4. The Development of the Computer User.

Although the majority of the population do not become directly involved, the computer indirectly effects a large number of people as it is now normal practice to computerise payrolls, bank statements, electricity bills etc. This implies that not only are more people becoming familiar with the advantages (and disadvantages) associated with computers, but more people are actually accessing computers. Consequently it is of the utmost importance that computing expertise should cease to be the domain of the relatively small number of "professionals". The user profile is now wider than ever before, and this trend of embracing further new groups of user is likely to continue. Currently bank clerks and doctors are just two such groups who are beginning to use computer technology. These users are not trained programmers and the computing they do is incidental to the main theme of their work.

The work represented in this thesis was motivated by the need to help these and other types of user by making the power of the computer more accessible which, in turn, should reduce the number of errors made and increase the effectiveness of the system.

§5. Object of Thesis.

Use of computer systems can be less restricted provided that it is possible to present the same user job at any one of several installations and still get the job processed. This can be achieved by job portability which can take the form of:

1) Transference of machine operating systems. The work involved in re-writing several existing operating systems to permit their use on other host machines renders this approach impractical.
2) Transference of job interfaces. There are alternative approaches which aim to translate either any job control into any other, or into an intermediate form.

It is clear that if a common command language were available in most machines this would be a first step in achieving "usable" systems. A portable command language would provide users with the benefit of:

1) the command language being machine independent relieves them from the tedium of learning and understanding the particular idiosyncrasies of each machine, its operating system and its operational environment,
2) only a single language, or a dialect of this language need be learnt,
and 3) savings in time if several computer systems are used.

This thesis postulates that a significant degree of machine independence can be attained if the user interface (i.e. the command language) is built upon a framework of operations and objects which do not reflect any particular machine characteristics. Equally, this frame-work must be independent of any particular user profile.

The semantics of the primitive functions obtained are expressed in a formal definition because it is necessary to have a precise description. However, the approach adopted is intended to be pragmatic and is considered to present a solution to the problem in a form which can be applied. As Anscombe has remarked "what is important is that we realise what the problem is, and solve that problem as well as we can, instead of inventing a substitute problem that can be solved exactly but is irrelevant" [1].

§6. Framework of Thesis.

Chapter II is a general survey of the current status of command languages and recent research work. In Chapter III this review is discussed and user requirements of a command language are formulated. The concepts of a "user profile" model and an "operating system interface" model are introduced. This leads into Chapter IV in which these two models are developed and combined to give an "abstract machine" model independent of user and system idiosyncrasies.

Chapter V develops the theme that a file is the object which the abstract machine will manipulate and examinesthe concept of a machine independent filestore.

Chapter VI uses the tools developed in the preceding three chapters to define the semantics of abstract machine operations: these are the filestore primitive functions. The filestore directory contents are extended showing that the need for non-filestore primitives largely disappears. The completeness and consistency of the filestore primitives are also demonstrated. In Chapter VII a prototype implementation of the filestore and its primitive operations is described.

Finally Chapter VIII concludes with remarks on the applications of this work, incidental results, and indicates where extensions would be feasible.

CHAPTER II

ADVANCES IN COMMAND LANGUAGES : A REVIEW

§1. Introduction.

This chapter seeks to record the major advances in command language practice and theory. In Chapter III these advances are discussed with special emphasis on their relation to the work contained in this thesis.

There are currently two methods of providing job control. The first of these and the most widely practised. is to provide a separate language which only handles commands. This approach is found in all the main manufacturers' computer systems and most of the independent modifications of existing systems. A more recent approach is the integration of programs and job control into a single language. On the theoretical side techniques have been devised which seek to improve the understanding of languages by formally defining the syntax and semantics. Lastly, committees have been established to obtain general agreement upon the definition of job control operations and to examine the possibility of standardisation.

This review is structured in four sections categorised under the topics identified above.

§2. Separate Languages for Job Control.

2.1. Manufacturers' Command Languages.

The languages chosen for discussion in this section are IBM's OS/360 JCL, ICL's GEORGE III, Burrough's Work Flow Language and ICL's SCL. These not only represent the languages used by most computer users but are also considered to be indicative of the trends in command language practice. This is because they have been produced by three independent manufacturers and from the earliest IBM OS/360 JCL, to the latest SCL, span twelve years of third generation technology.

2.1.1. OS/360 JCL.

General Discussion.

The antiquity of OS/360 JCL has resulted in frequent discussions and descriptions. The following review is largely based on an article by Barron and Jackson [5] augmented by IBM reference manuals (principally [22]) and Nicholls [35].

The antecedents of JCL are IBSYS and Fortran Monitor System; both card based batch systems for the IBM second generation machines.

It appears difficult to justify the word "language" when applied to JCL; its format and structure are comparable to assembly programming code, the syntax is symbolic incorporating commas, brackets, asterisks,

amphasands and stroke as integral components of the language. Even including or omitting a space character can be significant!

JCL has not changed since its inception in 1964, although it has been augmented by Time Sharing Option which is the on-line counterpart of JCL. TSO and JCL are incompatible, which means they cannot be used in parallel for job development, however there has been a proposal that TSO will be usable offline in later versions of VS.

The parameters qualifying a JCL command are either positional, each value appears in a predefined position within the parameter list, or keyword in which case the order is unimportant.

Every job is composed of one or more job steps, each job step is introduced by an EXEC statement which can be labelled permitting inter job step communication. The programs in a job step operate on "data sets" introduced by Data Definition statements. Data sets correspond to physical medium permitting programs to run unchanged regardless of the actual storage medium.

Job Control Routines.

The catalogued procedure of JCL is similar to the macro facility of assembly programming languages. The procedures save the user from the tedium of coding standard functions and permit him to run jobs without having to discover the details of the job control involved.

Normally the user provides parameters for a call on a procedure. Only keyword parameters are used and these can be in any order. Those parameters that are omitted are given default values by the system.

The body of a procedure can be modified by replacing statements as specified by the user in his job control.

Conditional Execution.

Each job step produces a return code in the range 0-4095 after its execution. By convention the lower the value of the code the greater the success of the job step execution.

A maximum of eight tests can be made prior to the execution of any job step which is obeyed only if the tests are satisfied. These tests can only be used on subsequent steps of the job control sequence and are limited to prohibiting one step from execution. There are not any looping or recursive execution facilities in JCL.

Input/Output and Data Handling.

The major new facility offered by JCL was permitting internal names for files to be related to specific external data sets or devices.

A detailed description of all data sets used in the job is necessary indirectly, however, thereafter the set can be referenced by the internal name. This facility allows intermediate results, in the form of data sets, to be passed from one job step to any of the subsequent steps

in the same job without the programmer needing to respecify the details of the file.

Data and programs can be stored on disc for user convenience and libraries, compilers and link/loaders are an integral part of the operating system.

2.1.2. George III.

General Discussion.

GEORGE III has previously been described in reference documentation, for example [37], the article by Barron and Jackson [5] and Newell's presentation [30]. These form the basis for the following review.

Much of the underlying philosophy of GEORGE III can be found in earlier Atlas systems; the filestore is one such example. Unlike JCL, GEORGE III is designed to encompass on-line, interactive, off-line and remote access. Similarly, the language is intended to be suitable for all types of user (although the context of the access can prohibit use of some commands in some circumstances, for example, commands which are reserved for the operators).

The most striking difference between JCL and GEORGE III is the presentation of the language. The number of special symbols is vastly reduced and the job control operations are written as a program-like description.

Job Control Routines

The macros in GEORGE III may be system or user defined. The parameter list in the call of the macro body is similar to program language subroutine calls although brackets are not used to enclose the list. In the macro body the formal parameters are denoted by the symbols %A, %B up to %X. (%Y and %Z are reserved for the user and job identifiers). The formal parameters in the macro body are replaced by the variable values specified in the actual parameter list when the macro is called. Keywords may be used in addition to the more usual positional parameters. The occurence of the formal parameter of the form %(<string>) in the macro body causes the actual parameter list to be searched for the first occurence of the associated string. (It is also possible to search for the n^{th} occurence of the string by using %n(<string>) as the formal parameter). The parameter value from the actual parameters is substituted into the macro body.

Conditional Execution.

During the execution of a program a number of distinct events may occur. Without the GEORGE III operating system messages corresponding to these events would be relayed to the operators' console. The event messages are divided into categories and an area of store associated with each category contains the current event message. Under the GEORGE III regime the messages

are intercepted and used as part of the conditional execution mechanism.

The string which forms part of the conditional is compared with the current event message for the specified event category. If the two strings match then the command associated with the conditional (which may be a forward or backward jump) is obeyed.

Another powerful facility is the WHENEVER command. This permits command syntax errors to be trapped and the action associated with the command may be used to circumvent the error. Alternatively the user can specify that certain actions are to be performed whenever a given runtime event occurs.

Input/Output and Data Handling.

GEORGE III is built upon a central filestore containing both system and user files. Programs merely refer to logical devices and the input/output components of the job control specify files that are to simulate the action of the peripherals. This allows data to be independent of devices (although the file needs to be compatible to the device type, e.g. only a text file may be sent to the line printer). Since space on device media is automatically allocated by the GEORGE III system the user no longer needs to provide explicit addressing information. This naturally imposes an overhead not found in OS/360 JCL. However the additional user and machine time required to get jobs expressed in

OS/360 JCL to compile and run correctly must be considered to offset the seemingly superior efficiency of OS/360 JCL . GEORGE III is obviously more acceptable to the user.

2.1.3. Work Flow Language.

General Discussion.

WFL [11] is a compilable block structured, high level Algol style language which was designed to meet the main objective of improving the efficiency of the machine by reducing the need for operator intervention.

The basic unit of user interaction is defined to be the job, each job consisting of one or more tasks. A task is an item of work and is not necessarily synonymous with the OS/360 JCL job step or the GEORGE III command but is more akin to the catalogued procedure or macro performing actions such as compilation, for example.

The variables in WFL are not of predefined type, the correct type is determined by the context in which a variable is used. The WFL variables can be used in arithmetic and boolean expressions and may be passed between the job control and programs.

Job Control Routines.

A crude subroutine facility is available for commands which are to be repeated or which form a

standard series of commands. Unfortunately it is not possible to pass parameters to or from subroutines which rather detracts from the value of this facility.

Conditional Execution.

The block-structure of WFL allows the Algol 60 "If-than-else" conditional to be used and is similar to the programming language implementation.

The outcome of the execution of any task can be determined by using a task variable, which is uniquely associated with the chosen task, in conjunction with the conditional statement. The variable can be compared with the task attributes "COMPLETED", "ABORTED" etc. to produce a boolean result.

A job can be suspended by the WAIT directive. This may be associated with a task value, or be conditional upon a given event such as the presence of a particular file or an operator message.

A FAULT directive, similar to the GEORGE III WHENEVER command, may be used to check for run-time errors. If a fault occurs, the command associated with the FAULT directive is obeyed, provided the fault is within the scope of the directive.

Input/Output and Data Handling.

WFL has access to a library which is an improvement on the JCL library but is not as general as the GEORGE III filestore. Commands are available to manipulate the

files in the library or use them within a job.

File attributes can be changed by job control commands permitting re-titling etc. The attributes that have not been specifically changed remain unaltered.

Output from a job is obtained by using a file with the attribute KIND set to printer. Input, specifically card input, can be external to the job (c.f. GEORGE III INPUT command) or part of the job itself. The card decks can be read in one of three codes (Burroughs common language, EBCDIC or binary) which appears as one of the file attributes immediately prior to the data forming the card deck.

2.1.4. SCL

General Discussion.

SCL, the job control language for the ICL 2900 series, has been reviewed by Barron [4] and the following comments are a précis of that article.

SCL and GEORGE III show some similarities, the new appears to be a development of the old rather than SCL being an original venture. The language has an Algol 68 appearance and unlike GEORGE III is block-structured, a block corresponds to a task to be performed in the user's job. The resources required to execute any task are allocated by the operating system prior to entry into the corresponding block, on exit from the block the resources are automatically relinquished. Each user's SCL job is given an initial set of resources by the system, and these can be fixed

by the individual installation manager. These resources are automatically returned to the system when the job terminates. Unlike GEORGE III, SCL has true variables, these have type similar to Algol variables and can be used in "rows", to give arrays. With these facilities the user could write simple programs in SCL rather than an accepted programming language. The usual arithmetic operators are available for handling these variables. Powerful string handling facilities are an integral part of the language to facilitate manipulation of file names etc.

Two features that are not in SCL which Barron sees as desirable are compound statements in conditionals and a simple repetition mechanism.

Job Control Routines.

SCL statements are calls on system procedures (c.f. MU5 philosophy) of which there are over two hundred. Most parameters are keyword, the value being equated to the name by an = symbol, the whole parameter list enclosed by brackets. Parameter names not assigned values by the user are given system default values. The user's own procedures can be specified either by predefinition as an independent job for subsequent use or by incorporation of procedure code bodies within the SCL code.

Conditional Execution.

Like Algol 68-R the IF and the FI in SCL act

as a pair of brackets so it is possible to nest conditional tests without ambiguity. A predefined variable of type "string" allows the user to test the result of an SCL action by comparing the current system message, which is stored in this predefined variable, with a particular string in the SCL text. (This is a refinement of the GEORGE III system message mechanism.)

Yet another GEORGE III feature can be found in the WHENEVER command, which has been extended to trap return codes placed in a variable in the outer SCL block independently of any user intervention. Because SCL has the arithmetic capabilities and conditional statements previously only associated with high level programming languages it is possible to incorporate tests within the job control allowing, for example, repetition of a program with several sets of data.

Input/Output and Data Handling.

The filestore concept of GEORGE III also forms an integral part of SCL. As in GEORGE III the name of the file is sufficient information for the system to locate the contents and attributes using a directory. Files are input to the filestore by providing the file contents prefixed by the file name and a user identifier. The other attributes of the file, space, disposition etc., are automatically provided by the system. If the user

wishes, he can preset or change attributes, thus modifying access privileges or requesting a specific storage medium. Files are output using the appropriate SCL procedure with the filename as a parameter.

The SCL file operations are more extensive than those of GEORGE III. Barron gives an example showing a filename as an indexed variable which has a literal string appended to form a composite filename. The contents of this file are then assigned to a further string representing a work file. Two work files are merged to produce an updated file. The total job control necessary is expressed in just a few SCL commands.

2.2. Modified Systems and Languages.

The independent command languages considered are notable for possessing similar origins each having been produced by a university or research centre. Generally, the main objective in each case has been the simplification of an existing manufacturer's system by reducing the number of commands and removal of the idiosyncrasies exhibited by particular machines.

2.2.1. Reduced Control Language.

General Discussion.

The first modified system considered, Reduced Control Language (RCL) [50] is hosted by IBM's OS/360 JCL.

RCL specifically aims to simplify the interface for the non-professional who merely wishes to use the computer to aid his other work. The scheme is intended to allow 95% of the jobs run at the installation in question to have all the job control necessary expressed in RCL.

Like the host language RCL has parameters which are either positional or keyword. OS/360 statements can be embedded in RCL code thus providing access to non-RCL facilities. Two different types of default are available; the first automatically supplies a default value for a parameter if the user has not specified a value in his RCL code, the second uses a single RCL parameter to represent several OS/360 JCL parameters and this, by implication, will provide all the necessary default values. As an example of the second type of RCL, default HY means hyper-density and specifies a seven track magnetic tape, 800 bpi packing density and the appropriate volume number.

RCL has the advantages of:

1) reducing the physical preparation because fewer job control statements are required,

2) simplifying the average user's job control by a default system,

and 3) minimising the user re-learning by making RCL similar to the command language previously used by this particular installation.

A disadvantage is that the OS/360 operating system messages are not decompiled. However, it appears that RCL has successfully achieved its objectives of simplifying

job control and easing transition to the new computer.

As RCL is a simplified version of an existing host system the comments on job control routines, conditional execution and input/output and data handling in the review of OS/360 JCL also apply to RCL.

2.2.2. MAXIMOP and CAFE

General Discussion.

The second development considered is the Queen Mary College twin system MAXIMOP and CAFE [51] based partly on existing ICL software. The philosophy of this project has been to provide easy access to the computer for the unsophisticated user by releasing him from almost all job control yet, at the same time, allowing the experienced user full access to the available facilities. This dichotomy in user profile has resulted in a system of two distinct components.

The on-line, interactive component, MAXIMOP, is primarily intended for the experienced user. The system is a development of MINIMOP which is the standard on-line package marketed by ICL for the smaller 1900 computers. Both systems run under the control of the ICL Executive program. The user is intentionally presented with a system which is similar to MINIMOP enabling existing users to transfer to the new system easily. The MAXIMOP parentage is apparent in terms of language, program environment, filestore access and command format. Macro facilities and text substitution

for parameters (as in GEORGE III) are two additional features provided in MAXIMOP which are not in MINIMOP.

The second component CAFE, is a batch system having a very restricted set of facilities. Input and output are limited to card decks and line printer listings respectively. The job control statements form part of the job header card which also acts as a job separator. This safeguards subsequent jobs in the batch against the effects of an omitted terminator. Job time and storage requirements are automatically allocated but may be respecified by additional control statements provided by the user.

Job Control Routines.

CAFE has no subroutine facility. The MAXIMOP macro facility is similar to the macro in an assembly programming language. Parameter values are placed in the macro body by text substitution except when a special symbol has been typed, in which case a system default value is used for the parameter. Macros may use other macros, there being no restriction on the depth of nesting other than the practical one of space. Each macro is expanded into its constituent MAXIMOP commands prior to execution enabling user errors to be found and increase the speed of execution. Users may create their own macros which are similar to the system macros.

Input/Output and Data Handling.

Specific MAXIMOP commands (INPUT, PUNCH, READ, LIST) input and output files between the filestore and standard devices.

Files are either serial containing program texts, data, etc., or random, containing unformatted data. If the user wishes to retain files for use in subsequent interactions he must store them in "userfiles". These can be stored on-line or alternatively off-lined to disc cartridges if infrequently used. 1900 Executive files, i.e. "exofiles", can be used to hold program libraries, work space data and input/output. These files are accessible to batch programs allowing interchange of work between batch and multiaccess.

2.2.3. UNIQUE.

General Discussion.

UNIQUE [34,35] is an operational system developed at Nottingham University. The objective is to provide an environment where the machine specific functions are not apparent at the job control level yet at the same time satisfies the needs of the majority of users. With UNIQUE the average user can specify his job control in a small number of machine independent statements.

The principal criteria used for the design of the language were:

1) short, meaningful names,

2) simple structure,

3) small number of commands for simple tasks,

4) a full set of sensible defaults,

5) suitability for on-line, off-line and remote interaction,

and 6) a user filestore.

In the design an attempt was made to identify likely user requirements and express them in user parlance. This policy is based on the premise that if there is a user need then most existing systems will provide a corresponding facility in some form. Consequently the language is not restricted to a common subset of all the available control languages.

The UNIQUE commands fall into four categories, system enquiries, program execution, filestore manipulation and interactive computing. An extensive default option allows the experienced user to specify his job control with simple statements yet by changing the default values the experienced user can access the full range of system facilities. The defaults are automatic; if the user does not specify a value for a parameter then the default value is assumed by the system.

The messages of the host operating system are decompiled by the UNIQUE system providing the user with meaningful replies to his commands.

UNIQUE has been demonstrated to fulfil its portability criterion as the system is available on

ICL 1906A, IBM 360/67, CDC 6600 and PDP 11 computers.

UNIQUE does not have a subroutine facility although files consisting of job control in the host system language can be accessed. However, it is not possible to pass parameters between the two levels.

Conditional Execution.

A UNIQUE job is composed of one or more "activities" which in turn contain "phases", each phase performs a complete logical section of its encompassing activity. The philosophy of the UNIQUE phase is similar to that of the block in structured programming languages. Consequently it is only possible to jump to the beginning of a phase, not into the commands within the phase. Control transfers are effected by the ACTION command which operates on the value contained in a flag; the possible actions are to ABORT, CONTINUE or transfer control to a specified LABEL elsewhere in the job control. The flag controlling the ACTION can be set by executing programs.

Repetition of phases is achieved by appending the REPEAT parameter to the appropriate phase command. The phase is repeated with the text strings qualifying the REPEAT parameter substituted in place of each occurence of a special symbol within the phase. The loop occurs once for each text string.

Activities can be sequenced before or after other activities or alternatively after a specified time has elapsed.

Input/Output and Data Handling.

Basic input and output is controlled by commands that transfer files to or from real devices. The devices are identified by standard names preceded by a special symbol. Input and output to programs is achieved by associating files or devices with channel numbers used in the program code. This is similar to the GEORGE III system.

A simple file structure is assumed whereby every file has three attributes, owner, name and type. There are two types of file, text and binary. Text files can only be accessed serially with the possibility of restarting from the beginning. Binary files have an arbitrary format and can be accessed either sequentially or randomly.

2.2.4. ABLE

General Discussion

ABLE, a research language designed and developed at Bristol University, has previously been described by Parsons [39] and reviewed by Rayner [40].

The main objective of the experiment was to produce a high level portable job control language suitable for all types of user and uses.

The language is block structured, a new block denoted by a BEGIN symbol and terminated by an END symbol. The permitted variable types are numeric, boolean, string, list or procedure. These must be declared at the head of a block and are subject to the usual scoping rules although system procedures are built-in and freely available to users. The ABLE commands are calls on procedures which perform standard job control functions for the user. The procedure parameters are usually called by value and generally positional. If a parameter is omitted a default value is substituted.

Syntax errors in the ABLE code are detected by the translator. Code generated by the translator is passed to an interpreter which interfaces to the target system. Messages from the host operating system are not decompiled. Translators have been written to convert ABLE into Multijob for a System 4, GEORGE III for a 1906A and Scope 2 for a CDC 7600.

Job Control Routines.

ABLE is based on an Algol 60 type language so, as expected, procedures can be defined and used in accordance to the usual block structure rules. The commands of ABLE are themselves built-in procedure calls.

Conditional Execution.

The implied sequence of job control execution is sequential but this can be altered using IF-THEN-ELSE conditional clauses and the Algol 68 feature of CASE

statements.

In addition WHILE and FOR loops may be utilised. The RUN command, a built-in procedure for executing programs, returns a termination code to the job control level. This is available to the user to check the execution of the program initiated by the RUN command. It is also possible to execute programs in parallel using this procedure.

Input/Output and Data Handling.

Input and output is achieved by using standard procedures which transfer the specified file(s) to or from the device implied by the procedure name. For example, the procedure PRINT implies output to the line printer.

No assumptions are made about the form of a file or the file handling system.

Files are referenced by names which are enclosed by quote symbols, and I/O streams and devices are referenced by a name or an integer. These identifiers are variables and can be used as parametric data for procedures.

2.2.5. General Control Language (GCL).

The final system considered in this section GCL [13,14] is designed primarily to operate in a satellite environment. The GCL code is translated into the target job control within the satellite system prior to its

transmission to the host main frame. The objective of GCL is to remove the idiosyncracies possessed by particular job control languages.

At the GCL level machine independence is achieved by expressing the language operations in terms of the user environment. These operations are mapped onto the target system, which is generally hidden from the user although it is possible to "drop-through" to the target job control language within the GCL code. This facility allows access to features which are part of the target system but are not implemented in GCL.

Portability of the job control expressed in GCL has been achieved by building the system upon a set of primitive functions which form an intermediate level independent of both user and target job control. The set is not closed so new primitives may be added if the designer's objectives cannot be realised by using only the existing set.

At the user level, that is the GCL code, many features of high level languages have been provided. GCL permits variables which can be either integers, strings, lists or primitives. Parameters are either positional or keyword. Positional parameters are mandatory and must be given values by the user whereas keyword parameters are optional and if omitted are given default values by the GCL translator.

Error messages from the target system are not decompiled although this feature is intended to be

to be incorporated at a subsequent stage in the language development.

Job Control Routines.

Job control routines, known in GCL terminology as functions, can be specified by the user. A function is formed by one or more GCL statements enclosed by special symbolic delimitors. The parameters in the function body are represented by fixed identifiers which are replaced by the actual parameters when the function is called.

Conditional Execution.

At the primitive level IF and LOOP functions permit optional execution of statements and repetition of sections of job control respectively. As there is no jump command the GCL statements are executed in sequence although commands may be omitted if a conditional transfers control to a subsequent statement in the job.

Input/Output and Data Handling.

In GCL a single conceptual framework embraces all types of input/output whereby a connection exists between an information source or sink, in GCL termed a device, and program sockets. GCL devices correspond to the physical devices such as card readers or line printers on either the target system, the local satellite system, or files in the target machine filing system. (The target machine file system is conceptually regarded as simulating GCL devices on a small number of direct access devices).

In addition to specifying connections between devices and sockets, other input and output operations such as listings can be achieved.

Physical devices are represented in GCL by suitable identifiers, thus PRINTER refers to the usual line printer. Text and file devices are defined by the user when he invokes a suitable function. Once defined, the device can be assigned to a user chosen identifier for future reference.

§3. Job Control Commands within Programming Languages.

3.1. An Outline for Unification-Wada's Approach.

Wada [47] believes that programming and command languages can be combined into a general multipurpose language. He argues that unification would:

1) improve efficiency by increasing modularisation of system programs,

2) ease language assimulation by the users

and 3) clarify the concepts concerning command languages.

His view is that unification is best approached by incorporating the commands into a programming language. This view is reasoned to be justified because standardised programming languages already exist whereas the user is accustomed to changes in his job control interface.

Wada states that at present languages can only be demonstrated as unifiable after implementation and that he intends to remedy this by producing a scheme to simplify the unification process. This is thought to involve the identification of common features and characteristics which have enabled languages to be unified. As an initial step, existing simularities are identified. As an example, rewinding a magnetic tape is said to be a function common to both program and command regimes.

Wada does not overlook impediments to unification. One of the major problems is the abundance of programming languages. Thus, the unified language could be based on a single programming language helping only the users of this chosen language or alternatively, the commands could be incorporated into several languages resulting in numerous independent unified languages.

A further impediment is created by the association of a command language with a particular computer system. Consequently the commands exhibit machine dependent characteristics. Thus, if a unified language is produced then both programs and job control could be restricted to a single system. At present programs are more than notionally machine independent.

A third difficulty is the inherent difference between programming and command languages; the former is for expressing solutions to problems while the latter is for defining the computer resources required and the control necessary to produce a solution.

Finally Wada sketches an approach for obtaining a unified language. He suggests that the operating system should be controlled by a set of procedures which correspond to job control functions. The algorithmic parts of user jobs are compiled prior to execution. At run-time the job control is handled by an interpreter which decodes the commands into calls on the operating system procedures.

Wada's paper must be regarded as only a scenario for unification as the technique has not been proved in practice.

3.2. Eradication of Command Languages.

A similar approach to that described by Wada has been adopted by Jensen and Lauesen [24] but their system has been implemented.

They propose that command languages are unnecessary as job control can be incorporated into a programming language. They have chosen Algol 60 as the host language and extended it to include control functions. The object of the scheme is to produce a simplified user interface by incorporating both problem solution and job control into a single language. The scheme has been implemented for a batch system.

The commands necessary to control a user job are expressed as an Algol 60 program which may also contain a coded algorithm. In either case the program code contains calls on procedures which interface with the operating system. These job control procedures permit file handling, resource allocation and program execution. The parametric data can be input streams, output streams, file names and program names. Integer variables can also be used to check the result produced by a procedure call. As each job begins execution the system automatically provides a minimal initial set of facilities. These are: a primary input stream, a primary output stream and an initial program for controlling the job stream entering from the primary input source.

Jensen and Lausesen have linked input and output files to the user program by a driver controlled by the operating system. This driver transfers physical blocks of the file contents to and from the program buffers.

It is concluded that the need for a separate command language has been removed by extending the Algol 60 programming language. Users of this language do not have to learn any other language and can utilise the full power of Algol 60 for expressing their job control.

3.3. Job Control on MU5.

Both Morris [29] and Frank [9] have described MU5 job control which they assert has been influenced by the structure of the MU5 operating system.

The operating system is described as a small kernel which performs the tasks of mapping the users virtual machine onto the real machine and driving the input and output devices. Each user job is controlled by its own job supervisor acting as a job initialisation mechanism. The job supervisor's task is to create a virtual machine environment for the user job. Other job control functions termed processes, are accessed through the set of library procedures available to all user virtual machines. Each process created by the job-supervisor is given a priority which depends on the resources the process is expected to consume. The user job is unaffected by errors in processes initiated by other jobs because each job is executed within the isolation of its own virtual machine. This would not necessarily be the case in a system which consisted of a single supervisor controlling all the jobs. Any number of MU5 job supervisors can co-exist without interference.

Inputs to user jobs are passed from the kernel to the job via input device controllers and similarly, job outputs are passed to the kernel through output device controllers. Input and output take the form of "documents" which are similar to GEORGE III files.

The job control functions performed by procedures in the system library are:

1) linking input and output to the job,

2) sequencing subtasks,

and 3) error handling.

Thus, there are procedures for initialising each facility of the virtual machine and performing tasks such as compilations. Errors in job control procedures are indicated by a global "status return" variable whose value corresponds to the result of the procedure. Serious errors cause an interrupt which forces the return into a trap procedure. If the user has not specified a procedure then the system provides its own by default.

Within the context of thisstructure, job control is only required to guide the job through a series of library procedures. Frank considers that a high level programming language is the natural method of fulfilling this function. The procedure parameters are necessarily language dependent and have to interpreted into the logical entities they represent by the individual compilers. The user is able to incorporate his job control into any programming language provided by the MU5 system because the library is available to all the compilers. However, most jobs that are run on the

system require a minimum of job control and for these an independent, simple command language has been provided. This was required because providing access to the full facilities of a programming language for simple jobs proved to be inconvenient for the user and imposed an unnecessary system overhead.

The aspect of job control which cannot be accommodated within the structure of the virtual machine is job scheduling. This is dealt with by an initialisation command which the user must supply at the head of the appropriate input document. The user must provide his identifier, a job name and password and also has the option of specifying a job time limit, size limit and priority.

The MU5 system is intended to be able to accept jobs written in other existing command languages. The systems designers believe that the additional software required to permit this facility would be a supervisor for each command language and the necessary library procedures. The plausibility of this is currently being investigated.

The MU5 system incorporates on-line and off-line usage and has been operational on the Manchester University's Computer Science Department computer complex for some time.

§4. Formal Definition of Command Languages.

The three methods described in this section are considered to be fully representative of the work on the formal definition of command languages. There have been proposals that BNF and axiomatic methods could be extended but it would appear that these techniques are unsuited for this type of application.

4.1. Application of VDL.

The well known formal description technique VDL [26] forms the basis for Niggemann's work on the definition of command languages [36].

Niggemann sees the objectives of a formal definition as:

1) explaining the working of the system,

2) proving the correct working of the system,

3) proving the correctness of fundamental properties,

and 4) analysing the system.

The view is expressed that formal definitions may result in the development of languages based on the problems to be expressed, rather than the operating system. Niggemann also intends that the description should help explain the command language to the user.

The formal description is based on the premise that the system can be defined by the commands, the system tables, the changes in the system tables and the replies to the commands. The effect of a command on a system table can be described by four components: the set of commands applicable to that table, the algorithm for

checking and altering the table, the set of replies to the commands and, the set of issued commands and received replies.

Each job is processed by its own abstract machine operating on the commands entered for the job. Conceptually, a supervisory abstract machine is in overall control of the job processing abstract machines and is not part of the formal definition. This machine monitors system resource requirements of the user jobs.

Each command is processed by its own interpreter. If a command is not recognised by the system then it is invalid. The system replies to the commands are interpreted so that the subsequent action, which is dependent on the reply, can be determined.

These concepts of the system structure provide Niggemann with a basis for command language definition. The formal description consists of a set of algorithms for manipulating the system tables.

Using the VDL notation Niggemann applies his formal method to a hypothetical batch command language. In this context the abstract syntax is given and a diagramatic representation of the structure of the file in the abstract language is shown. This is followed by the definition of a "Rewind" operation for a magnetic tape file.

Niggeman sees the remaining problems as obtaining definitions of the file characteristics and the operations which are valid for a file. He observes that general agreement has yet to be reached on the effect of operations such as opening and closing files.

As the next stage in the development of his method, Niggemann intends to apply VDL to a subset of an existing command language.

4.2. Semantic Description using Predicates.

Weller [49] has devised a method which is specifically for the semantic description of command languages.

A semantic description of command languages is said to be needed for:

1) proving the properties of the language,

2) investigating how the language works,

3) providing a definition for implementation,

and 4) providing reference documentation for the user.

The objectives of Weller's study are:

1) to find the properties required of a semantic description by users,

2) to compare existing methods with these desired properties,

and 3) to develop a new method when the comparison shows existing methods to be unsatisfactory.

Weller limits the range of his study to defining a system just for application programmers. In this context he believes that the command language can be divided into two parts. One part controls the solution to the user problem; the other provides information to the operating system. Weller introduces the concept of a Programmer's Abstract Processor (PAP) which is operated upon by the part of the command language associated

with the solution to the user's problem. Thus, the PAP constitutes the user's interface to the system and the user necessarily understands its operation. Other processors exist but as these do not form part of the problem solving environment the user need not be aware of them. However, Weller concedes that knowledge of other processors may be required when the user misunderstands the operation of the PAP or when he needs functions outside the scope of the PAP. The method presupposes that the user is aware of the effect of his commands on the PAP state as this affects subsequent processing. The processing of a command depends on the values of the objects involved in the intermediate stages. However, Weller believes that the user is only interested in the final result of any command and not in the intermediate processes of the PAP.

Weller proposes a formal description of the PAP consisting of a finite set of truth functional predicates. Each user command is composed from one or more of these predicates. The truth value of the command is found by evaluating the conjunction of its composite predicates. This evaluation process can also be expressed as a decision tree; the route taken is dependent on the initial values of the objects involved. A final assertion can be constructed for any initial set of conditions and consists of logically connected predicates.

Weller shows how commands can be represented by a command table and outlines the PAP output to the user.

4.3. Graphical Representation.

The method described by Bredt [7] is concerned with the processing aspect of command languages and the presentation of the operations involved to the users.

Bredt draws attention to some of the inadequacies of BNF, VDL and axiomatic presentation techniques when used to describe command languages. As an alternative he proposes syntax directed graphs to represent the command processing by the operating system. This technique, he believes, would improve the specification of the semantic operations and provide a method of estimating the response and throughput of a system prior to its implementation.

The graphs define the syntax of the commands. The semantics are in textual form accompanying the graphical representation of the command. The method is demonstrated using a hypothetical computer system.

§5. Command Language Committees.

The professional institutions and computing organisations involved with standardisation have indicated an interest in this topic by independently forming committees to investigate various aspects of job control. The terms of reference and, where possible, indications of preliminary achievements for the three most active committees are reported below.

5.1. CODASYL OSCL Task Group [43].

This group has been meeting since August, 1973; its objective is to investigate the possibilities for, and definitions of, a standard command language.

As a first stage many of the major operating system command languages have been studied and summaries of their similarities and differences produced. The categories of user to be served by the standard language have also been examined.

A model is currently being developed to determine the operating system functions necessary to define a standard command language interface. Four major functional levels are considered. These are Source, Link, Load and Execution. The model is intended to express the relationships between the functions and is seen primarily as a design and teaching aid.

A second model is also being developed and this indicates the hierarchy of execution and control. The structures considered can range from those consisting of a single level to those involving several operating systems

controlled by a "Super Operating System".

From this work the committee is in the process of studying the topics outlined below:

1) A continuing evaluation of other efforts.
2) Expansion of the hierarchic model to cover other operating system functions.
3) Examination of the role of defaults.
4) Refinement of the command language model by:
   a) validation on existing systems,
   b) projection to theoretical models.
5) Development of a model of users.
6) Development of a working command language using either:
   a) existing systems,

or b) a simulated system.

5.2. Dutch Command Language Committee [48].

This committee first met in September 1971, its objective being the development of a language containing the basic job control functions required by the user.

Nine existing systems have been categorised into a function matrix to verify existing ideas concerning the functions currently used. From this initial survey the notion of binding classes has been proposed. Three binding classes are identified, job structure, job resource, and job interface. The job control functions previously identified have been divided into these classes and are being considered by separate groups within the committee.

The structure binding group has produced an

interim report. Firstly, the group defines the terms describing the basic objects such as files. This is followed by a description of the language semantics composed of the following statement types:

1) job start - specifies job parameters and starts job,
2) selection - chooses the next statement of the job to be processed,
3) synchronisation - allows parts of a job to be run before or after other parts,

and 4) assignment - provides a value for a variable.

Work is also proceeding on resource allocation and interface binding but is still at an early stage.

5.3. BCS Group 5 (advanced programming) [27].

The BCS group have assessed other work in the field and are formulating ideas for the definition of a machine independent job control language. The first stage has been the construction of a framework for determing the user requirements. Six types of user have been identified along with five types of usage of the computer system. The user requirements have three aspects: the objects that are manipulated in a particular job control context, the operations that are required and a semantic framework for these objects and operations.

Following from this study, the committee intend to produce language formats which satisfy the user requirements. These are data types, structures, facilities and formal syntax.

Finally, the various requirements are intended to be integrated and a study initiated to try to determine if a single language can be developed satisfying all the specified criteria.

CHAPTER III

BUILDING CLARITY AND PORTABILITY INTO COMMAND LANGUAGES.

§1. Introduction.

Recently, as is apparent from Chapter II, computer command language research has attracted increasing interest. The paramount reason is evident when "In order to understand how to use a powerful, flexible operating system, even to run small simple jobs, one has to be a powerful, flexible programmer" [10] and [3].

Many users limit their job control statements to those previously acquired whilst developing other jobs. New jobs are tailored to suit the existing job control in preference to writing specific job control for each problem.

Users who are slightly more proficient only utilise a well-known subset of the possible commands and still encounter job control errors. Even the "expert" acquires his knowledge over a period of years, and this expertise is liable to become valueless when the computer system is replaced. A further indictment of present day systems is apparent when many potential users are deterred by the involved logistics of accessing the computer.

The inadequacies of existing job control languages have been commented upon by Barron [3] and Shearing [42].

Barron observes that in the utopian situation the operating system is the result of the implementation of the chosen command language. From this notion, it is not unreasonable to believe the command language should be a realisation of how the user wishes to interact with

the computer. It would seem that the current situation could be improved if the user requirements of a command languages formed some part of the design criteria. The user requirements have been expressed independently by a number of authors (Newman [31], Dakin [13], Rayner [40], Shearing [42] and Sibley [43]). The salient points are outlined in §2.

The previous studies which were described in Chapter II are evaluated in §3. This exercise is intended to identify the trends present in command language practice and determine which methods are likely to be of future use. This section also highlights the problems which remain unsolved. The final section discusses one of these outstanding problems; the realisation of a usable portable command interface. Achieving a solution to this problem forms the remainder of this thesis.

§2. User Requirements of a Command Language.

The user needs to communicate with the computer system expressing his requirements and, in turn, the system must inform the user of the events that have occurred relating to his requests. Thus, a dialogue takes place between the user and operating system in which the command language and system messages act as intermediaries.

From the evidence of published work there appears to be general agreement concerning the defficiencies of existing command languages and, in the majority of cases, features that are considered necessary. No one advocates that a command language should be difficult to use, yet few, if any, of the existing languages can be considered to be simple and easily understood. From this observation it can be inferred that while the requirements are readily defined, achieving them is no simple matter. It is also believed that different uses of the system should define different requirements at the interface level although the overall general requirements are the same for each type of user.

As stereotypes the users considered are engineers, technicians, scientists, system programmers and application programmers, these groups containing between them members whose adroitness covers the whole spectrum of computing ability. Other types of user do exist (system managers, operators etc.) whose requirements have not explicitly been considered in this analysis, although of course, many of their requirements will be similar to those of the groups that have been examined.

The main requirements are:

Simplicity

Most users are running simple jobs most of the time, thus it might be assumed that the job control should be simple too. Simplicity in this context implies ease of usage, understanding, and readability. In the past languages have contained a host of special symbols which have no counterpart in natural language and only serve to confuse the user. System messages are often incomprehensible and bear little, or no, relation to the commands issued by the user. Both of these practices seem undesirable, ideally the semantics of both the language and system messages should be tailored for the users.

Extensibility.

If the straightforward job should be simple to run, then it is reasonable to suppose that a slightly more complex job should only be marginally more difficult. Extensibility should allow the full facilities of the operating system, or any subset of them, to be available to the user by building upon his current knowledge. He should not be forced to discard previously acquired knowledge just because he wishes to access a new (to him) facility.

Machine Independence.

It is evident that the user does not wish to be aware of the particular computer used to solve his problem. It is extremely undesirable for him to learn a different command language (and to interpret different system messages) for each computer with which he comes into contact.

These three requirements form the main considerations for structuring the user interface. Two further requirements are necessary for practical application:

1) The command language should not be inherently inefficient. Time spent processing job control commands is time wasted!
2) The system must be able to determine the computer resources required by the user job either through explicit commands forming part of the job control or as implied by the tasks within the job.

These two requirements are discussed in Chapter IV.

§3. Evaluation of Previous Studies.

A discussion of the studies considered in Chapter II appears to divide naturally into the following three subheadings:

1) the practical implementations of a command interface,
2) the formal definition methods,
and 3) the committee approach.

These are discussed below.

3.1. Implementation of a Command Interface.

It would appear that the practical implementations can be judged by how closely they resemble the ideal command language but unfortunately this cannot be defined. However, it is reasonable to suppose that the languages may be compared provided there exists a common standard which can be applied as a benchmark. Consequently it is taken as axiomatic that the user requirements expressed in §2 form part of the evaluation. These requirements have been subdivided into features and other, non-user specific categories added which are considered to define further desirable properties.

The language is considered to be simple to use provided it is easily learnt with a small number of basic commands which have a flexible syntax. The commands should also permit the user to express his requirements in terms that he understands. The messages received by the user should relate to objects which are familiar to him. To permit simple jobs to be expressed without voluminous job control an extensive default system is

obviously advantageous. However, as the user becomes more experienced he should not be restricted by limitations of the language. Consequently the user should be able to employ commands that he has not previously used without discarding those that are already known to him. Similarly the language should incorporate all types of computer usage and ideally the commands should apply to all contexts where this is feasible. Also all types of user should be permitted access to the system although this does not imply that the command interface is necessarily always the same.

By providing a high level interface the users have available powerful facilities which permit him to construct job control using programming language techniques.

An inefficient system would be undesirable whether it be for the user or machine. In the past user efficiency appears to have been sacrificed for the benefit of machine efficiency.

A language which is restricted to a single computer system also imposes a limitation on the user. If the language is machine independent then the user is not forced to know individual machine idiosyncrasies. In addition if the system is portable then jobs can be executed on one of several machines.

A formal definition does not directly aid the user but permits consistency between implementations which means that the user commands always produce the same

result in the same situation.

The comparison of the implemented systems is shown in Table 3.1, each entry has been scored out of a maximum mark of four. The composition of the table will be seen to be similar to one devised by Rayner [40] when comparing UNIQUE, GCL and ABLE. Many of the features in the two tables are the same although Rayner also considers details of language design.

In any exercise of this nature it is inevitable that the scores given are to a degree subjective and although the range of values is small there remains the possibility of disagreement. However, it is not the intention that the scores should be viewed as absolute values but rather as a relative measure for comparison only.

Where more than one language is potentially involved, for example the host programming language on MU5, a compromise score has been used and the symbol "C" has been inserted in the table to denote this.

Category	Feature	OS/360 JCL	GEORGE III	RCL	CAFE/ MAXIMOP	ALGOL60 JCL	UNIQUE*	MU5 JCL	GCL*	ABLE*	WFL	SCL
Simple to use	Easy to learn	0	1	2	2C	1	3	1	3	3	2	2
	Good diagnostics	0	1	0	2C	1	3	1	0	3	2	2
	User orientated	0	2	1	2C	2	4	2	2	4	2	3
	Default system	1	1	2	2C	1	4	1	4	3	1	3
Extensible	No relearning	0	1	2	0	2	4	2C	4	4	1	2
	Complete for users	1	2	1	0	2	4	3C	4	4	2	3
	Complete for uses	0	3	2	0	1	4	3C	4	4	2	3
Language Style	High level	0	2	1	1C	3	4	3C	4	3	2	4
	Procedures/subroutines	1	2	1	1C	3	0	3	4	4	1	4
	Structured	0	0	0	0	3	0	3C	0	3	2	3
	Repeated Execution	0	1	0	0	3	3	3	3	3	0	2
	Conditionals	1	2	1	1C	3	3	3	3	4	3	3
	Variables	0	1	0	0	3	0	3	4	4	1	4
Efficient	For user	0	2	1	1	2	3	2	3	2	2	2
	For machine	3	2	2	2	2	4	3	4	2	2	2
Machine independent language		0	0	0	1	3	3	3	4	4	2	0
System demonstrated as Portable		0	0	0	0	0	3	2	2	2	0	0
Formally Defined	Syntax	1	1	0	0	2	1	2C	0	4	1	1
	Semantics	0	0	0	0	0	0	0	0	0	0	0

TABLE 3.1: Comparison of Practical Implementations

Notes: 1) Low scores indicate poor performance.

2) Ratings for languages marked * taken from Rayner [40], where applicable.

3) C indicates a compromise score.

From the table, which has placed the languages in chronological order, most recent on the right, the following observations can be made:

1) The languages have become more usable. If table 3.1 is examined the two categories, "Simple to Use" and "Extensibility" show a progressive increase in the values given for the more recent languages. The amount of job control necessary for the novice to run a simple job is generally small and easier to understand. The messages generated by the system are not as clear as they might be. The use of defaults permits the user to employ computer facilities without the need to know the details and he can progressively extend his knowledge as and when required because the languages are structured to avoid relearning.

2) The language style is becoming similar to high level programming languages. OS/360 JCL is comparable to a mnemonic assembly programming language, whereas ABLE, WFL and Jensen and Lausen's command language are similar to Algol 60. SCL the most recent language has features previously found only in Algol 68.

3) There is a trend away from machine dependent components in command languages. OS/360 JCL is totally machine dependent and RCL and GEORGE III are very closely linked to their respective machines. The Jensen and

Lausen's proposal and SCL give a semblance of machine independence coupled with a high level interface but portability is by no means illustrated for either. UNIQUE, GCL and ABLE have been specifically designed to be machine independent. GCL and ABLE are translated into the job control for one of the potential host systems whereas the UNIQUE system itself can be ported to other machines.

4) Existing command languages have not been formally defined. Clearly informal definitions of the syntax exist by virtue of the compiler or interpreter validating the command structure. Similarly, the semantics are defined by the code of the appropriate system programs. Formal Definitions are available for some programming languages which have been used to host job control. The additions have been in the form of procedure calls which, while conforming to the syntactic rules of the languages, do not provide the semantics of the commands other than as program algorithms. It would be advantageous to obtain a semantic description independent from any particular programming language. This would provide a universal definition which would not be dependent on limitations of any programming language or the idiosyncrasies of a particular implementation.

From the above it would appear that at present existing command languages are unlikely candidates for

satisfying the user requirements and ad hoc implementations merely simplify the user interface at a particular installation.

3.2. The Formal Definition Methods.

It is convenient, although not essential, to be able to express the definition of a command language in a clear and unambiguous manner so each separate implementation is consistent and the user is able understand the action of each command.

Three definition methods were discussed in Chapter II. VDL may be unambiguous and perhaps clear to the initiated, but the average user can hardly be expected to decipher such a formal definition. The syntax directed graphs avoid these drawbacks but separate the semantics from the syntax which could lead to misinterpretation. Weller's approach using system tables and truth functional descriptions of the commands is both readable and intelligible while at the same time succinctly presenting an abstract semantic description method.

From the methods considered it appears that a major disadvantage of applying a formal definition technique to command languages is that the description is invariably incomplete. BNF, for example, is suitable for defining the syntax of a programming language but the semantics have to be described separately. This is not a significant disadvantage when applied to programming languages as the problem description is essentially self-contained. However, because command languages

interact with the machine operating system the semantic descriptions of the commands must also involve descriptions of the operating system actions required although the syntax of the language is independent of the machine.

Thus, one of the main deficiencies of VDL, BNF and the syntax directed graphs is their concentration on the syntactical definition of the command language. Weller in his method is primarily concerned with the semantic definition but this has yet to be applied to either real or hypothetical operating systems.

There are further disadvantages to the methods proposed by Niggemann and Weller. The objects manipulated by the VDL description are often machine or installation dependent while Weller, who deals specifically with user objects, does not demonstrate that they are transferable to real systems.

However, it appears that the most practical approach for obtaining a formal semantic description of command languages lies in the extension of the method devised by Weller. This view has led to a truth functional representation in tabular form which is the semantic description as described and applied in Chapter VI.

3.3. The Committee Approach.

The committees are partially concerned with the evaluation of existing systems which, while providing a basis for further work is arguably too closely allied with current practice. It also appears that there is a tendency to become involved with the details prior to

the acceptance of overall concepts. However, the committee view, while invariably being a compromise, is more likely to be accepted as a standard than the isolated opinions of an individual. However, the individual proposals can be expected to influence the decisions of the committee.

§4. Re-appraisal of the Problem.

4.1. Discussion.

It is the fundamental proposition of this thesis that a viable realisation of a usable, portable command language is possible. To achieve this objective it is necessary to consider, in parallel:

1) the computer user,

2) the machine operating system,

and 3) the interaction between the user and the system as work is performed.

Different users will make different demands of a command language. Consequently, it would be a mistake to define a user interface consisting of the commands used by the user in the job control. This approach has been attempted by ICL in GEORGE III, for instance, and results in a general command language which, while sufficient for all users, satisfies no particular group. It appears reasonable to suppose that the user interface (that is the commands for expressing the user requests) is to a large extent a function of the usage made of the computer by a particular user. Hence it would seem to be of small value to define a standard command language as this would merely perpetuate the faults inherent in existing languages. A similar view has been expressed by Morris [29] who states "Not only am I against it (standardisation), I am afraid of it. Instinctively I feel that standardisation on any current system would have the same constraining influence on system architecture that Fortran has had on CPU design, and systems are not yet well structured".

Thus it seems most likely that individuals will be best satisfied by using "dialects" of a more general language. Consequently each installation must provide commands which suit their own users. These user orientated commands can be macros or procedures constructed from the basic commands that are intended to form the portable framework. The user command functions will depend upon the individual installations; a card based system would necessarily provide different commands (or the commands would operate in a different context) to a paper tape based system.

If it is conceded that the user commands defy standard definition it seems reasonable to attempt portability at a conceptually lower level than the user.

The lowest possible level is the machine hardware. This is unsatisfactory both in terms of machine independence and usability. Therefore, it is necessary to seek a basis which is conceptually between the machine and the user.

The machine operating system forms a software link to the hardware, and as such is inevitably machine dependent. For practical reasons of time and expense it is inconveivable that the myriad of existing operating systems could be replaced by a single machine independent system which potentially could be interfaced to any machine. Consequently, any portable command language would have to function with existing operating systems. This does not exclude the possibility that the portable basis could be interfaced directly to the machine, in fact, this may even be the ultimate solution for new systems. This

requirement imposes the condition that the portable basis must map onto existing operating systems yet not exhibit machine dependent characteristics. Similarly, it has to map onto the user yet not exhibit user dependent characteristics, that is, it must not be dependent on the particular usage of any single type of user.

These arguments lead to the suposition that the usable, portable basis forms an intermediate level between the user and the operating system. Therefore, the intermediate level can be defined when the user requirements and the functions of the operating system have been analysed.

4.2. Tailoring the User Inferface.

4.2.1. Realisation of Clarity and Portability.

The twin objectives of Clarity and Portability can be achieved by the definition of an intermediate abstract machine conceptually connecting the user to the real machine operating system.

From table 3.1 it is apparent that none of the command languages discussed are clear to use. It is believed that this is caused by the practice of providing a user interface which is machine dependent and designed to function with an existing operating system.

To obtain clarity it seems that the user interface to the machine must suit the individual user. With the present day approach to operating system and command language design this is not a practical proposition. However, if the operating system could be mapped onto an

intermediate level then individual user interfaces which map onto the intermediate level can be readily designed.

Two main methods of providing portability have been proposed. Translation between job control languagues (e.g. Krayl, Unger and Weller [25]) appears to have three difficulties. These are:

1) incompatibiltiy of data structures and programming languages between systems,
2) incomplete mappings between job control languages,

and 3) the difficulty of incorporating new commands. The other main method which has been proposed aims to incorporate job control into programming languages. This approach also has difficulties which are:

1) Interfacing to existing systems requires compilers, loaders etc. to be modified.
2) The command language is not totally user orientated. In MU5, for example, the high level approach has been shown to be deficient in practice since it did not provide a suitable interface for the inexperienced user. As a result an explicit command language is now provided for the MU5 system.
3) New programming languages have to be incorporated into the system.

The intermediate abstract machine provides a means of achieving portability which either circumvents these difficulties or permits a solution to be more readily implemented. The definition of the abstract machine is discussed in Chapter IV.

4.3. Tailoring the System Interface.

4.3.1. Job as an Independent Unit.

The computer performs work presented by the user in the form of jobs. These are composed of job steps which, in turn, consist of programs and control statements. Each job may be considered to be independent from other jobs, interacting only with the operating system. (Jobs which do in practice communicate with other jobs can be considered as part of a "superjob", each superjob being the amalgam of the dependent jobs).

Within this structure the user interaction is a self-contained unit, thus removing the need to make restrictions that would otherwise be required if side effects caused by other jobs were possible. However, the analysis remains applicable to the full class of user jobs considered.

4.3.2. State of the Operating System.

It is reasonable to suppose that since the job interacts with the operating system then changes occur in the system as the job is processed. In fact, both Weller [49] and Niggemann [36] have proposed that the operating system can be modelled by a finite number of states which can be described, in essence, by system tables, the commands operative in the system and system messages. The effect of any command can be defined in terms of the initial state of the system when the command is issued, and the final state of the system after the command has been processed.

All jobs are independent, as defined in §4.3.1. so it is possible to isolate the interactions of each job with the system. Thus, the result of any command can be defined independently of any other jobs forming part of the system when the command is executed. The failure to apply this constraint of formally describing the semantics of commands is a deficiency in current command languages with the result that even the manufacturers cannot be certain what the effect of a particular command should be!

4.3.3. Processing of the Job.

Under the conditions specified in the preceding two subsections, a job is inert while no change of state has occurred. Conversely, if a change is detected some event must have taken place. This implies that the job is dependent on the operating system because any change of state in the job is caused by actions of the operating system. Consequently the job is a series of interactions with the operating system. Each interaction is completed when control returns to the user command stream. Until then the job is unable to proceed any further along the series of interactions (other series are possible in a parallel processing environment but this would be equivalent to a collection of dependent jobs).

In the next chapter these observations are extended and lead to the development of two models. The first represents the user profile based on the user requirements,

the second represents the structure of the operating system. It is from these models that the independent abstract level is produced.

CHAPTER IV

COMMAND LANGUAGE MODELS

§1. Introduction.

In Chapter III the basic ideas underlying the development of the user orientated model and the operating system structure model were introduced.

In this chapter these two models are studied in more detail and used to develop a third. It is this model that is specified to form a common intermediate level conceptually lying between the other two. Later in Chapter VI this third model is used to construct the primitive functions necessary for the machine independent filestore.

§2. The User Orientated Model.

This section is concerned with the development of a model which is intended to reflect the user requirements stated in Chapter III. The conventional command language structure is shown to be unsuitable to meet these requirements and is rejected. As an alternative a hierarchical model is proposed, which takes the user as the pivot around which the remaining components are constructed. The precondition of extensibility is imposed; consequently the default system structure forms an integral part of the model.

The user interaction with the machine is a dialogue. Thus the operating system replies are also part of the user orientated model. Also the model would be incomplete if it did not take account of the relationship between user jobs and their processing by the operating system.

2.1. The Conventional Command Language Structure.

The previous chapter introduced the notion that the discrepancies between satisfying the user requirements and existing command languages can be reconciled by a simple, extensible, machine independent user interface. If existing command languages are examined none satisfy all, and in extreme instances any, of these properties. It would appear that these conditions have seldom, if ever, been considered when any of the present

command languages were originally produced.

When a new computer system has appeared in the past it has seemed that the first component designed has been the hardware. This is driven by the next development, the suite of programs forming the operating system. Once the operating system is established it is a relatively easy task to define a set of commands that correspond to the operations of the system. Further commands can be added for scheduling and control of jobs. The commands are described in reference manuals, the product of technical writers and are generally a voluminous set of tomes. Sometimes a more manageable user guide is also produced.

A conceptual view of existing command languages is shown in figure 4.1. The absence of structure is apparent when the operating system forms the centre of the diagram with the system facilities as the next conceptual level. The user commands are interfaced at this level and the, seemingly, least important component is the user himself. Thus, the system possesses a "bottom-up" structure whereas the main view required by the user is "top-down". Usage of the commands can be made easier if the user's computer centre provides macros, or procedures, for frequently used sets of commands.

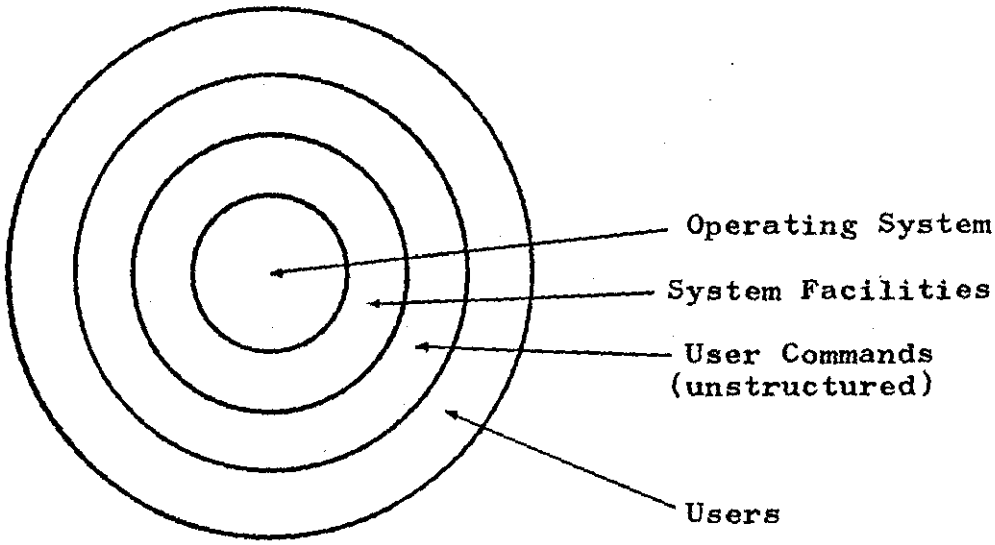


FIGURE 4.1 : Existing Command/Operating System Structure.

This structure is obviously unsatisfactory when the objective of any system is helping users solve problems. If the actual user profile is examined an alternative view becomes apparent.

No matter what the user background, academic, industrial, or commercial, he invariably begins his computing career with what can be termed simple jobs. These consist of card decks (or paper tapes), program or package call, with appropriate data. He becomes aware of the facilities he can use through observing what the computer does to his simple job, conversing with other users, through formal courses, and by the demands of his work. He may require additional facilities to help him solve his problem, in which case

he is motivated (or forced) to seek them out. As a consequence the user may develop his expertise from the simple job through definite phases to become a proficient user. However, not all users by any means attain this pinnacle. In fact, a large number never progress past the simple job stage, and those who do, still frequently run simple jobs.

The user population profile can be built upon this basic premise. Thus as the facilities become more complex and esoteric fewer people try, or need, to make use of them. In fact, it is possible to place the facilities in a loose order with natural extensions - the deeper the level, the fewer the users. This model is not a definitive solution due to continuing developments, but it can reflect the current situation.

From these considerations it is suggested that a hierarchical structure would be the most effective method of satisfying the specified criteria.

2.2. The Hierarchical Command Language Structure.

An alternative view to that of figure 4.1 is the hierarchical structure shown in figure 4.2 which presents a "top down" approach for providing a command language interface to existing computer systems. The approach commences with a small subset of commands at an inner-most (least complex use) level.

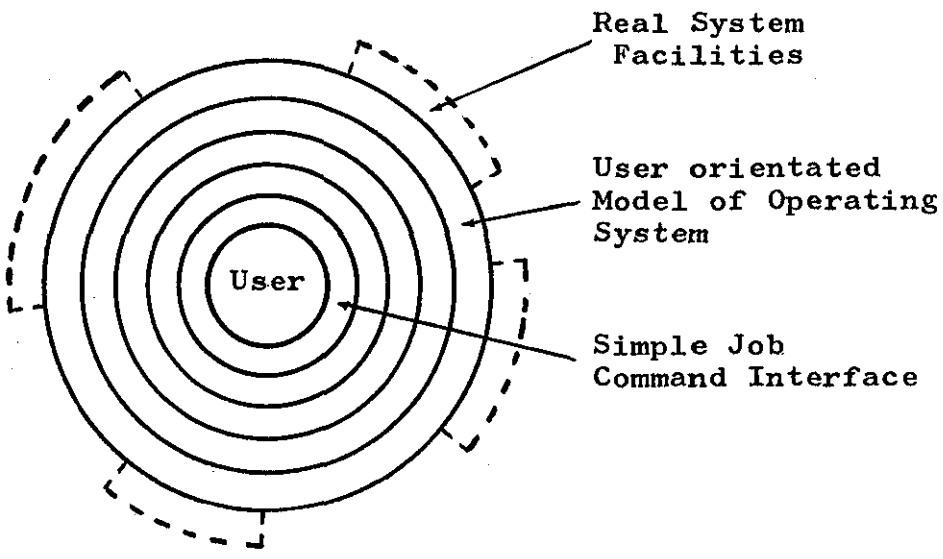


FIGURE 4.2 : The Hierarchical Command Language Model.

The subset is then continously and coherently extended to encompass more complex facilities. Each level in the model encompasses all inner levels (i.e. is a superset of inner levels), conversely each inner level is a true subset of the preceeding level. The outer-most level of the model is thus the complete representation of a user orientated operating system. This must in turn be interfaced to real systems which is indicated in figure 4.2 by an extra broken ring (broken since any real system may have some restrictions in terms of the facilities which can be implemented).

This method satisfactorily solves the problem that no language can ever be designed to encompass all

possible uses now and in the future, however it is the outer-most (most rarely programmed) facilities which normally need to be added to the model, while the subset presented at the inner-most level changes very rarely. Thus it is believed that the user can receive all the benefits of the computer system without needing to know details of the complex facilities.

The model implies a high-level user orientated command language such that a user with a simple job only requires to know a very small subset of the possible commands (and command uses). Both the number of commands and the scope of a given command can be extended as the usage requirements become more complicated.

As computing stands at present it is necessary for the developed command language to be mapped on to existing JCL's. This is not the most convenient method as the commands have to be either interpreted or else translated into the target JCL. Both methods demand the converse activity when decompiling system messages which are in terms of the target JCL. As a consequence the high level command language is probably less efficient than well written host JCL. If well designed, however, the performance of the machine and users should not noticeably be impaired. Because of the transparency of the language, the user should be able to employ his computer time more profitably than before, probably even saving overall machine time.

It is implied in the preceeding paragraphs that the high level approach will incorporate an interface which is minimal for simple tasks but is capable of piecewise expansion to cope with more complex jobs. Current systems do not provide a simple extensible interface although there are no impediments in principle which would render such an interface impossible. The combination of simplicity and power is achieved by using a new conceptual approach to defaults and default handling described in the next sub-section.

2.3. Creating the User Environment.

In conventional systems the user environment is produced from the defaults, and the associated parameter values, specified in the job control file. If a facility is required the user has the option of using the default values provided by the system or he may overwrite some or all, with his own values. The macro and catalogued procedure systems are unstructured so both the simple and the complex jobs use the same procedures. Thus the simple-minded user needs to be aware of the default values that exist for the general and more complex cases. Consequently the user environment is a hotch-potch of macros or catalogue procedures calls with some, or all, of the default parameter values replaced by user specified values.

The hierarchical user orientated model shown in figure 4.2 can be employed to create the user environment by mapping it onto a tree structure as shown in figure 4.3. As the user moves up the structure more system variables which can be manipulated are apparently exposed to him.

He need not be aware of the variables unless the facilities used demand knowledge of this level of the user model. If he employs a facility at level "n" say, then he "sees" only a box at this level. Levels "n+1" etc. will have the variables pre-set according to the current system default values and these will be masked from the user. Should he wish to modify a default value at level "n+1" then he, figuratively, is allowed to look inside the box at level "n" and thus is made aware of the parameters representing the facilty at level "n+1". In fact to be aware of level "n+1", the user must also be aware of a path through the preceeding "n" levels of the structure . As Beech [6] has remarked "If you cannot understand the top level without referring to the lower levels then you have not structured the description properly."

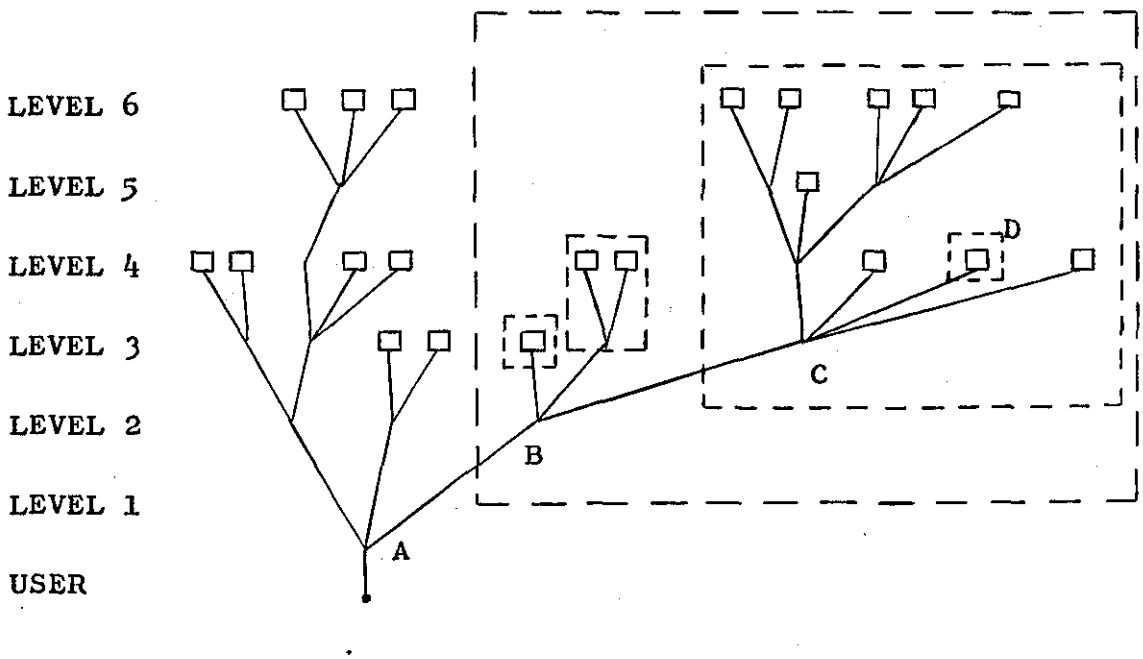


FIGURE 4.3 : The User Environment.

In figure 4.3 the small square boxes represent the terminal leaves of the structure. These can be thought of as being the possible values for variables provided by the system. The user at LEVEL 1 (the node A) is aware of three facilities one of which is indicated by the largest dotted rectangle. If he uses the facility at B then he sees the next level which again consists of three facilities each of which possess default values. At point C he sees three further default values, and so on. (Figure 4.3 should not be taken as implying that any actual model would consist of six levels.)

The establishment of a user environment which can vary with the user and his different uses, is the fundamental prerequisite of an "intelligent" user

orientated command language. Ideally the system could keep a history of the interaction of each user and anticipate his likely needs. More pragmatically a number of likely scenarios can be established, either by the system or by the user, and the user can choose one of these at the start of an interaction. The defaults so established, must be modified in the light of the actual commands issued by the user to supplement and replace the default stream.

So far only the user-system part of the dialogue has been considered. However, it is also necessary to examine the system responses to the user requests.

2.4. The System Interface to the User.

A much neglected area of command language research has been the treatment of "errors" or system messages. The term "system message" is used here to describe any message generated by the system in response to a request. Error messages are a subset of the system messages and in addition to not being readily identified, for example, a failed compilation need not necessarily be an error for some users, require no special analysis as the message merely denotes termination of that particular stage of the dialogue. The user action may depend upon the message, but the analysis is performed by the user and is not part of the operating system. The operating system should be capable of preventing subsequent invalid requests, for example,

an attempt to execute a program which failed to compile, so the results of the previous exchanges in the current dialogue should be available for inspection. Most existing computer systems merely return the messages generated by the operating system directly to the user. Consequently the user is assailed by cryptic and often incomprehensible messages (UNIQUE [34] and ABLE [39] are the only systems known to decompile messages into a form relating to the users commands). It has already been stated that the commands, through default structures, can be simple, extensible, and machine independent. Little will be gained by designing a new user-machine interface if the messages still remain system orientated. If the interface is intended to suit the user, then the messages should exhibit the same characteristics as the commands.

Newman [32] has discussed the feasibility of "user friendly" messages in the database environment. This concept should be equally valid for the command language interface. The class of user that would derive most benefit from such a scheme would be the non-specialist, yet all users should find the system easier to understand.

The system message chosen to be returned to the user is ideally within the scope of his understanding of the machine function. The messages received by the job which only uses simple commands may be different to those received by the job which deals explicitly with sophisticated facilities. Consequently the

messages should correspond to the level of default setting used by the user. In fact, the system messages have a hierarchical structure similar to the user environment, and both are characterised by the default settings. (In an advanced system the user scenario could be used to determine the level of the system messages).

2.5. The User Job and its relation to the User Orientated Model.

Each user job, regardless of its complexity, can be considered as a series of steps, every job step is either a user program or a system function and is regarded by the user as an indivisible unit. The processing of a job step has two phases. There is the upper level consisting of commands, command parameters and their values and system messages. The lower level is formed by the actual evaluation or execution of the job step. At the upper level it is necessary to provide a complete specification of the requirements of the lower level and after the execution has terminated, determine the result of the job step (i.e., compilation failure, file not available etc.). Thus, inputs and outputs exist on two levels. The job step itself has the resources required: hardware-CPU time, main memory, peripheral devices, and software - system utilities, compilers, user programs. These can be implicitly or explicitly defined and to a certain extent are machine independent.

At the lower level the inputs and outputs are generally file contents and interactions with the operating system , both machine dependent. The structure is shown in figure 4.4.

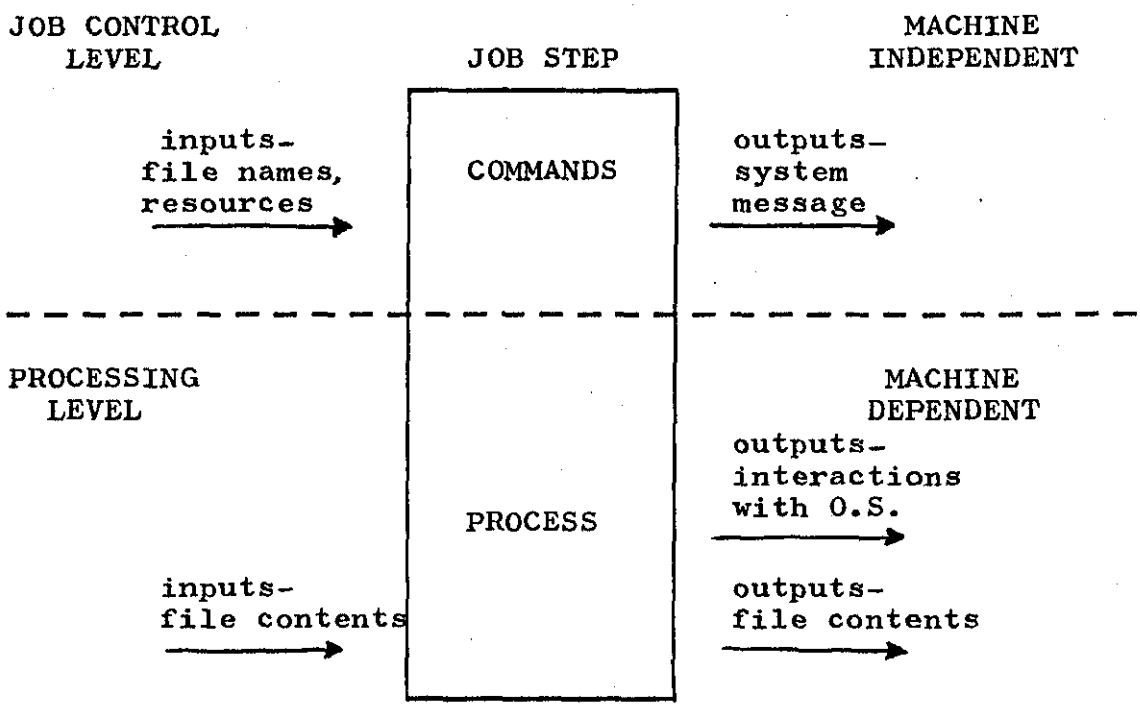


FIGURE 4.4 : Structure of the Job Step.

It is seen from figure 4.4 that the structure begins with machine independent definitions but as the user-orientated model is progressively applied machine specific items soon appear because file contents are dependent upon device and storage media. The inputs and outputs at the processing level are each a series of information transfers, each series can be regarded as a self-contained unit, that is, a compute file.

The properties and use of files are discussed in the next chapter.

An expanded conceptual view of the job step structure can be seen if figure 4.3 is referenced. The user awareness of the true extent of the structure depends on the defaults used. The system completes the structure for the user by combining the user scenario with the existing default set. The defaults are filled in at the job control level prior to execution of the job step; which obviously cannot be started until all the parameter values are known.

§3. Considerations for developing the Operating System Structure Model.

In this section it is shown that the user orientated model alone is insufficient to provide a complete description of the total system. The main deficiencies can be obviated by coercing the operating system into a suitable structure.

3.1. Inadequacies of the User Orientated Model.

The user orientated model has inherent inadequacies which prevents it from being applied directly as a basis for a general command language development. Two problems arise when an attempt is made to devise a structure containing user objects by recursively applying the full "top down" analysis presented in §2. The model produced is found to become machine dependent at different levels depending on the path followed. Furthermore some paths fail to map onto real machine facilities on any machine. Also the user profile chosen at the initial level significantly influences subsequent levels so that the machine hardware required could be dependent on the user. Although this would be possible in principle it seems unlikely to be realised in practice.

The user orientated model cannot be considered a complete representation of a real machine because it makes no provision for the real machine functions, for example, job scheduling is essential for obtaining an acceptable level of resource utilisation. Nor does the model contain any provision for expressing the user

criteria of the job results being returned within the time and cost requested.

These faults are believed to be inevitable if a "top down" only approach is adopted. However, this approach was favoured because the "bottom up" method described in §2.1 has been shown to produce an unstructed machine dependent user interface. It seems, therefore, that neither the existing approach starting with the machine hardware nor the user orientated approach are in themselves the complete answer. Thus some combination appears to be required but before it can be formulated it is necessary to determine the essential components of, and a suitable structure for the operating system.

3.2. The Operating System Structure Model.

The operating system serves two functions. It must satisfy the individual user requirements as expressed in the jobs submitted and maximise the amount of useful work performed by the computer. The second of these functions is outside the scope of this thesis, although any conclusions drawn from this work would be invalidated by gross inefficiency.

The principle task of the operating system, as far as the user is concerned, is the execution of jobs, or more generally superjobs as defined in Chapter III, §4.3.1. At present, it is necessary for the user to be conversant with the methods of expressing the machine resources required by his jobs both for scheduling by the operating

system and for limiting his own use of the system in terms of turnround and cost so as to be commensurate to the resources available to him. Scheduling is independent of job processing but the job requirements must be available to the other components of the operating system prior to execution. Thus, it may be concluded that the user job has two distinct types of interaction with the operating system; the actual processing of the job involving doing "useful" work, and the preparation of the operating system for some part of the processing of the job. Weller [49] draws a distinction between the commands which have a direct bearing on obtaining a solution to the users problem and the commands required by the operating system for planning, administration and accounting etc. which are independent of the problem solution. Weller believes that each command type is processed by the appropriate command interpreter. As an alternative it is proposed that all the commands within a job interact with the operating system and these commands are all part of the user image. Thus, a job can be said to have two contexts: the command context over which the user has control, and the operating system context over which the user has no influence.

Although the operating system in reality may be a monolithic whole, conceptually this need not be the case. Trivially different commands produce different effects on the operating system. (If this were not so then all commands would be identical.) Consequently it

can reasonably be surmised that the operating system can be conceptually viewed as a collection of disjoint modules similar to the actual construction of MU5 [19] and OS6 [44]. It is not unreasonable to suppose that it is possible to separate each module of the operating system into a unit containing pure program code and a table containing all the variables referenced by the coded program. Consequently, the operating system consists of a set of programs and a system table which is the amalgam of the module tables. When a module is used the values of some variables in the system table may change, and this corresponds to a change of state of the system as explained in Chapter III. Thus any change in the operating system is a change in one or more values of the system table relating to the set of system modules.

The structures of the user-system and system-user interactions are shown below.

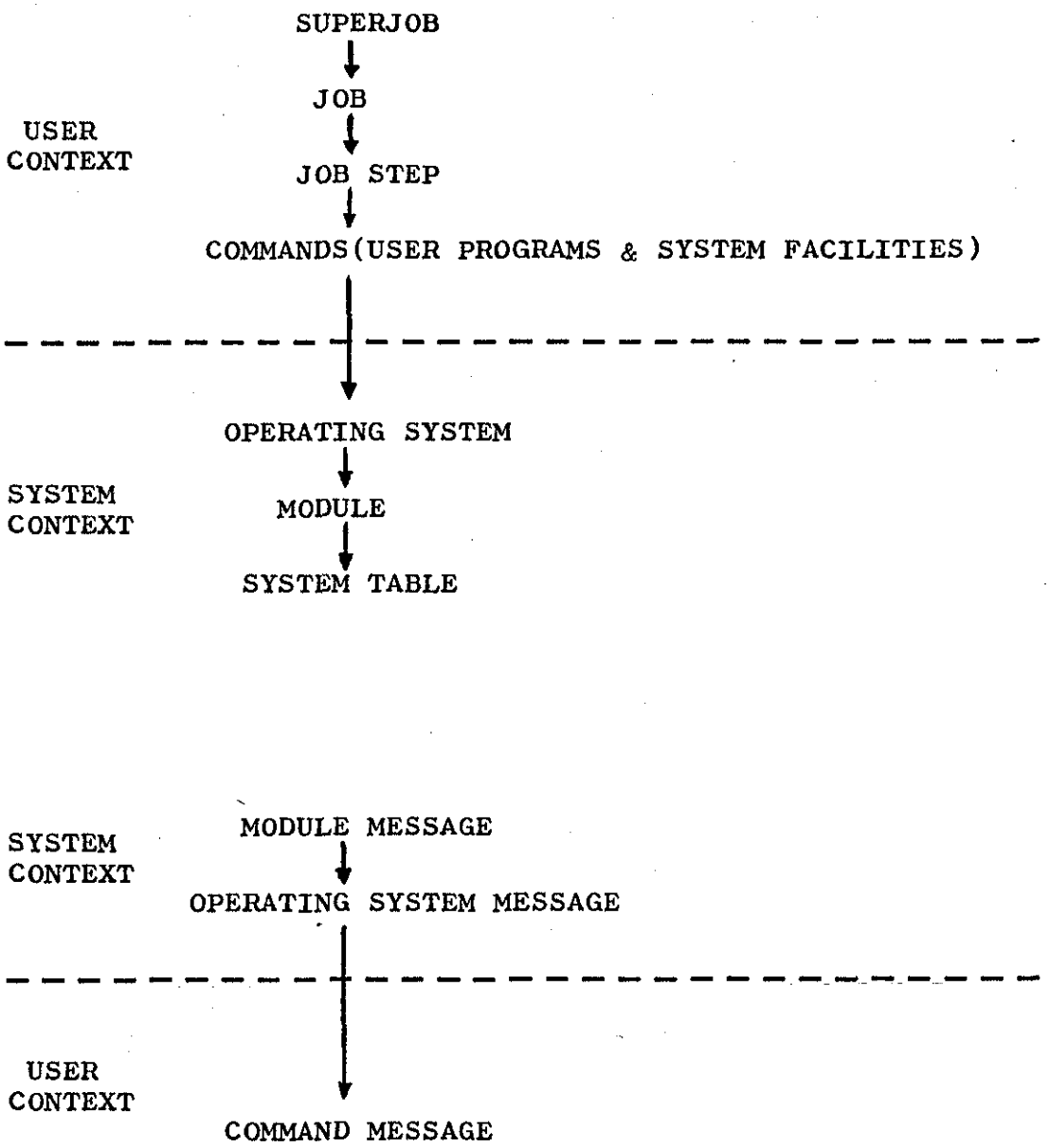


FIGURE 4.5 : User-System Interaction.

In existing systems the module message is often returned directly to the user, bypassing the intermediate levels. By using the structure shown in figure 4.5 it is possible to represent an actual operating system as a collection of modules, a table of values and a table of messages (c.f. Niggemann [36]). In fact, MU5 is arguably a demonstration of the module structure concept as it is an operating system consisting of procedures whose parameters effect changes in the system state, thereby processing work.

§4. The Intermediate Abstract Machine.

The abstract machine is defined to be an intermediate level between the user orientated model and the operating system structure model. The objects in the user orientated model can be mapped on to the abstract machine while the abstract machine possesses a structure and contains objects that correspond to a real machine operating system.

4.1. Integration of the User Orientated and the Operating System Structure Models.

If the user orientated model and the operating system structure model were independent then it would be impossible to link them. However, it is believed that two equivalences exist between the two models.

1) The possible values taken for each user variable at the lowest level of figure 4.3 are equivalent to the entire system table in the operating system structure (the defaults correspond to an additional set of tables)
2) The commands forming the user profile map onto the modules of the operating system. (The mapping functions may be non-trivial.)

These are both reasonable assumptions to make since operating systems do in fact provide a user service however incomplete!

4.2. Composition of the Abstract Machine.

A convenient preliminary view of the abstract machine can be obtained if the elementary ideas concerning finite automata are considered.

A finite automaton is a device with a finite number of inputs and outputs, each of which is capable of two physical states which may be regarded as corresponding to the truth values True and False. The automaton is itself capable of assuming at a given time any one of a finite number of physical states and the state of each output is determined solely by the states of the input, and the internal state of the automaton. (Automata with an external infinite memory are Turing machines[46] but for practical purposes this work restricts itself to finite machines).

The automaton can be defined by a machine table which shows the resultant state arising from each value of the inputs on any current state. This leads to the supposition that the abstract machine is itself defined by:

1) The possible states of the abstract machine.
2) The set of "Activities" acceptable as inputs to the abstract machine for each of the possible states.
3) The current state of the abstract machine.
4) The change in the values of the variables representing the state of the abstract machine caused by each activity on each state.

An activity is defined as an operation on the abstract

machine which potentially causes a state change.

The number of variables representing the state of the machine and the number of activities acceptable as input to the machine must be finite as the automaton is itself finite. The variables forming the state table can be denoted by V_1, V_2, $V_3 \ldots V_n$ $(0 < n < \infty)$, and the activities, denoted by A_1, A_2, $A_3 \ldots A_m$ $(0 < m < \infty)$. It is implied that both the set of variables and the set of activities so defined are distinct. This provides a static description of the abstract machine M viz:

$$M = \{V_i | i = 1,2,\ldots,n\} + \{A_i | i = 1,2,\ldots,m\}.$$

Each variable V_i in the abstract machine can take a finite number of values and these will be denoted by v_{i1}, $v_{i2}, \ldots v_{ik_i}$ $(0 < k_i < \infty)$. Each state of the machine is represented by the variables V_i taking a particular assignment of possible values. The total number of assignments "T", of values to the variables is

$$T = \prod_{i=1}^{n} k_i .$$

This must be regarded as the maximum number of assignments as some assignments may not be possible due to correlation between the variables. However, the number of valid assignments correspond to the

number of states that the abstract machine can attain. These assignments can be ordered and denoted by $S_1, S_2, S_3 \ldots S_t$ $(0 < t \leq T)$.

When a valid initial state is operated upon by an activity, the resulting state must, by definition, be one of the valid states. Thus, under the operators A_i $(i = 1,2,3,\ldots m)$ the set of states is closed. Hence

$$A_i(S_j) \rightarrow S_\ell \qquad i \in \{1,2,3,\ldots m\}$$
$$j, \ell \in \{1,2,3,\ldots t\}$$

The dynamic definition of the abstract machine can be represented by a machine table as follows:

	Initial State			
Activity	S_1	S_2	S_3	S_t
A_1	S_{11}	S_{12}	S_{13}	S_{1t}
A_2	S_{21}	S_{22}	S_{23}	S_{2t}
A_3	S_{31}	S_{32}	S_{33}	S_{3t}
.	.	.	.	.
.	.	.	.	.
A_m	S_{m1}	S_{m2}	S_{m3}	S_{mt}

The final state of the operation of activity A_i on initial state S_j yields the final state S_{ij}.

For convenience the activities have been defined to be parameterless operations. However, the analysis can be generalised to include operations which do have parameters. As there are only a finite number of machine states, there can only be a finite number of values for each parameter of the operation. An ordering can be imposed without loss of generality, such that each set of parameter values corresponds to a particular assignment of values for the operation. If this ordering is mapped onto the activity space there will be a one-to-one correspondence between a subset of the activities and the possible operation calls. This yields a subset of parameterless activities which correspond to the operation.

From the above table it is clear that provided the activities and the states can be defined, the abstract machine will also be defined. However, this would be a non-productive exercise unless it can be shown that the abstract machine forms an intermediate level between the user orientated model and the operating system structure model. The next two subsections show the relationship between the three models.

4.3. Relation of the Abstract Machine to the User and Real Machine.

4.3.1. The Activities and the unit of user interaction.

The only input to the abstract machine is the activity. At least some of the variables in the state table are known to represent user objects and, as such, form part of the user interface. These objects can be operated upon by the activities as they form part of the state table. The user, however, in his interaction with the abstract machine issues requests which are generally part of a conceptually higher level. The highest user level of all is the job, which itself is composed of commands. The commands are the smallest unit of user interaction but are still beyond the scope of the abstract machine.

The commands are assumed to form a stream which is not necessarily serial although the general practice is for each command to be executed in sequence. The commands enter the system through an interpreter or compiler which produces as its output a list of activities. Commands which are syntactically incorrect do not produce any corresponding output although messages from this stage are returned to the user. Each command which is accepted by the preprocessor is divided into a corresponding series of activities, there generally being several activities for each user command. As each activity operates on the abstract machine changes occur in the values of the state table. The changes which take

place are defined by the initial state and the activity. The sequence of activities executed for a given user command will vary depending on the initial state of the abstract machine.

For the user, it is reasonable to suppose that the outcome of a command either produces the result expected or the system is unaltered by the command. However, because the command has been decomposed into its constituent activities, some changes to the system table may have occured before the command "failed". To preserve a consistent view of the system for the user the system must revert to the state which existed prior to the execution of the command.

Commands which do "fail" produce an "error" message at the user interface level. The user who is connected directly to the computer system can react to the message when it occurs (even if this reaction is merely to terminate the session). However, the user in batch mode is unable to adopt this technique and he must anticipate the occurence of errors if he wishes his job to continue processing. Furthermore, the user, whatever his mode of access, needs to be safeguarded against his own reactions to replies which may have been indirectly caused by previous commands. Therefore the system should ensure that messages are consistent with "failures" previously generated within the interaction. This protection cannot be guaranteed by the abstract machine alone as the state table does not contain a

representation of the user interaction. However, if an interface between the user commands and the abstract machine is incorporated then a complete description of the user job can be retained at this level. Thus, the series of events generated by the job as it is processed can form an integral part of this additional level which provides the necessary "memory" of the previous user commands and system messages. The intermediate level is deciduous, its lifetime limited by the duration of each user job.

Because the individual commands become a series of activities which may be bound together as a non-trivial sequence, decisions must be made concerning the subsequent activity required. The processing of commands is controlled by "Activity Handlers".

4.3.2. The Activity Handlers.

It has been shown that the abstract machine has static and dynamic definitions. Equally, any definable user command can be statically described by the subset of activities used to express the command in terms of the abstract machine components, and dynamically by the actual sequence of these activities used to operate on the abstract machine. Invoking an activity does not automatically imply that the system table will be altered because the changes requested may be invalid. Thus the validity of each activity must be determined prior to its operation on the abstract machine. If the activity request is valid the system table is

changed accordingly. However, the activities which form the abstract machine equivalent to a user command do not necessarily operate on the abstract machine in the order in which they are generated by the interpreter. Similarly, some may be omitted. This is because the interpreter has to generate code which can deal with all the possible states of the abstract machine. The sequence of activities actually performed will be dependent upon:

1) The variable values in the system table as this is a complete definition of the abstract machine.

and 2) The previous events which have occured within the current user interaction. These form the job table.

Thus, a logical structure is necessary which is independent of the activities yet determines the sequence required for the evaluation of the user command in any given circumstances. This function is performed by the Activity Handler which:

1) Contains conditionals involving system and job table variables,

2) Contains calls upon activities which operate on the abstract machine,

3) Updates the job table as the user commands are processed,

and 4) Formulates a message for the user.

It seems clear that the activity handler is the ideal level for the generation of user messages. The success of the user command ultimately depends on the state table of the abstract machine. Messages generated directly from the abstract machine would generally be unintelligible to the user, and some messages would be reporting machine faults which are of no interest to the user who merely wishes to know why his job was unsuccessful.

The relation of the activity handler to the user and the abstract machine is shown in figures 4.6 and 4.7.

FIGURE 4.6: Preparation of user request for execution.

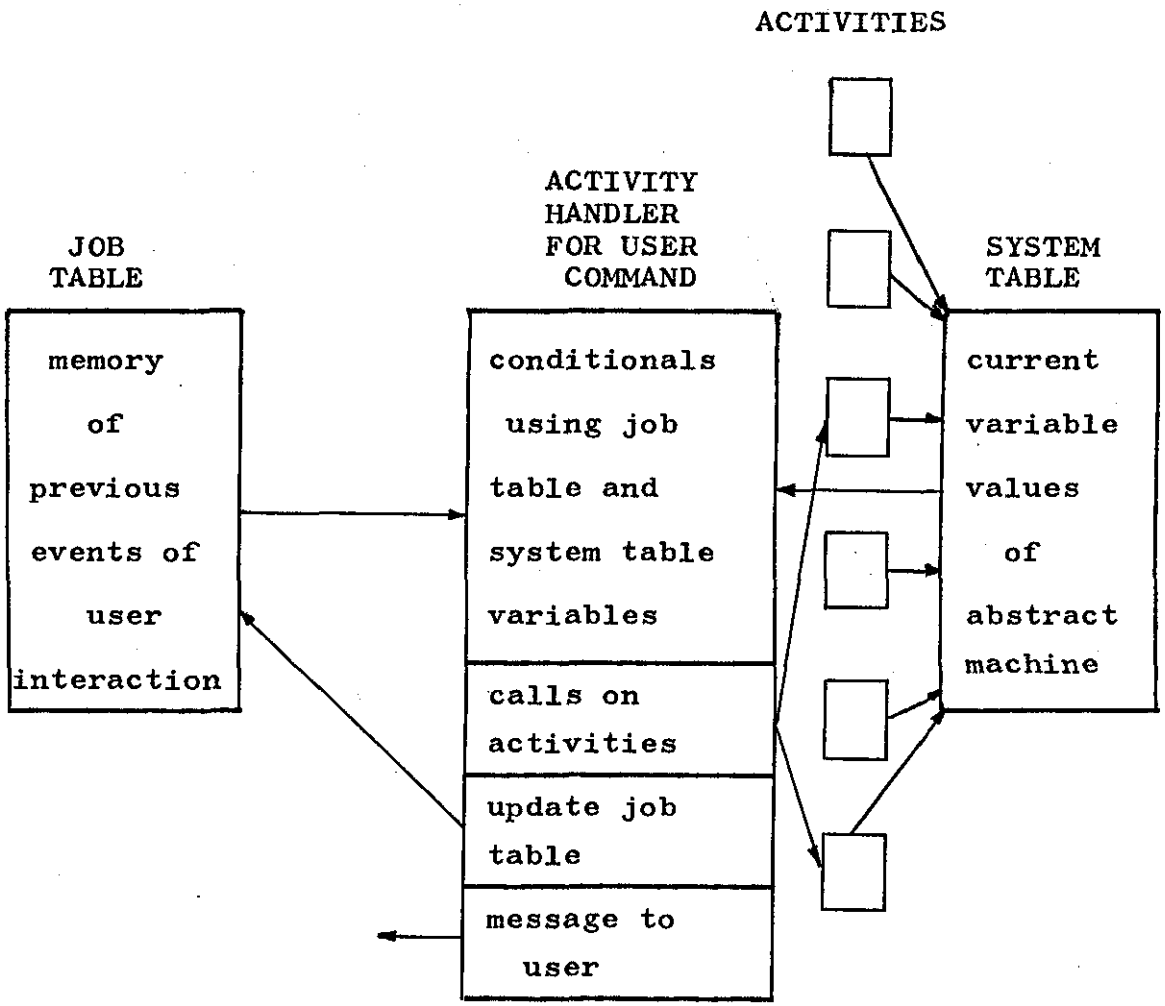


FIGURE 4.7: Execution of User Request.

4.3.3. Practicalities Influencing the System Structure.

To be a realistic model the abstract machine must function in a multiuser environment. The possibility of interaction between several input streams of activities, each stream under the control of independent activity handlers cannot be disregarded. In the generalised abstract machine it is assumed that any finite number of simultaneous activity streams can be processed. However, the activities streams must not be permitted to operate independently, neither must activity processing within one activity handler be affected by the processing of other streams. Activity handlers whose activities use disjoint subsets of the system table can co-exist without interference, however, to prevent more than one activity handler using the same part of the system table a supervisory activity is necessary. This examines the resources required by each activity handler and suspends execution until the system table components needed are free of other activity handlers. This solution implies that the resources required by each activity can be determined prior to execution. This is not unreasonable as the operation of each activity is known and the variables associated with the activity indirectly specify the values in the system table that may be altered. Furthermore, it would be unrealistic to expect the software of the operating system to be fault free or the hardware never to malfunction. In either eventuality an infinite loop could be caused in the execution of an activity. However, it is necessary in any practical realisation of the abstract machine

principles that each activity , regardless of the initial state values, always terminates within a finite time. To ensure this a convenient solution is to structure the system so the activities are subservient to a further, controlling activity which allocates processing time to other activities.

Each activity handler can be given an allocation of time from the controlling activity. If this time is exceeded the activity chain can be aborted.

4.3.4. The Activities and the System Modules.

It will be shown that OS6 and MU5 conform to the modular structure which was suggested would lead to a more usable system. Both systems consist of a set of procedures. Calls on these procedures form the user interface, and the operating system is composed of the procedure bodies. This structure clearly rationalises the design of operating systems, however, it can be shown that this structure also provides the link between the real computer and the abstract machine developed herein.

In OS6 the procedures form the sole level of the operating system, the structure is represented by figure 4.8.

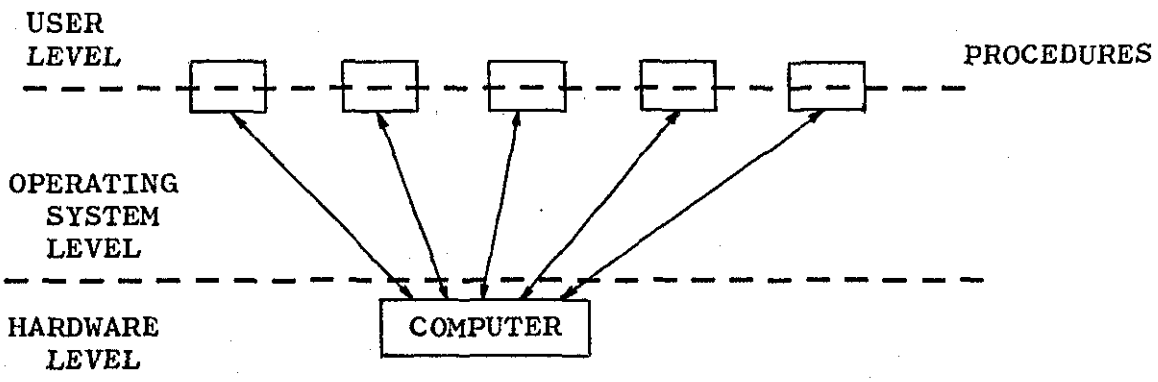


FIGURE 4.8: The Simple Procedural Operating System.

The simplicity of the structure is a consequence of the small number of machine facilities provided and the single user environment. For larger systems, typified by MU5, the structure is somewhat complicated by the need for non-user procedures to control job scheduling, job accounting, prevention of user interaction, data management etc. The more complex structure required is represented by figure 4.9.

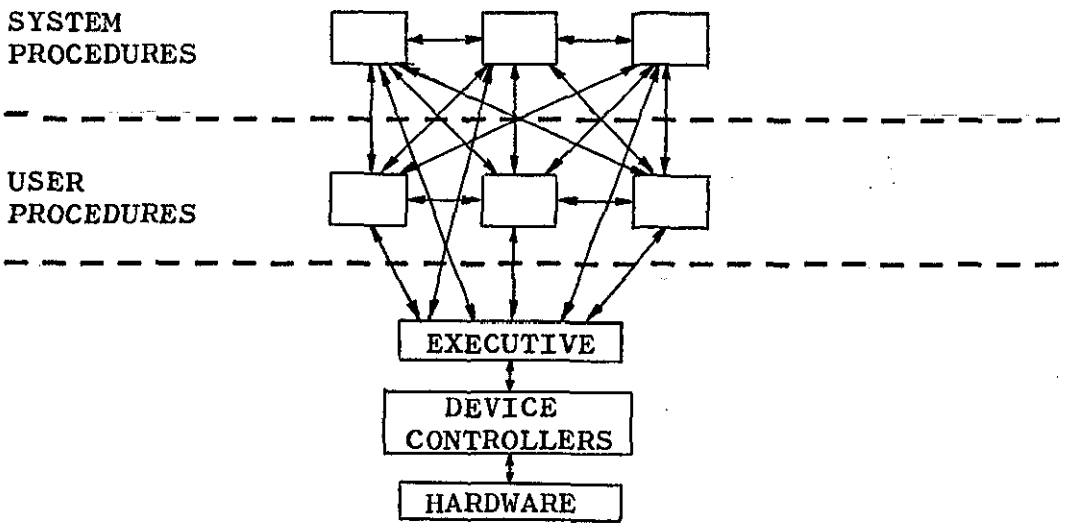


FIGURE 4.9: The Complex Procedural Operating System.

From the diagram it can be seen that each system and user procedure has access to the Executive which controls the hardware devices and provides the link between the processor(s) and the procedures (c.f. Executive of GEORGE III on 1900's and Kernel on MU5). It is also implied that the connection between the system and user procedures is very general, any user procedure being capable of accessing any system procedure. There are two levels of procedure neither of which links to the computer directly as a further level is required - the executive. For the structure of the operating system it is convenient to view the system procedures as being at a conceptually higher level than the user procedures. This is because the system procedures deal with objects encompassing a wider field than the user procedures. Thus the system procedures are concerned with the utilisation of machine resources, interactions between jobs etc.

In both the structures shown in figures 4.8 and 4.9 it is apparent that there exists a parallel between the activities of the abstract machine and the procedure module of the real machine. Each user request involves the use of one or more procedures which are processed under the control of the executive of the real machine. Procedures are similar to the activities of the abstract machine but the activity performs the same change of state, if valid, each time it is executed, whereas the procedure is perturbed by a list of parameter values which affects the execution. However, each procedure will have a finite

number of distinct sets of parameter values which may be used. Hence it is possible to define a one-to-one correspondence between these and a subset of activities of the abstract machine. The controlling activity and the executive both serve the same function in their respective systems. Thus it can be seen that the abstract machine is an intermediate level which can be mapped onto the real machine operating system.

§5. Implementation of the Abstract Machine.

5.1. The Operating System as Independent Sub-systems.

Weller [49] and Niggemann [36] independently advocate that the operating system can be considered as a set of isolatable sub-systems. This proposition forms the basis for the ensuing discussion.

For the purpose of this work a subsystem is defined to be a set of variables taken from the system table with an associated set of activities which operate on these variables.

The possible definable subsystem sets range from the monolithic subsystem which is much in evidence today, to the machine code instructions of the system programs.

It is conjectured that there exist compromises between these two extremes such that the subsystems can be defined to have the following properties:

1) machine independence,
2) each subsystem is orthogonal to every other, no subsystem being a combination of the other subsystems,
3) the totality of the subsystems constitutes a sufficient representation of real machine operating systems,
4) within each subsystem the actions performed are small in number to allow precise semantic definitions,
5) each subsystem reflects some clear, separate portion of the system as seen by the user.

Several sets of subsystems could exist which satisfy these properties and the operations on such subsystem sets would be suitable to form part of the intermediate

abstract machine definition. If the subsystems are truely orthogonal then only the activities which belong to a subsystem will cause a change of state in the subsystem variables. The concatenation of the subsystem tables represents the state table of the total system.

A subsystem can be defined by its corresponding state table. The "value" of a subsystem can be obtained by examining the variable values in the table. A change of state of the machine will only occur when a value is changed in one or more of the subsystem state tables.

5.2. The Subsystems and the Primitive Functions.

The previous subsection advocated the view that the abstract machine could be considered as independent subsystems. The tables corresponding to the subsystems are therefore disjoint containing no common variables but the concatination of these tables is the whole system table.

The activities are known to be operators on the system table of the abstract machine, but as a consequence of the independence of the subsystems the domain and range of each activity will be limited to just one subsystem table. There may be several activities which operate on any one subsystem table.

The foregoing discussion leads to the belief that the activities of the abstract machine form a suitable basis for the definition of the primitive functions

which are to form the intermediate portable level. To be able to define the primitive set it is necessary to identify the subsystems of the abstract machine and the objects contained in the subsystem tables. For each subsystem the subset of primitive functions operating on that subsystem table must be developed such that the conditions of orthogonality and completeness are satisfied. Clearly the division of the total system into smaller units permits this task to be more readily accomplished.

5.3. Selection of the Subsystem for Definition.

It is necessary to consider the subsystems as isolated units in order to obtain a formal definition of the abstract machine which is manageable. One subsystem which appears to have an obvious separate existence is the filestore since the existence and management of files is largely independent of the other activities of job control. It also appears that the file is an entity which is part of both the user image and the system, and consequently its existence in the abstract machine seems very desirable. Also, an examination of the existing job control language operations reveals that most commands involve manipulation of files so that the definition of the filestore would form a significant part of the whole system.

This thesis proceeds by examining the concepts which underlie filestores in order to determine the

composition and the operations required for a notionally machine independent filestore. This yields a definition of the subsystem table which represents the filestore.

CHAPTER V

THE MACHINE INDEPENDENT FILESTORE.

§1. Introduction.

Over the years two groups of computer user have emerged, each group having evolved its own techniques for data handling. This dichotomy owes much to the different working environments of the two groups.

The commercial user deals with data that concerns the day to day organisation of his company. The volume of data is generally large, and production programs require execution at specific times controlled by external constraints,for example weekly payroll program suite. Because of these considerations data and programs were, and often still are, stored on magnetic tapes, each tape file capable of regeneration from a cycle of previous updates. The management of data and programs is necessarily a significant proportion of programming effort and a major concern of every programmer.

Scientists, engineers and non-commercial users have usually worked as individuals, each maintaining his own programs and data files, and being responsible for his own data security.

In either case the user has to:

1) deal directly with the data storage media,
2) have a precise knowledge of the data storage method,

and 3) handle the physical devices directly in his programs.

The third generation of computers, with specifically

large main memory, random access devices and device management through enhanced system software have provided an element of data independence from storage media. In IBM OS/360 JCL [22] the user is expected to know the physical location of his data at the job control level for efficiency (although this is no longer required in his programs) but only limited file management facilities are provided. Alternatively, in GEORGE III [37] the user can be unaware of the media on which his file is stored.

In GEORGE III the space required for a file is automatically allocated on a suitable storage medium and information concerning the file is kept in a file directory. Access to the file contents is obtained merely by using the file name; the filestore system can find the file contents from addressing information stored in the file directory. Incremental dumping of the file contents helps to safeguard against system failures. This also permits the filestore to be larger than the on-line storage capacity as the contents of files which are accessed infrequently can be removed from the on-line media since they will be contained on dump tapes. Previous copies of a file contents can be saved by individual users as a further precautionary measure. However, even with an existing advanced filestore system such as that provided with GEORGE III there remain inconsistencies or areas that require clarification.

The problems that remain are:

1) The plethora of file types, each pertaining to original input/output or storage media.

2) The multiplicity of accessing and addressing methods which are available. These are often confused with the type of file, since available access methods are usually dependent on the storage medium involved. (The two are not synonymous since the most suitable method depends on how the data is used).

3) The definition of meaningful and useful operations which can be performed on files (different filestore systems currently provide very different facilities).

4) The difficulty of achieving filestores which are common to several machines. Hitherto filestores have tended to be limited to a single machine and even then only if it is running a particular operating system. (Often a filestore has been restricted to a specific physical storage medium e.g. disc packs or magnetic tapes).

5) The preservation of the integrity of files without involving unnecessary expenditure of system resources or of programmer time. Automatic preservation is wasteful for those files which are not used again, whereas placing the requirement on the user to copy files for protection explicitly is unsatisfactory from his point of view.

These difficulties arise largely because of the ad hoc development of practical filestores with their attendant requirement of satisfying the needs of programmers who had handled devices and files directly. Existing systems do not seem to be a sound basis for the development of a machine independent filestore. The alternative approach is to obtain a more formal description of the file, the filestore and the program environment. As an initial definition the term "filestore" will be taken as embracing all collections of files. Thus, one type of the more general filestore is the database. This has the additional property of interrelations between the contents of distinct files.

In this chapter an abstract representation of a filestore is developed which enables most of the difficulties outlined to be resolved, while retaining the possibility of practical, efficient implementation.

It is shown that a parallel exists between the mathematical set and the computer file which enables a formal description of the file to be obtained. The constituents of the file are discussed with emphasis on describing the addressing of file contents and the attributes possessed by files. Two further requirements needed for practical realisation of the machine independent filestore, the system integrity and a user structure, are also developed.

§2. The Logical Filespace and its Application.

2.1. Discussion.

The logical filespace is analogous to a mathematical set S which is itself the union of sets $s_i (i = 1,2...n)$. Each set s_i represents a file, the elements of s_i representing the "records" of the file.
Thus $S = \{s_1, s_2, ...s_n\}$. Each set s_i contains elements e_{ij} such that

$$s_i = \{e_{i_1}, e_{i_2}, ...e_{i_{m_i}}\} \qquad (1 \leq i \leq n,\ 0 \leq m_i < \omega)$$

and

$$e_{ij} \in s_i \in S \qquad (0 \leq j \leq m_i,\ 1 \leq i \leq n)$$

The elements of a set that is part of the logical filespace have at least one common property. Trivially the elements belong to the set, by definition. Alternatively complex relations may be necessary to express the properties shared by the elements of a set. A set may also be composed of elements that are themselves sets, or an element may be contained in several sets.

The relations r that are satisfied by the elements of s_i can be used to define the elements giving

$$\{e \in s_i \mid r_1(e),\ r_2(e)...r_k(e)\} \qquad (1 \leq k < \omega)$$

The set s_i can be defined by

$$s_i \equiv \{e|r_1, r_2, \ldots r_k\} \qquad (1 \leq k < \infty) \qquad (1)$$

Standard set notation (see Green [20] for example) can be used to express the interrelationships between sets, which are essentially interrelationships between elements.

Consider, for example, the set S which contains elements which satisfy relations r_1 and r_2, or the relation r_3, but not the relation r_4.

Then

$$s = S'_3 \cap (S_1 \cup S_2)$$

where $s_1 = \{e|r_1, r_2\}$,

$s_2 = \{e|r_3\}$,

and $s_3 = \{e|r_4\}$.

With the operators $\cup$, $\cap$ and complement (denoted by ') all possible relations can be expressed.

It is clear from (1) that if the relations binding the elements of a set are known then a complete definition of the set and the elements can be obtained. The importance of this will be explained in the application of the logical filespace to computer filestores.

2.2. Uniqueness and use of Set Identifiers.

As an abstraction a set consists of the name (set identifier), the list of relations satisfied by the elements, and the elements themselves.

For consistency, each identifier must only relate to

a single list of elements. If two distinct sets of relationships are associated with a single identifier then the elements obtained satisfy:

either 1) the first set of relations only,

or 2) the second set of relations only,

or 3) both sets of relations.

Clearly, if more than one of these interpretations is possible then the elements obtained could differ with each access. Alternatively, if only one of the above interpretations ever occurs then this is equivalent to there being only one set of elements associated with the identifer. Consequently use of this set would be consistent. Thus, the identifier must be unique. The interrelationships between the set elements could be used to identify the set although this would be unwieldy. It is convenient, therefore, to use a name as a notional set identifier. However, once an identifier has been associated with a list of relations the name itself may be considered a complete definition representing the list of relations. The identifier and the list of relations are therefore synonymous and define the elements required.

Thus, the complete definition of a set is

$$\{e|r_1, r_2 \ldots r_m\} \qquad (1 \leq m < \infty).$$

If this is associated with the name s_i then

$$s_i = \{e|r_5, r_2 \ldots r_m\} \qquad (1 \leq m < \infty)$$

and the elements of s_i are such that

$e \in s_i$ if and only if $\{e|r_1, r_2, \ldots r_m\}$ $\quad (1 \leq m < \infty)$

The next subsection shows how the set concepts can be applied to computer files and the significance of the set relations in a computer filestore is explained.

2.3. Relating Computer Files to the Set Concepts.

It is proposed that, similar to the set, the computer file has an identifier, elements or contents, and attributes which correspond to the set relations. Thus, the file will be completely defined by its name which provides a link to the file attributes.

The file identifier must be unique. This can be shown by a similar argument used previously to prove the uniqueness of set identifiers. If the file contents are not changed then every access to the file must always produce the same contents. This will only be possible provided only a single set of contents are associated with each identifier.

In set theory it is possible to take an arbitrary element and determine if it belongs to a particular set by verifying that it satisfies the relations possessed by members of the set. To perform this process in a computer filestore is impractical in terms of efficiency and usability. However, the analysis will be applicable if the relations can be used to identify the elements

more directly. Thus, if some relations are used as generators the contents can be readily determined. The next subsection discusses the composition of the file and indicates the method of accessing the file contents.

2.4. The Conceptual Spaces of the Filestore.

The file may be regarded as consisting of four conceptual components. These are the filename, the file attributes, the logical record identifiers, and the record contents. The relation between these spaces is shown in figure 5.1.

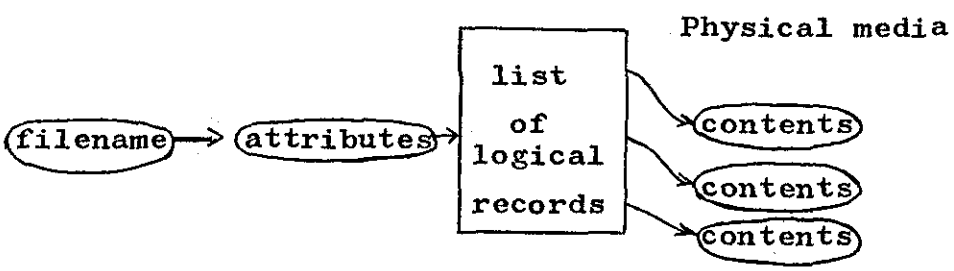


FIGURE 5.1. Relation between the Filespaces.

For each file there exists an ordered set which forms the attributes, some or all of these have values for any given file. If the file is empty then the logical record space is void and no access to the contents space is possible. For the non-empty file the contents are stored on physical media and are addressable through the logical records. In contrast to the set notation, the logical record space concept must impose an ordering on the contents of the file.

The specification of the contents in the attribute list allows the records to be accessed in the implied serial order of the logical records and/or in an indexed order (see §3.3.1.)

2.5. Users and the Logical System.

Users are basically interested in the contents of files and will wish to manipulate the file records. The filestore system on the other hand has no interest in the file contents themselves, it merely stores them as the file is its minimal manipulative entity. Thus the contents are only altered when a user runs a program. In contrast the file attributes are only of interest to the programmer as information - he does not care about the form in which they are stored although he may wish to know their value and occasionally may change some attributes. The filestore system can and must supply the data structure to store the attributes and the code to manipulate them. These details should not be the concern of the user.

The consistency of the system is thus only at risk from the user when he actually runs programs which change file contents. Accessing the attributes is always under the control of the system code.

The concept of an execution environment is introduced to simplify the protection of the system integrity.

2.6. The Execution Environment.

It is the contention of this thesis that an executing program and the files used by the program have an independent existence from the remainder of the filestore system. The advantages of this concept are twofold, firstly it allows programs to be treated the same as other files when they are executing, and secondly, the execution "envelope" thus formed can protect the filestore from programs which fail during execution.

The access to files from an executing program must be consistent to other forms of access, yet it is not directly connected to other system requirements. In existing systems program access to files is through direct access to the file contents allowing the user program to read and write records directly to the files.

As an alternative, the execution envelope can be considered to contain a complete definition of the program requirements for execution independent of the files in the filestore. This permits access to file images from within the envelope through PUT and GET commands issued by the program. If the program fails the original file contents remain unaltered, otherwise the new file contents from the execution envelope are transferred to the filestore by the operating system.

For database applications this approach greatly simplifies the retention of consistency. As the

Database Management System cannot in general recognise a consistent state of the database, it is usual for the application program to indicate the start and end of a consistent sequence of changes. If, for any reason, a sequence of changes cannot be completed the database must be returned to the last consistent state. By executing application programs in a self-contained environment the management required is reduced to copying the new records from the execution envelope to the filestore provided the program is "successful".

It will be seen in Chapter VI that the commands PUT and GET are consistent with the primitive operations designed to operate on file contents.

2.7. Storing and Addressing the File Contents.

The file contents may be stored on any of the computer accessible media; main store, disc, magnetic tape, cards, paper tape etc. The difference between the media lies in the cost of storage and the speed of access to the information in the file. Some of these media, specifically those that are magnetic, permit records to be directly addressable by the system. Conversely, if a file is stored as a card deck, for example, then it is not directly addressable. However, reference can be made to the file contents regardless of the storage media provided the filename and device are known to the filestore. Files which are not directly addressable can be considered as possessing a logical record space which addresses the appropriate device for the media.

In practice this results in a request to the operators to load the file contents into the device. This is comparable to loading an off-line disc pack when a request has been made to access a file whose records are on this particular pack.

Clearly, until a file has been given a name it can have no existence in the filespace. Contents for files have to be explicitly provided from a source e.g. another file, on-line terminal, card reader etc. If the contents are supplied from another file (a copy operation within the filestore) then either the logical record identifiers can be replicated involving no physical transfer of the logical records or the record contents can be copied. Alternatively, if the contents are entered via an external source then the device through which they are supplied must be regarded as a source of new logical records presented in a fixed order. The logical records can then be copied from the input device to one of the filestore media and appropriate logical record identifiers constructed for the file directory. Output of information from the filestore is the reverse process, copies of logical records are transferred to an output device but once dealt with are lost to the filestore system.

§3. The Attributes of a File.

The main difference between the mathematical set used earlier as a description for the logical file and the computer file is that the latter is concerned with objects requiring more than an abstract representation. Consequently it is only to be expected that the file directory contains information which would not normally be associated with the relations defining a mathematical set.

The information contained in the file directory forms the file attributes and can be regarded as falling into two categories. Ideally, in a truly user orientated system all the information required would be for the benefit of the user. However, it must be conceded that some of the file attributes are of little or no interest to the normal user, but are necessary to provide a functional system in an imperfect world.

Whatever attributes are chosen, it is necessary that the value of each attribute in the file directory is one of the permissible values for that attribute. Also, the total set of attribute values forming the file directory must be self-consistent. While these observations do not form preconditions on the choice of attribute types, those that are chosen must each be given a list of acceptable values. Similarly, the combinations of values in a file directory which give rise to inconsistent states must also be specified.

Analysis has shown that the file attributes describe and define:

1) How the file can be, is being and has been, used.

2) The location and construction of the file contents.

and 3) The characteristics possessed by the file.

These are described in subsections 3.1, 3.2 and 3.3 respectively.

3.1. Usage of the File.

In a multiuser environment usually found with filestore systems, the security and integrity of the files must be preserved. However, the protection provided must not be over restrictive or the system may be difficult to use and inefficient. It is essential that each user of the system is positively identified before any interaction can be permitted. Thereafter usage of files is limited to those whose attributes contain the user's identifier for the mode of access requested. Even if the user is permitted to use a file his request must be consistent with other current usage of the file. This problem of file integrity has been discussed by Tozer [45] who also outlines possible solutions.

Having determined which users have access to a file and how users are permitted to access a file concurrently, it still remains to check that the system is performing these requirements correctly. Monitoring of file usage can perform this function.

3.1.1. User Access to File Record Contents.

File record contents can form part of a computer process in three ways:

1) forming the algorithm,

2) constituting input to an algorithm,

and 3) constituting output from an algorithm.

It is a trivial exercise to coerce all computer processes into this basic format. If the records of a file are used as an algorithm or as an input to an algorithm process then it is implied that the original file records in the filestore contents space are unaltered. This philosophy does not prohibit the images of these records undergoing changes as the execution of the process proceeds. (The file images form part of the execution envelope concept).

When a file record constitutes part of the output from a process it is implied that the logical record space of the file is altered. The alterations can take two distinct forms. Either the mapping of the logical records to physical media is changed, or additional logical records (and their corresponding mapping functions) are added to the logical record space. The former of these is comparable to changing the record space, while the latter is comparable to enlarging the record space by concatination.

There is no necessity for updating records in situ. Excluding this simplifies the task of preserving the integrity of the filestore but may have consequences in the design of the space allocation and garbage collection

algorithm in a practical system.

From this discussion four types of access to the file records have been identified:

1) Input,

2) Execution,

3) Output of new record contents to existing records,

and 4) Output of new record contents with new records.

(By isolating the process it will be seen in §3.3.4, which describes the execution envelope, that input and output are different forms of copying the file contents).

3.1.2. Access to Attribute Space.

It is reasonable to suppose that since some of the file attributes are for the benefit of the user, he should have some influence over their values. Typically, he should be able to determine which, if any, of the other users have access to execute his file. However, other attributes, for example, the physical mapping of the logical records, should only be changed by the system when the contents of records are altered.

A fifth type of access is identified.

5) A user may be permitted to change attribute values.

This could conceivably produce a situation not unlike a Gilbert and Sullivan opera [21] whereby an infinite series of lists are required, each list containing the users who are permitted to access the preceding list. For a practical solution it is believed that a single list for each attribute would be sufficient.

3.1.3. Access to Filename Space.

To have existence in the filestore, a file must have an entry in the filename space. It seems unnecessarily severe, in general, to impose restrictions on the entry of filenames other than those of uniqueness and conformity with practical considerations of size. However, it is undesirable that users could, without due regard, remove any filename whether or not it "belongs" to them. Similarly, to have no mechanism for removing unwanted files is equally abhorent. Consequently, users may be permitted to delete an existing file from the filestore.

3.1.4. File History.

The file history, while not essential, is an attribute serving a practical function providing the user and filestore system with diagnostic information. The history is intended to be a full account of the transactions performed on the file namely, the type, time and date of the transaction, with the job or user identifier. For practical purposes the history may be limited to either the most recent or a particular category of transactions.

The historical information retained can provide an insight into the characteristics of file usage which is particularly helpful in a database. The user can interrogate this attribute to check security of his files and to aid the detection of the program responsible for any existing file corruption. Retaining this information may become a statutory requirement if proposed legislation on

privacy of computerised information is ever passed. If the past transactions are linked with the mapping of file records discussed in the next subsection then a complete history of a file, including all previous versions can be maintained.

3.2. Location and Construction of File Contents.

To the user the contents generally form the raison d'être for the file. Independently of the computer system the user must construct the data so it possesses at least a simple structure which will depend on the data and how it is to be used. This process is required even if the user is not intending to store the data in a computer file. Thus, the user imposes a naming convention on the records which constitute his data. Trivially, this implies that records are sequential although complex naming is feasible. Having supplied his data to the computer system in some structure, the user naturally expects to be able to access the records in the same structure subsequently. Thus, when he retrieves a record, the contents obtained should correspond to the record name used.

Consequently the system must preserve the user's naming convention, no matter how it chooses to store the information or how often the data is moved. An attribute of the non-empty file is thus the mapping information which allows records to be located in the filestore when they are named by the user.

3.2.1. Mapping Logical Records to Physical Media.

For a non-empty file the filestore system has to locate the file records by establishing a mapping between the physical storage media and the file records with an ordering conforming to the external data structure required by the user. It is an elementary requirement that the mapping is complete and consistent. Each entry in the contents space must have a corresponding logical record permitting the contents to be accessed. Similarly any single record must address the same physical contents while the contents remain unchanged (system housekeeping may change the mapping but not the correspondence of record to contents).

The amount of information required to form a complete mapping will depend upon the storage media. For discs, the disc, tracksand blocks will have to be identified whereas files in the form of card decks can only be mapped by media description. These external files require human intervention for their use which is obtained by a request passed to the operators from the filestore system. It is the task of the operating system to transfer files from one medium to another through suitable devices. Transfer of cards to discs implies reading via the card reader into a buffer (main store) then transfer to a suitable disc under the control of the operating system. The mapping of the records on disc produced by the operating system is available to the filestore system which contains two files with the same

physical contents but stored on different medium.

The user may legitimately wish to reconstruct the state of his file as it existed at some time in the past. Thus it could be a requirement to record not merely the present mapping between logical records and their physical representation but also the mapping of all previous versions of the record together with their period of existence. This would need to include all records which had ever existed even if there was no corresponding current record. Obviously this facility would be very expensive on storage space if the file contents were volatile and would not be required for most files. However, it is a possible requirement and one that could be implemented.

Similarly, there may be a need to have more than one mapping of the current logical record onto physical storage. This aspect of security is discussed in more detail in section §4.1.

3.3. File Characteristics.

Some of the characteristics required are predetermined by the user in the form of constraints imposed on the file contents, these are structure and storage profile. The system also needs information to:

1) choose appropriate storage media,

2) validate interfile operations,

and 3) complete and validate execution environments.

There is some overlap between the requirements of the user

and system. The characteristics are described below.

3.3.1. File Structure.

Non-empty files have contents which are accessed through the list of logical record mappings. There appears to be two methods of using the list.

1) The records can be accessed in list order, beginning at the first logical record (i.e. the one which provides the link between the attribute space and the record space), continuing one record per access until the last record is obtained. There are variants which permit restarting at the first record or accessing previous records, but basically records are used serially.
2) The records can be accessed through an index which indicates the record required.

3.3.2. Storage Profile.

It must be possible to identify how the file contents are stored in order to validate transfers and operations. Contents can be retained in either internal or external form. Thus, a file can be binary (which is machine dependent) or a common internal text code, or one of the numerous external text codes. Retaining this information prevents binary output to the line printer, execution of text etc.

3.3.3. Record Template.

The size of file records is required for transferring files from the filestore to output devices, chosing suitable storage media and for providing executing programs with a suitable buffering mechanism.

3.3.4. Execution Characteristics.

An attribute previously specified was the list of users who were permitted to "execute" the file. If this list is empty then the file is non-executable by definition (this may be a temporary state) otherwise, the file is executable by the named users. In addition, it is a necessary condition that the file is self-consistent with the requirements for execution.

All executable files, that is programs, need at least one input and one output to be meaningful when executing; the minimal input is the program itself; the minimal form of output is a system message. If a program were permitted to have neither input nor output then its execution would be a null event whose effect on the system would be, by definition, non-existent.

Prior to execution it is necessary to create an environment compatable with the requirements of the program as determined by the programmer when he coded the problem. This environment is defined by the input and output file definitions and the description of the processor requirements. These are not necessarily permanently associated with the file as they may be,

and usually are, changed with each execution. Furthermore, the environment must be fully specified before the program can be executed either explicitly by the user or by system default values. The description of the program/programmer requirements have not been retained in existing systems, but it is the contention of this work that they should be stored. Checking can then be performed at job control load time, not at program execution time.

The description in the directory specifies the requirements for each file that has to be connected. Before execution the operating system must check the completeness and validity of these connections.

A typical execute environment requirement could be as follows:

```
FILE1,    INPUT,     SERIAL,     TEXT
FILE2,    INPUT,     OUTPUT,     INDEXED,    BINARY
FILE3,    OUTPUT,    SERIAL,     TEXT,       LIMIT 2000 LINES
STACK     PROCESSOR,     MAXIMUM TIME  5 MINS
```

Before the execution request can be complied with filenames have to be associated with the file definitions although system defaults may be implied. The processor description defines the processing requirements of the program but need not be an explicit machine name (e.g. BASIC, ICL1900).

If the list of execute users contains more than one entry, for example a compiler is generally accessible to all users, then connecting the filenames must be independent of the directory of the executable file unless an arbitrary constraint is imposed on the number

of concurrent users.

An actual CPU can only execute instructions if they are loaded in a specified set of media which are termed main storage and at present have the property that the information stored in it is transient and only (sensibly) used for executing programs. Main storage is different in this respect from the other filestore media which are used for long term storage of information. Hence, for execution, the executable file must be transferred to the main store associated with the processor selected to execute this particular program. Thus execution implies a copy of the binary file from permanent storage into main storage or virtual main store. (This conceptually occurs only once but may in reality occur many times in a multiprogramming environment with swapping under the control of the operating system. It is only the concept of transferring the file that is important here.) Also, each user who is executing a given file is apparently given his own copy of the file and this is independent from any other copies (again, this is not necessarily true if the code is re-entrant but the concept of independence always holds). Thus each user has his own copy of the file in main store.

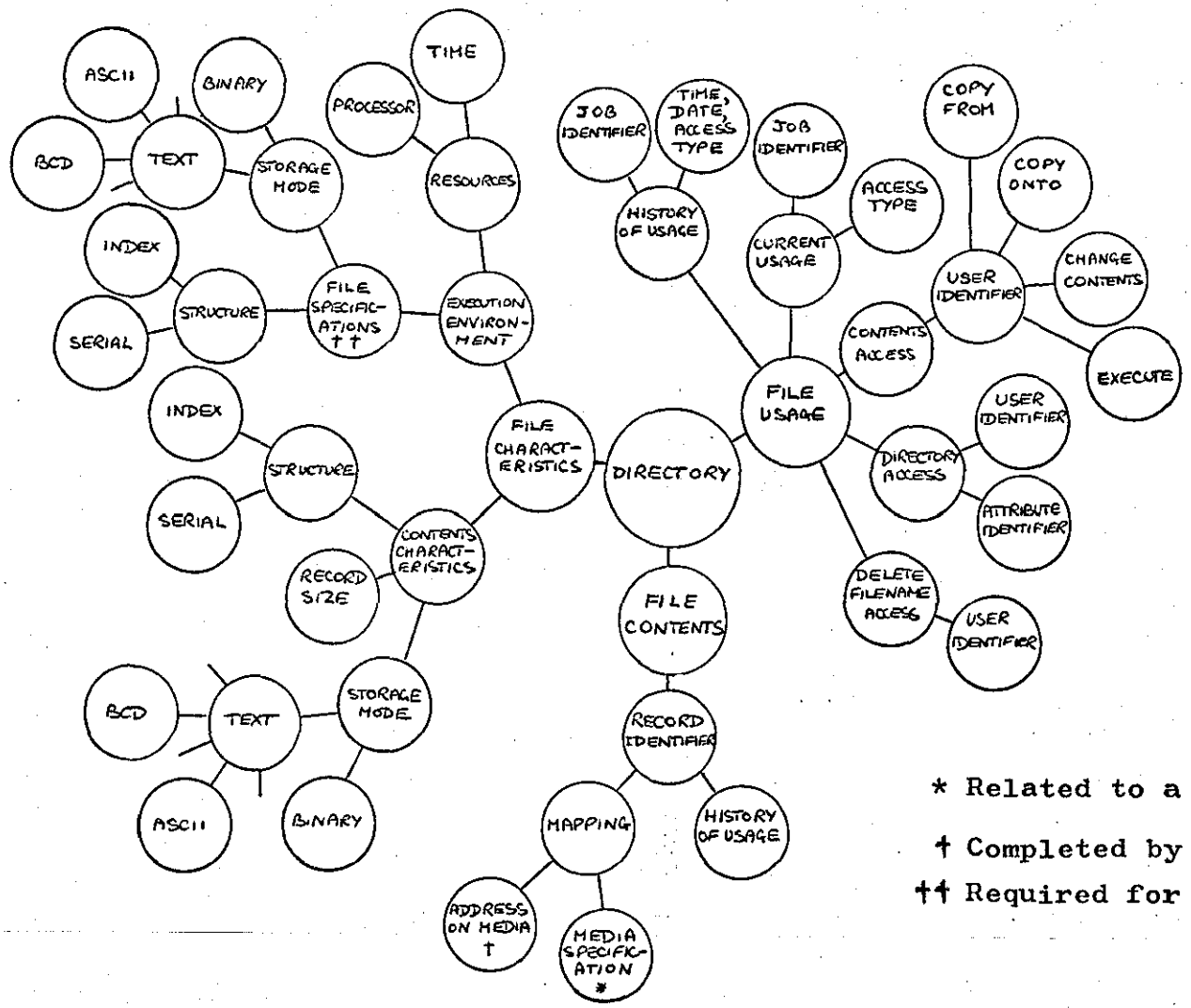
Despite differences it is convenient to regard main storage as part of the filestore and the executing program is a file stored on this medium. Transfer of a file to this medium implies execution. Since the transfer to main storage involves adding extra characteristics

to the original file it is convenient to regard the process of "loading" a file as equivalent to creating a new copy of the file with a new name. Each new copy of the file needs a directory entry to define the file.

The execute request at the user level can take several acceptable forms. It is possible that the user will need to define explicitly the file connections before execution and disconnect when execution is finished. This has the advantage that the system can determine if the file connections are incomplete before execution takes place. Alternatively the user may be given a default system. However, either the user or system must link the files to the program. Thus a user request to execute a file generates a new file directory entry which contains the input/output links and processor characteristics. In this way the filestore concepts allow for executing programs.

There is an interesting and important consequence of dealing with executable programs in this way which is outlined as an application of the machine independent filestore in Chapter VIII.

The above are considered to be a description of the contents of the file directory which constitutes a full definition of the file. The contents and structure of the directory are shown in figure 5.2.



* Related to a complete media specification.

† Completed by operating system.

†† Required for each file used.

FIGURE 5.2. The File Directory Structure and its Contents.

§4. Practicalities of Implementation.

Two topics relating to practical filestores have not been dealt with in the user view of files. These are:

1) System preservation of the integrity of files

and 2) User structure in relation to the filestore structure.

4.1. System Preservation of Integrity.

Some security measures have been previously discussed. The history enables the file contents to be reconstituted while a duplicate of the contents can be accessible from the logical record space. This facility could be used for vital system files and important user files with updates automatically operating on all the copies. If the file is subsequently lost or some records are corrupted the copy can be accessed, without user intervention, to make good the file contents. Similarly, only permitting transactions through the execution envelope greatly enhances the security of the filestore by preventing malfunctioning user programs from changing file records. There are two aspects of integrity which the filestore system should manage, preservation of consistency and prevention of access to corrupt records.

4.1.1. Preservation of Consistency.

When a file is used in a multiuser environment it is vital that interactions between the concurrent users of the file are strictly controlled. The first requirement is that of producing consistent retrieval. On each occasion

that a particular record is examined the contents should be the same. This imposes the restriction that until the user interaction is complete the records of the files retrieved cannot have their contents altered. Clearly this permits any number of concurrent users to retrieve any particular record. When a user interaction involves updating records, an essential condition for consistency is that no one record can be updated concurrently by two or more processes. (For this work it is assumed that the logical record is the smallest unit which the user can specify for access through the filestore system. Records may, and generally are, subdivided into smaller units which can be accessed by user programs and some system facilities such as editing routines.) Consequently, when a record is used in an interaction which has update access, it must be excluded from use by any other interaction.

It appears to be reasonable to expect the resources, in this instance the files (or file records) required for a user interaction to be known prior to the execution of the interaction. If alternatively user interactions were permitted to utilise an unspecified amount of the filestore then it is conceivable that the "deadly embrace" would occur. In these circumstances recovery is possible by abandoning one of the user interactions which automatically releases all the records used by that job.

Whichever techique is employed the management system needs to identify the type of access given to each file and/or record and the user job involved. By coercing this information into the directory space the management is much simplified. If the interaction proceeds in a self-contained envelope (see execution environment) then repeated usage of records does not involve any management and interactions which "fail" automatically leave the filestore unaltered because the envelope is independent of the filestore.

4.1.2. Data Corruption.

It must be accepted that hardware devices malfunction and users inadvertently (or otherwise) destroy file record contents. Corruption occurs at the record contents level produced by erroneous updating (e.g. incorrect format) or by parts of physical store being rendered inoperative. The user is only concerned with the record contents of files so only needs to be aware of which records are affected by hardware faults.

Prevention of user corruption cannot be entirely stopped by the filestore system, although it is to be expected that an attempt to update a text file with binary records, say would be a recognisable abuse.

Records which do fail the system prechecks should not be used (it may be necessary for some users to disregard data corruption in certain instances), and it

is convenient to mark the relevant records as being corrupted. This has the advantages that the user can still access the uncorrupted records and may be able to replace, or reconstruct, the records that are corrupt. The obvious disadvantage is the additional space required to store this information in the file directory space.

4.2. The User Structure.

It is postulated that unlike the principal characters of Orwell's "Animal Farm" [38], users are not all equal. It is desirable, for the benefit of all, that some users will have limited access to filestore facilities.

It appears reasonable to suppose that each user has a set of file attributes that he can influence. Each such attribute has a set of possible values and a default value.

If a user creates a file then he can only set values for those attributes of which he is aware and the values given must be from the subset permitted. If no value is specified the default is provided by the system.

The set of attributes of a user includes a list of the permitted operations. Thus he may only be allowed to create working space for executing programs. Permission to create a permanent file would not automatically imply that the user could remove the file from the system. Hence delete access may be permitted at both the user level and for individual files.

For practical reasons of speed of access and organisation it is envisaged that the file directories will be ordered with a subsection associated with each user. Files can be located by searching the relevant section of the file directory space.

Having specified the file directory and the values that can be taken by the file attributes, a position is reached whereby the operations which can be performed upon the filestore can be defined. As will be seen in the following chapter the validity of a primitive function operating on the filestore is dependent on the values of the attributes in the directories of the files involved. Thus the set of file directories is the subsystem table for the filestore operations.

CHAPTER VI

THE FORMAL DESCRIPTION OF THE FILESTORE SUBSYSTEM.

§1. Introduction.

It has been proposed in earlier chapters that portable command languages may be obtained by defining a set of primitive functions which are orthogonal and complete. The primitive functions form an intermediate, self-contained level between the user and the operating system which:

1) allows any user request to be precisely defined by the semantics of the primitive functions used in the request regardless of the particular implementation,
2) permits the command language to be independent of the host operating system,

and 3) allows the command language built using the primitive functions to be designed to suit the particular user environment required.

The primitive functions are synonymous with the activities of the abstract machine but can be related to objects of the real world. It has been necessary, however, to incorporate parameters with the primitive functions for simplicity and brevity of definition.

As the primitive functions are derived from the activities of the abstract machine they are capable of producing a change of state in the operating system. It will be seen later in this chapter that it would be reasonable to permit the user to issue requests which are not composed from primitive functions, but are

conditionals formed from sub-primitive objects.

This chapter employs the abstract machine concept to produce a formal definition method. This is then applied to the filestore subsystem developed in chapter V. The primitive functions obtained are shown to be independent and complete. It is also shown that the requirement for non-filestore primitive functions largely disappears if the file attribute space is extended.

§2. Application of the Abstract Machine Concept.

The use of the term "primitive function" in command language development is not new. However, as will be seen from the following summary no attempts have been made in the previous work to impose conditions on the primitives chosen nor have they been expressed in a formal structure.

2.1. A Note of Previous Work with Primitives.

GCL developed by Dakin [13] permits access to a variety of operating systems through satellite links to mainframes. GCL is based on a set of primitive functions which can be extended as required.

The language is composed of procedures that form a hierarchical structure based upon the primitive functions at the lowest level. Thus, the user image is represented by a set of GCL function which are at a higher level than the primitives. The primitive functions can be used to determine the type of an identifier, manipulate lists, test conditionals, loop, globally assign values, and return values from subroutines. Rayner [40] comments that the design is "implementation-driven, meaning that good ideas are thought of, implemented, and then rules formulated to cope with side effects with other parts of the language implementation". Two examples of this fault are that global assignment is only global back to the last local assignment within whose scope it is made, and parameterless functions must be provided with an

empty parameter list.

New facilities may require new primitive functions that are inconsistent with the existing set or render some redundant.

It is clear that the primitive functions are chosen arbitarily; no rules concerning completeness, consistency or redundancy are imposed. Thus the GCL primitives do not possess the properties which were specified in Chapter IV as being desirable.

A second system OS6, described by Stoy and Strachey [44], is primarily based on input and output primitive functions. OS6 is restricted to those machines which have an BCPL compiler and only needs to provide a single user environment. The primitives are operators implemented as procedure calls whose parameters can be program names, data stream names or variables and form a user interface to the system. The procedures are often linked to particular devices such as a flexowriter or Olivetti terminal. The primitives Next[S] and Out[S,x] are used to input and output respectively to stream S. Next causes the next character of the stream S to be read while Out transfers the object named by its second parameter to the stream S.

Primitives for error recovery, testing end of a stream and closing streams are also available.

The filing system has primitive functions for creating and deleting files, creating a file stream, transferring blocks of files and indexing blocks of files.

The OS6 primitves show a remarkable similarity to the GCL primitives if the functions alone are considered, although GCL primarily operates on strings whereas OS6 deals with streams.

The comments concerning the GCL primitives and their performance in view of the criteria of Chapter IV also apply to the OS6 primitives. The user interface is the OS6 primitive set and forms part of the BCPL language so that no separate job control is necessary. The primitives are a realisation of the operating system at the BCPL level and, as such, are similar to procedures of MU5.

Newell [30] states that the GEORGE III commands are themselves primitives.

A major design criterion of GEORGE III was to identify each primitive operation and subsequently transform it into a command. Thus, each command is supposed to perform one basic function. However no further conditions appear to have been imposed. Consequently, GEORGE III has a large number of commands which are not orthogonal (e.g. listfile can perform many of the edit functions) or machine independent.

It is apparent from the above that no clear consensus of opinion exists for deciding upon the function of primitives,nor on a method for obtaining them. Of the examples cited, only GCL can be considered as portable, GEORGE III and OS6 both link the user interface directly to the machine operating system. None of the primitive sets have been constructed with

orthogonality as a precondition, and only GCL can be considered as having usability as part of the design criteria.

The next subsection indicates how a set of primitive functions can be obtained satisfying the criteria stated earlier. This is considered to produce primitives which are an improvement on existing sets.

2.2. Application of the Abstract Machine for Primitive Function Definitions.

In Chapter IV the concept of the abstract machine was developed and the conditions that the primitive functions should satisfy have been stated. To continue with the analysis for obtaining primitive function definitions it is necessary to have a framework providing a structural basis to allow the primitives to be linked to a model of the actual computer system. The abstract machine model has been specifically developed for this purpose. The activities of the abstract machine can be regarded as synonymous with the primitive functions but have the advantage of constituting part of a system which has its complete structure defined. This enables the structure of the primitives to be obtained.

Clearly, a specification of the primitives alone would be incomplete, for it is also necessary to define the subsystem tables and the changes of value that may occur in the table. A primitive will only change the

state of the machine if the action is valid. Consequently, not only must the action of each primitive be defined, but also the conditions which must be satisfied before any change can occur. This latter (passive) constituent of the primitives is discussed in the next section as is the semantic definition of the primitive functions.

§3. The Formal Definition Method.

3.1. The Primitive Substructure.

For the ensuing analysis it is assumed that hardware and software malfunctions are dealt with independently of the execution of the primitive functions. If a user request can be compiled or interpreted, then the request is valid and can be executed by the system. However, the result of the execution may not meet the user's expectation!

It is a basic requirement that each user request can be expressed in terms of primitive functions which are bound together by handlers as described in Chapter IV. If the user request is obeyed then a change in the system table occurs corresponding to some combination of the primitive actions comprising the user request. Each primitive function which is obeyed has the potential to alter the system table but these changes do not occur automatically whenever a primitive is obeyed. The system table will only be altered when the primitive action is found consistent with the current state. Consequently it is necessary to define the circumstances under which changes will occur. This can be achieved by defining a set of prechecks associated with each primitive which operate on the system table. It is the result of these checks which determines whether the primitive action is to be performed. The prechecks are the substructure of the primitive and are:

1) finite in number,

2) definable by the action implied by the primitive,

and 3) dependent on the initial state of the system.

Thus given any initial state of the system the particular set of checks necessary to determine the consistency of the action to be performed by a primitive can be specified. All such sets of prechecks can be combined to form a flow diagram consisting of logical tests yielding the truth value True if the primitive action can be performed, otherwise False.

This division of the primitive into a component for checking the primitive action and a component for performing the change to the system table permits the definition to be in two parts corresponding to these components. Together, these form a complete definition of the abstract machine.

3.2. Formal Description of the Primitive Substructure.

If it is accepted that a primitive P is dependent on a finite number of checks $c_1, c_2 \ldots, c_m$ which are truth functional then it is possible to obtain a formalisation for the primitive P.

It is an elementary result that given a formula $Q(R_1, R_2, \ldots, R_n)$ containing exactly the n propositional variables denoted by the syntactical variables $R_1, R_2, \ldots, R_n$, then the truth value of $Q(R_1, R_2, \ldots R_n)$ can be determined when the truth values of $R_1, R_2, \ldots R_n$ are known. Since

there are exactly 2^n ways of assigning truth values to these n variables there are exactly 2^n such determinations. The results of these 2^n determinations may be set out as a sequence in any of $(2^n)!$ ways (Rose [41]).

In the previous section it has been stated that a primitive consists of checks and possibly, depending on the result of these checks, some change of state of the system table. If the change of state is to occur, then the primitive P is designated the truth value True, otherwise False.

It is possible to represent the evaluation of P by a truth table containing the 2^m assignments of truth values to the $c_1, c_2, \ldots, c_m$ as shown in Table 6.1.

	1	2	3	4	2^m-1	2^m
c_1	T	F	F		F	F
c_2	T	T	F		F	F
c_3	T	T	T		F	F
.	.	.	.		.	F
.	.	.	.		.	.
.	.	.	.		.	.
.	.	.	.		.	.
c_m	T	T	T		T	F
P	V_1	V_2	V_3		V_{2^m-1}	V_{2^m}

TABLE 6.1.

The $V_1, V_2, \ldots V_{2^m}$ represent the truth value T or F taken by P under the assignments of truth values to $c_1, c_2, \ldots c_m$.

Again, it is an elementary result that a logical formalisation can be produced which is a representation of any truth table (Rose [41]).

Hence it is possible to express P as a logical expression which contains the propositional variables $c_1, c_2, \ldots, c_m$ connected by logical functors. Given any assignment of truth values to the propositional variables $c_1, c_2, \ldots, c_m$, the logical expression yields the truth value of P. It may be possible to simplify the logical expression as some assignments would not affect the truth value of P. However the principle of obtaining a logical equivalent for P is sufficient for this work.

The logical expression obtained may involve all the non-trivial unary and binary logical functors. However, the functors NOT and AND are functionally complete and thus it is possible to represent P by a logical expression containing the propositional variables $c_1, c_2, c_3, \ldots, c_m$, connected by NOT and AND only.

3.3. The Primitive Evaluation and the Associated Environment.

It has been suggested that the operating system is represented by the system table, the commands that can manipulate the system table, and the replies [36]. The commands that manipulate the system table in this analysis are the primitives. The replies are the verbal equivalents of the checks performed within the primitives and are not necessarily the messages of the

user profile. In §3.2 it was shown that the validity of a primitive action can be determined prior to its execution. The checks required are based on the values of the sub-system table constituting the initial state of the machine. If the primitive action is performed the change of state must be definable. Thus the final state of the system table can be shown to be different in some aspect from the initial state. The change itself is implicitly defined by the primitive action. The actual change of state effected by each primitive cannot be determined until the contents of the subsystem table have been specified. However, it is possible at this stage to devise a description method which will be used later. Thus, if a primitive P operates on a subsystem table defined by S and a change of state of the subsystem to S^* occurs then $P(S) \rightarrow S^*$. To define P it will be sufficient to specify how S^* differs from S. The relevant initial state will be defined by the prechecks made upon the table, and the changes will be dependent upon this state.

§4. Formal Description of the Filestore Subsystem.

The file directory has been defined to contain all the information concerning the file for both the user and the filestore system. Therefore, if the filestore primitives operate on the file identifier, the directory and the logical records only, then the filestore subsystem is clearly independent of the rest of the system provided no other primitives exist which operate on files.

It is postulated that if the filestore system is one which does not include the creation of users (i.e. the manipulation of a user structure comprises an independent subsystem) then "user" becomes a property of the file. Analysis has shown that in these circumstances seven primitive function types form a necessary and sufficient set for the definition of the filestore subsystem. Of these seven, there are three pairs, each consisting of an operation and its converse, while the seventh operates on the execution envelope. The first pair of primitives operate on the file name space, the second pair on the attribute space, while the final pair operate upon the logical record space.

The primitive functions and their semantics are described in the following subsections.

Notation.

File identifiers will be denoted by f, f_1, f_2. The state of the system consists of the file identifiers, the file directories and file contents and is denoted by S. The user context of the execution of a primitive

is denoted by u.

User identifiers are denoted u_1 , u_2 .

a_i denotes the i^{th} attribute.

V is new value attribute a_i.

A primitive is denoted by P.

The operation of primitive P on an initial state S with a user context u, dependent on files $f_1, f_2, \ldots f_n$ and with a final resultant state of S^* is denoted by

$$P\{u; f_1, f_2, \ldots, f_n(S)\} \rightarrow S^*$$

Other symbols $|$, $\in$, $\notin$, +, - have their usual set conotations.

The primitive preconditions are represented by a truth table and the state change by denoting the changes in S to produce S^*.

? in the truth table denotes that either T(true) or F(false) can be inserted without affecting the final truth value.

4.1. Filename Primitives.

The most basic state of the system which must be defined is the empty filestore. In this state no file identifiers, directories or logical records are present.

Clearly it must be possible to generate a file when the filestore is empty. Equally, it must be possible to remove existing files to produce the empty state. It would be nonsensical to permit files to be randomly introduced or removed from the filestore so it is reasonable to suppose that the primitives involved should be selective, operating on the smallest practical unit (the file) within the context of a given user identifier.

Similarly, because the conceptual spaces of the filestore are linked in a unidirectional chain (V §2.4) a filename must exist before a file can possess attributes or records.

The first primitive function is thus identified to be the removal of a filename from the filename space. If the name does not exist then the state of the filestore will be unchanged, viz:

$$P\{u;f(S|f\notin S)\} \rightarrow S$$

If the filename does exist it is conceivable that the request will not be compatible with the file attributes. An examination of the specification of the file directory defined in Chapter V shows that for a user to delete a filename he must have the correct access and the file must be free from other users before the request can be obeyed.

Hence

$$P\{u;f(S|f\in S,\ u\in\langle\text{delete filename list}\rangle,\ \langle\text{file free}\rangle\} \rightarrow S - f$$

This can be written as a truth functional, denoting the primitive by the symbol DELETE as shown below

$$\text{DELETE} =_T \sim\langle\text{file exists}\rangle v(\langle\text{file exists}\rangle \& (U\in\langle\text{delete file name access}\rangle)\&\langle\text{file free}\rangle)$$

where $=_T$ indicates truth value equality.
Thus, if this equation takes the truth value True, then the filename is deleted from the filename space which is equivalent to deleting the file from the filestore. Deleting a non-existent file is regarded as valid in this context. A truth value of False for the truth functional equation indicates that the primitive function cannot be performed and a suitable message can be produced. Any of the following messages could be suitable.

"User does not have delete access to ⟨filename⟩",
"File ⟨filename⟩ is being used",
or "File ⟨filename⟩ is not in the filestore".

Deleting a filename will not alter the contents space although the directory entry for the file can no longer be accessed.

The truth table equivalent of the DELETE primitive is shown below. (It is clearly only necessary to define the relevant part of the subsystem table which directly affects the truth value of the primitive function).

	1	2	3	4	5
file f exists	F	T	T	T	T
user has delete access to file f	?	F	T	T	?
file f free	?	?	F	T	F
DELETE(f)	T	F	F	T	F
Resultant State	S	S	S	S*	S

where

S* is S with the filename f deleted from the filename space.

The second primitive is the converse of DELETE and introduces a filename into the filename space. If a file exists which has the same name as the file to be created the operation is invalid as the uniqueness of filenames condition would be violated (V §2.2). Otherwise, the filename is added to the filename space. No attributes are associated with a newly created file.

The specification is as follows:

$$P\{u;f(S|f\notin S)\} \rightarrow S + f'$$

where f' denotes an entry in the filename space with a pointer to a null attribute entry.

$$P\{u;f(S|f\in S)\} \rightarrow S$$

Writing this as a truth functional equation and denoting the primitive by the symbol CREATE gives

$$CREATE =_T \sim\langle\text{file exists}\rangle$$

This produces the simple truth table

	1	2
file f exists	F	T
CREATE(f)	T	F

Resultant State	S*	S

S* is S with the addition of the filename f

4.2. The Attribute Space Primitives.

As each attribute has a name it would be possible to introduce primitive functions which operated on specific attributes only. However, the majority of the prechecks required for each primitive function would be identical and as a consequence only two primitives are needed for the attribute space provided the attribute identifier is included as one of the

parameters.

The primitive functions are to add or remove values from the attribute space of a specific file. From the file directory advocated in Chapter V it can be seen that the attributes are either lists of values, e.g. users with delete filename access, or a single value, e.g. storage mode. For attributes which are lists the new value can be added or an existing value deleted provided the operation is valid. Attributes which consist of a single value can be treated in different ways depending upon the interpretation of the primitives. For instance, when deleted the attribute could be undefined (c.f. variables yet to be assigned values in a programming language). Similarly, introducing an attribute value could overwrite the previous value (c.f. := of Algol 60). Alternatively an additional primitive could be used which changed the existing value. This primitive would only operate on single value attributes.

However, in this analysis for consistency, the primitive functions introduce and remove values for all attributes. This will not confuse the user nor produce undefined file directory values because the changes will be part of a user request. Thus, the primitives generated will be such that single value attributes are altered by a delete value operation followed by a create value operation. Attempts by the user to add values to this type of attribute will be trapped before primitive functions are generated.

The primitive functions can be expressed using the notation of assertions showing the change produced. However, for the sake of brevity, only the truth table definitions will be given which are shown below.

	1	2	3	4	5
file exists	F	?	?	?	T
user create access to attribute	?	F	?	?	T
new value valid	?	?	F	?	T
change consistent	?	?	?	F	T
CREATE ATTRIBUTE VALUE	F	F	F	F	F

Resultant State	S	S	S	S	S*

$S^* = S + f(a_i + v)$ if a_i is a list, otherwise

$S^* = S + f(a_i = v)$

The truth functional equation is:

CREATE ATTRIBUTE VALUE $=_T$⟨file exists⟩&⟨user access to attribute⟩&⟨new value valid⟩&⟨change consistent⟩

	1	2	3	4
file exists	F	?	?	T
user delete access to attribute	?	F	?	T
change consistent	?	?	F	T
DELETE ATTRIBUTE VALUE	F	F	F	T

Resultant state	S	S	S	S*

$S^* = S + f(a_i \sim v)$ if a_i is a list, otherwise $S^* = S + f(a_i = \text{void})$

The truth functional equation is:

DELETE ATTRIBUTE VALUE $=_T$⟨file exists⟩&⟨user delete access to attribute⟩&⟨change consistent⟩.

It is necessary that any change produced must be consistent with the remaining attributes within the file directory, thus interrelationships between attributes must be identified. For example, the attributes relating to user permission can become inconsistent for the following reasons:

1) Remove user identifier from "copy from" access when user "copying from"
2) Remove user identifier from "copy to" access when user "copying to"
3) Remove user identifier from "empty" access when user "emptying"

4) Remove user identifier from "execute" access when user "executing"

5) Remove user identifier from "attribute" access when user "changing attribute"

This type of change must be prohibited as clearly the system would no longer be consistent. However, other changes may occur at the primitive level which do create inconsistent states, but these are temporary, their duration lasting until the host user request has been completed. Thus at the user level the system will remain consistent.

Similarly some attributes, such as the media specification, will be machine dependent while others will apply to some machines only (a machine without magnetic tapes cannot have files whose directories state that the contents are stored on this medium). To prevent this type of inconsistency each attribute can be associated with a list of valid values. It is suggested that the values will depend upon the machine and the user contexts.

4.3. Logical Record Space Primitives.

The pair of primitives operating on the file logical record space have to some extent been anticipated by the file directory contents specified in Chapter V. It is expected that the file contents will be changed when user programs or system utilities, such as Editers, operate on the file. As stated in Chapter V these changes take

place within an execution envelope on facsimiles of the actual file contents. Once the execution process has terminated the operating system can either replace the contents in the filestore by the new contents from the execution envelope or leave the filestore unchanged depending upon the "success" of the execution process.

Altering the file contents in the filestore can take the form of either

1) Changing the contents and/or the order of existing records.

or 2) Adding new records to the existing records.

At the job control level these two types of change are sufficient as the differing types of detailed alterations take place within the execution envelope and are outside the job control function.

It is convenient, for the sake of simplicity, to have a single primitive which appends the contents of one file onto another. This adequately deals with 2) above but also, with the aid of the sixth primitive, permits changes of type 1). This primitive empties a file by clearing the logical record space of the file. The attribute space is unchanged except for the history which is updated accordingly. This permits file updating by applying the sixth primitive followed by the fifth which empties the file, then appending the updated file (which may be in the filestore or part of an execution envelope) onto the empty file.

In order to preserve consistency the two files

involved in the COPY APPEND must be marked as "copied from" and "copied to" as appropriate before the primitive function can be performed. The histories of the two files must be updated after the operation is completed. The COPY APPEND does not itself alter the attributes. A copy can only be performed if the attributes of both files are mutually consistent but an empty file can accept any input provided the user has the correct access to the file.

The fifth primitive changes the logical record space of a file taking the general form of

COPY APPEND ⟨file 1⟩ ONTO ⟨file 2⟩

The truth table for this primitive is shown below.

	1	2	3	4	5	6	7	8
File 1 exists	F	?	?	?	?	?	?	T
File 2 exists	?	F	?	?	?	?	?	T
Copy From access file 1	?	?	F	?	?	?	?	T
Copy to access file 2	?	?	?	F	?	?	?	T
File 1 free for Copy From	?	?	?	?	F	?	?	T
File 2 free for Copy To	?	?	?	?	?	F	?	T
File characteristics compatible	?	?	?	?	?	?	F	T
COPY APPEND	F	F	F	F	F	F	F	T

Resultant State	S	S	S	S	S	S	S	S*

where S* differs from S by

contents f_2 = contents $(f_2 + f_1)$

The assertion "File characteristics compatible"

is produced from the following table.

	1	2	3
Data type compatible	F	?	T
Storage mode compatible	?	F	T
File characteristics compatible	F	F	T

This table is the product of two further tables

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
File 1 binary	T	F	T	T	T	F	T	T	F	F	T	F	F	F	T	F
File 2 Empty	T	T	F	T	T	F	F	T	T	T	F	F	T	F	F	F
User access to change attribute contents type of File 2	T	T	T	F	T	T	F	F	T	F	T	F	F	T	F	F
File 2 binary	T	T	T	T	F	T	T	F	F	T	F	T	F	F	F	F
Data type compatible	T	T	T	T	T	F	T	F	T	F	F	F	T	T	F	T

The truth functional equation corresponding to this table is:

$\langle$Data type compatible$\rangle =_T$ ($\langle$file 1 binary$\rangle$ & $\langle$file 2 binary$\rangle$)

v ($\sim\langle$file 1 binary$\rangle$ & $\sim\langle$file 2 binary$\rangle$)

v ($\langle$file 2 empty$\rangle$ & $\langle$User access to change attribute contents type of file 2$\rangle$).

In the above, it has been assumed that data is either binary or text, however, there are many different text codes and this could result in incompatibilities when files are appended. To overcome this difficulty the analysis could be extended to prevent files which have different text codes from being appended. Alternatively, the operating system could automatically recode the file as it is copied making it compatible with the text code of the other file contents. This would result in new actual contents being produced. Ideally all files would be stored using a single internal code form. The table for "Storage mode compatible" is the same as that for "Data type compatible" except the assertions "File 1 binary", "File 2 binary" are replaced by "File 1 serial", "User access to change storage mode of File 2" and "File 2 serial" respectively. A truth functional equation can be generated corresponding to the resulting truth table.

The truth functional equation of COPY APPEND is:

$$\overline{\text{COPY APPEND}} =_T \langle\text{file 1 exists}\rangle \& \langle\text{file 2 exists}\rangle \& \langle\text{copy from access file 1}\rangle \& \langle\text{copy to access file 2}\rangle \& \langle\text{file 1 free for copy from}\rangle \& \langle\text{file 2 free for copy to}\rangle \& \langle\text{file characteristics compatible}\rangle.$$

A truth functional equivalent for ⟨file characteristics compatible⟩ can also be produced.

Emptying file contents is synonymous to deleting a file name or a file attribute as it takes the form of deleting the file logical record space. The actual contents space is unaltered. The primitive takes the form

EMPTY⟨filename⟩

and is defined by the following truth table.

	1	2	3	4
File exists	F	?	?	T
user has empty access	?	F	?	T
file free	?	?	F	T
EMPTY	F	F	F	T

Resultant State	S	S	S	S*

where S* = S - contents (f). (the attributes must be altered by attribute space primitives so as to be consistent with an empty file.)

The truth functional equation is

$EMPTY =_T$⟨file exists⟩&⟨user access to empty⟩&⟨file free⟩

4.4. Execution Envelope Primitive.

The final filestore primitive also deals with the contents of a file, but only passively as the contents are not changed directly by the primitive, although changes may occur as a consequence of the primitive having been invoked. The primitive action links a file contents to the machine processor(s) enabling user programs to be executed. The truth functional definition is as follows:

	1	2	3	4	5	6	7
File exists	F	?	?	?	?	?	T
File free to execute	?	F	?	?	?	?	T
File Binary	?	?	F	?	?	?	T
File stored on execute medium	?	?	?	F	?	?	T
Execute environment complete	?	?	?	?	F	?	T
User access to execute	?	?	?	?	?	F	T
EXECUTE (f)	F	F	F	F	F	F	T

Resultant State	S	S	S	S	S	S	S*

where S* differs from S by the file f being marked as "executing" which is a transient change only; the system reverting back to the initial state when the execution terminates.

In the above table it is assumed that the EXECUTE primitve can only operate on binary files stored on suitable media. This implies that prior to execution a primitive must be obeyed to copy the file contents from their normal storage medium to an appropriate medium accessible to the processor. The input and output files used by the executing program must be linked explicitly by the user, or implicitly by the system to form the execution envelope. Unless the envelope is complete the execution will not proceed.

§5. Independence, Completeness and Consistency.

It is possible to categorise the filestore primitives that have been developed in the preceding section. It can be seen that there are four types of primitive, three of these types corresponding to the parts of the file identified in Chapter V. Thus primitives exist to:-

1) manipulate the file identifier,

2) manipulate the contents of the file directory,

3) manipulate the file logical records,

4) execute file contents.

This is shown in figure 6.1.

Area Manipulated	Primitive functions
file identifier	CREATE DELETE
file attributes	CREATE ATTRIBUTE VALUE DELETE ATTRIBUTE VALUE
file records	COPY APPEND EMPTY
file contents in Execution Envelope	EXECUTE

Figure 6.1. Relation of Primitives to the file components.

If the operation of the primitive functions was strictly confined to the areas as shown in figure 6.1 then it would only be necessary to show that the primitive functions in each area were independent since the areas have been shown to be logically distinct.

File identifiers can only be manipulated by CREATE and DELETE and logical records are only altered by COPY APPEND and EMPTY. COPY APPEND and EMPTY do not themselves alter the attribute space although attribute space alterations must be performed in each case as a consequence of these operations if the filestore is to remain consistent. CREATE at first sight does appear to change the attribute space since it must provide a pointer to a null entry in that space. However, this does not involve any changes in the attribute space. In fact, all four of these primitives require CREATE ATTRIBUTE VALUE and DELETE ATTRIBUTE VALUE operations to be performed indivisibly with them in order to provide consistent updating.

5.1. Independence of the Pairs of Primitive Functions.

For each pair of primitive functions, one generates entries in the particular area manipulated while the other deletes entries. It is self-evident that no possible combination of deletions can ever produce the effect of a creation. Thus the members of each pair are mutually independent.

5.2. Independence of EXECUTE Primitive Function.

The file identifier and attribute spaces form the directory which is logically independent of the record space. EXECUTE only operates on the contents of logical

records thus it cannot alter the directory space and is therefore independent of CREATE, DELETE, CREATE ATRRIBUTE VALUE and DELETE ATTRIBUTE VALUE.

Similarly the execution envelope is distinct from the logical record space. EXECUTE can only change contents of records within the execution envelope thus it cannot alter the logical record space. Therefore EXECUTE cannot perform the functions of COPY APPEND or EMPTY.

The transfer of a file from logical record space to the execution envelope using COPY APPEND is identical to the output of a file contents i.e. the contents are transferred to a device and no record is retained of the contents in the filestore. Similarly, the transfer of file contents from the execution envelope after execution is equivalent to inputing files from outside the filestore. In both cases all operations on the directory and logical record spaces are performed by the appropriate combinations of primitive functions other than EXECUTE.

5.3. Completeness of the Primitive Functions.

CREATE and DELETE clearly allow any number of file identifier entries to be generated. Similarly, EMPTY and COPY APPEND permit any acceptable combination of existing complete files to be formed. Unlike logical records, attribute values have the same

property as files themselves in that every attribute has a name. Therefore CREATE ATTRIBUTE VALUE and DELETE ATTRIBUTE VALUE form a complete set operating on the attribute space. Thus, from the above, any consistent change can be made to the file identifier, attribute and record spaces.

It is clear that without EXECUTE the resulting set of primitive functions is deficient as the execution envelope would always return to the filestore unaltered which is equivalent to a "failed" execution. Thus without EXECUTE the contents of records are unchanged and no useful work can be performed.

5.4. Consistency.

It is imperative both for the user and the system that information in the filestore is self-consistent. Changes only occur by the operation of primitive functions. Assuming that every user request operates on an initially self-consistent state then this property will be retained provided user requests can only generate sequences of primitives which produce self-consistent changes.

It was noted in Chapter IV that the sequence of primitives generated as a result of the translation of a user request must be such that if the operation has to abort, then other primitive functions must be obeyed to return the system to the state existing when the user request was made. This is illustrated in

figure 6.2.

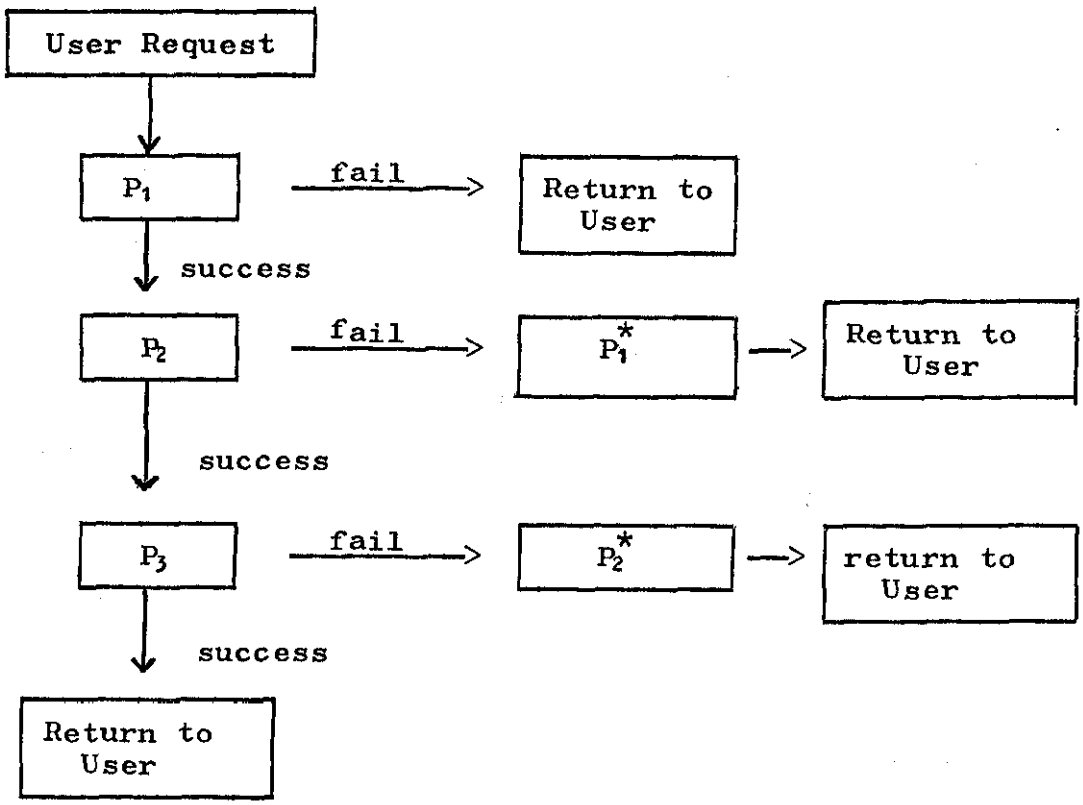


Figure 6.2. Retention of Self-consistency.

In the above the user request generates the stream of primitives P_1, P_2 and P_3 (for this example a simple serial execution is assumed). If P_1 is "successful", P_2 is obeyed and similarly P_3 is obeyed if P_2 is "successful". If P_1 "fails" no change will have occurred in the state table so the request can be abandoned. However, if P_1 is "successful" but P_2 "fails" then the state table will have been changed by P_1. Consequently the primitive stream P_1^* is obeyed reversing these changes. Similarly if P_3 "fails" the stream P_2^* will reverse the changes produced by P_1 and P_2.

Hardware malfunctions which create inconsistencies are regarded as outside the scope of the filestore system as described in this thesis.

§6. The Non-filestore Primitives.

The primitives obtained can be regarded as equivalent to the abstract machine function, producing changes in the state tables. It would appear that other primitives are required to define the resources of the abstract machine necessary for the set of primitive function described so far. The abstract machine could have infinite resources but this is an unrealistic model of the actual system. Thus, access to the abstract machine (jobs), can be visualised as a finite number of independent inputs each consisting of a finite length stream of primitive functions. For any of these streams a resource profile can be constructed and modified as the stream is processed. The profile is determined either by the implicit requests of the primitives or explicit requirements of the decoded user request.

It seems apparent that for primitives other than EXECUTE specific, fixed, predefinable resources are required, whereas, for EXECUTE each program will have differing needs. Many of these are implicitly defined by the files forming part of the execution environment. The input channels of the program are attached to files and information about them is available in the directory. The processor time and main storage required for execution, must also be supplied. However, in Chapter V it was deduced that main store is one of the filespace media and therefore can be attached to the execution environment as any other file. Consequently the user

can assign data space to a process by setting the desired value(s) in the execution environment for the main store file(s). (The space required for the process will be known to the system from the file directory).

Limits on the amount of information to be output to a serial file can be incorporated as an attribute "maximum file size" when the file is created, while access rates for direct access files would be required as an attribute both to enable scheduling when an execution takes place and to ensure that the logical records are stored on a suitable medium. These attributes would normally be set by default, however, if on a particular run the user anticipates a higher level of output he must be able to overwrite the default value for the limit prior to execution. Setting of defaults is considered as part of the next section.

The execution environment can also be used to contain information concerning the processor requirements for the execution and a time limit for the process. This has the advantage that all the information concerning the process is held together simplifying the function of the operating system.

Using the above concepts the execute environment has been extended to encompass specification of machine resources. The information is available to the operating system during execution and to the scheduler prior to execution from the directories of the files

involved.

A further part of the abstract system that has previously been mentioned in Chapter IV is the Activity Handler consisting of the logic binding user requests into calls on the primitive functions.

The logic has been shown to take the form

If X then Y

where X is a condition
and Y is either a sequence of primitive functions
or a further conditional.

(Infinite sequences are not possible as the input stream is of finite length).

This implies that the condition X is a test which may take the form of checks on system table values (e.g. existence of a file identifier). A discussion of the full range of available tests and their implementation is beyond the scope of this thesis, however, it seems reasonable to suppose that the primitive function prechecks would formpart of the test environment.

The "If-then" construction essentially provides a forward jump. At the user request level a method of looping and unconditional jumps are also desirable. However, these facilities are not considered part of the abstract machine although the job stream interpreter would need to cater for these operations. Any construction which does not lead to an infinite loop can clearly be expressed although recursion and loops

of unknown duration are not so readily managed. However, the job stream interpreter may be considered as a further automaton which either feeds commands to the abstract machine or obeys a job stream command. The job stream may be visualised as a finite length tape containing instructions which are the commands of the user interface. Some commands will generate primitive operations, some will be scheduling criteria while others will specify job stream manipulations such as jumps or repetition of commands. It is this last type that are obeyed by the job stream automaton which conceptually re-positions the tape at the appropriate command.

§7. Practical Considerations.

The analysis has concentrated on the definition of the primitive functions in terms of user requests made to the operating system. For the system to be user orientated the operating system must communicate with the user to inform him of either the changes which have taken place as a consequence of the requests, or the reasons why changes have not occurred. In Chapter III it was stated that the user interface to the operating system and the interface from the operating system to the user are both hierarchical. Consequently the information from the system at the user level may bear little resemblance to the actual events in cases of hardware malfunction. However, primitives in a user request will not be executed for specific reasons and these form a sub-class of system message. These messages relate to the truth tables of the abstract machine but may be decoded to provide user understandable messages. At the Activity Handler level conditions are inserted by the user request interpreter to bypass some primitives in the stream. These conditionals are used to determine the success (at the user level) of program compilation, for instance. The results of these tests will give rise to another sub-class of system message. The tests can be either string matching (as in GEORGE III) or numerical because, in general, the primitive stream will not be part of the user interface.

As the user requests are interpreted, default values will have to be included in the output stream of

primitives for values omitted by the user. This includes file names representing compilers, editers and other system utilities in addition to the more mundane items such as time, listing and storage limits. The default values must be chosen by the system manager to suit the machine and user environments. Thus a paper tape based machine running mainly Fortran jobs has defaults that reflect this machine and job profile. As the user interface is hierarchical the default set will also conform to this structure as stated in Chapter IV. The use of a default system implies the existence of the system utilities such as compilers and device drivers.

Many of the files which form a job within the computer system have temporary existence for the duration of the job and are not part of the user level. These must not only be removed by the system when the job terminates, but must be linked to the correct job as it is processed. It is reasonable to expect that each job will have a unique identifier supplied by user or system. As defaults for naming semi-compiled programs, binary programs, data etc. will be required, it appears to be practical to use the job identifier as part of the name for the components generated by the job in conjunction with further distinguishing codes.

The scheduling and sequencing of a job or job step is again either explicitly requested by the user or implicitly set by the system. The job scheduling must

be determined by variables preset or defaulted at the user level, but available to the operating system as a total description of the job requirements prior to job execution. This enables the scheduler to build a profile of the job so that the processing meets the user requirements without conflicting with the other scheduling criteria. The low level scheduling is independent of the job control and hence need not be considered.

Sequencing jobs or job steps, or parallel processing is generally at the request of the user and provision for this facility can be built into the job control language and the interpreter can generate the necessary code as part of the primitive output stream.

When a job step begins execution it is conceptually placed in a self-contained envelope as the EXECUTE primitive is obeyed. This requires the input and output files to be connected to the executing process. After the execution has terminated the envelope can be used to update the actual file contents provided the process has been "successful".

These considerations lead to an overview of the system as shown in figure 6.3 which indicates the constituent parts of the user interaction and the relevant parts of the system used at each stage. The scheme is valid in either on-line or batch mode the difference being that the replies/messages are returned to the user in on-line interactions whereas in batch

jobs they are returned to the command interpreter which can decide the subsequent action, that is repeat a loop, jump, or process the next command in sequence.

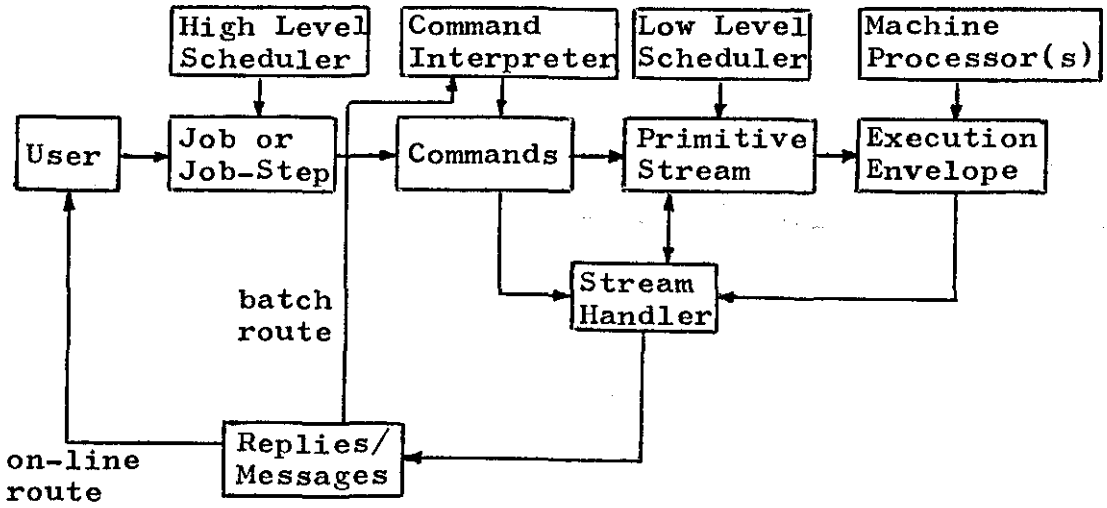


Figure 6.3. System Structure Overview.

In this chapter it has been shown that the primitive functions can be defined by a truth functional representation. It has also been demonstrated that the filestore forms a separate self-contained subsystem and that primitives can be chosen which are orthogonal and complete as discussed in earlier chapters. Furthermore it has been shown that the filestore primitives provide for a very significant part of the user requirement for doing useful work.

The next chapter describes how these primitives and the notions of the machine independent filestore have been used to implement a prototype system.

CHAPTER VII

IMPLEMENTATION OF THE FILESTORE SUBSYSTEM.

§1. Introduction.

In Chapter VI the primitive functions forming a sufficient set to define filestore operations were specified. To demonstrate that the theoretical development of these primitives was both sound and practical a prototype system has been implemented.

1.1. Objectives of the Demonstration System.

Notwithstanding the apparent feasibility of the theory, practical viability is the sole criterion by which such a theory should be judged. The aim of any demonstration must be that of showing the theory to be applicable even if this is limited to proving only the underlying principles. However, any restrictions imposed must not result in a system that is so artificial that it bears little resemblance to the theoretical ideas.

This chapter explains how the concepts developed in Chapter V for a machine independent filestore can be translated into a form suitable for implementation. Also the filestore primitives of Chapter VI are rewritten in an algorithmic form suitable for program coding. Extension of the practical ideas to a full implementation are also discussed.

The demonstration is intended to show:

1) That such a system can be implemented.
2) The completeness of the file directory and the primitive functions.

and 3) The system can be virtually free of machine idiosyncrasies.

It is obviously advantageous to minimise any limitations but those that have been found necessary appear to fall into two categories. Restrictions are required due to:

1) theoretical considerations, discussed in the next subsection,

and 2) limited time and manpower, discussed in §2.6.

1.2. Limitations of the System due to Theoretical Prerequisites.

The theory demands that the hardware of the host system can support a filestore. This not only implies that random access devices are available but also that the storage capacity is sufficiently large to permit a reasonable number of files to be on-line. Additionally there is the need for auxiliary storage on magnetic tapes, etc. for storing infrequently accessed files and security copies. The main store supports the peripheral devices by providing transfer and working space for file manipulations, again implying minimum requirements.

It is also to be expected that the system software of the host system is capable of handling requests associated with a filestore.

§2. Implementation of the Filestore System in Principle.

2.1. Structure of System.

The composition of the system is shown in figure 7.1.

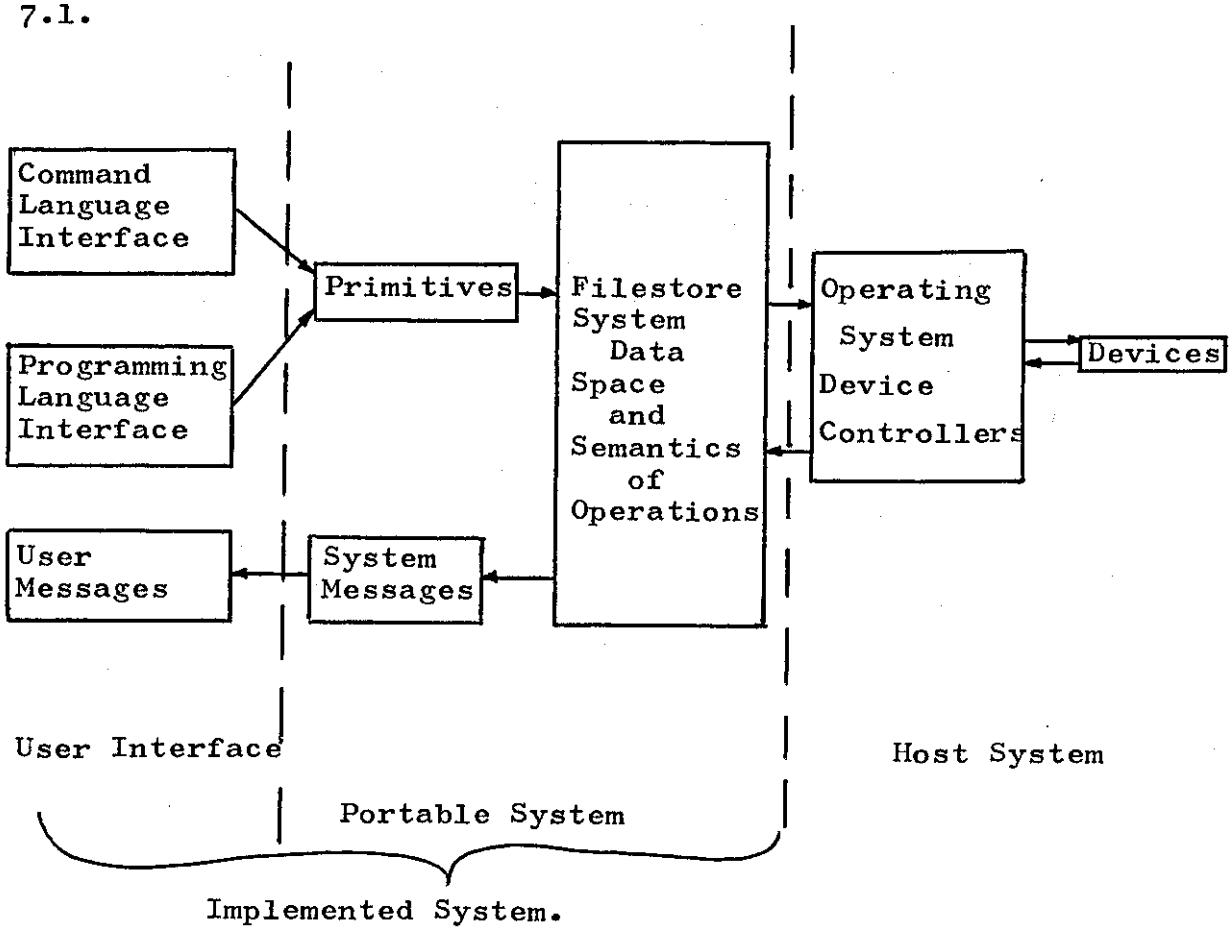


Figure 7.1. Structure of Implementation in Principle.

The three components of the portable system are:

1) the stream of primitive operations which have been compiled or interpreted from the user request stream,
2) the messages returned to the user,
and 3) the filestore system which consists of the data space for the files and the semantic definition of the filestore operations.

It is known from Chapter V that the files in the filestore consist of the file name, the file directory, and an optional set of logical records. In the implementation it is necessary to link the three conceptual spaces and the actual data in the file by indexes. Thus the file name has an associated physical address which is the position of the appropriate file directory in the file directory space. Similarly if the file is not empty the file directory contains the address of the list of logical records which, in turn, reference the actual file contents.

For practical purposes the filename space is a single entity held either in main store or, if too large, on fast random access devices. Efficiency requires that the number of transfers and search time are minimised, consequently the location of a name in the filename space should be determined from the information contained in the name. Typically, a hashing technique is considered suitable.

It may be that a more generalised access method is permitted whereby a subset of the required file's attributes can be specified as a substitute for the file name. In this case the user must expect the initial access to be slower because additional work is involved. Also there may be an added complication as several files may satisfy the attribute values specified.

The individual file directories could be stored with the corresponding filename in the filename space but this

would be unnecessarily large and difficult to manage. Thus it is envisaged that the directory space is separate from the filename space each stored in a file belonging to the operating system. It is conceivable that the logical record identifers could be incorporated into the directory space in some instances, but the contents space is formed by physically diverse media which are generally divorced from the directory space.

The organisation of the four filestore spaces is shown in figure 7.2.

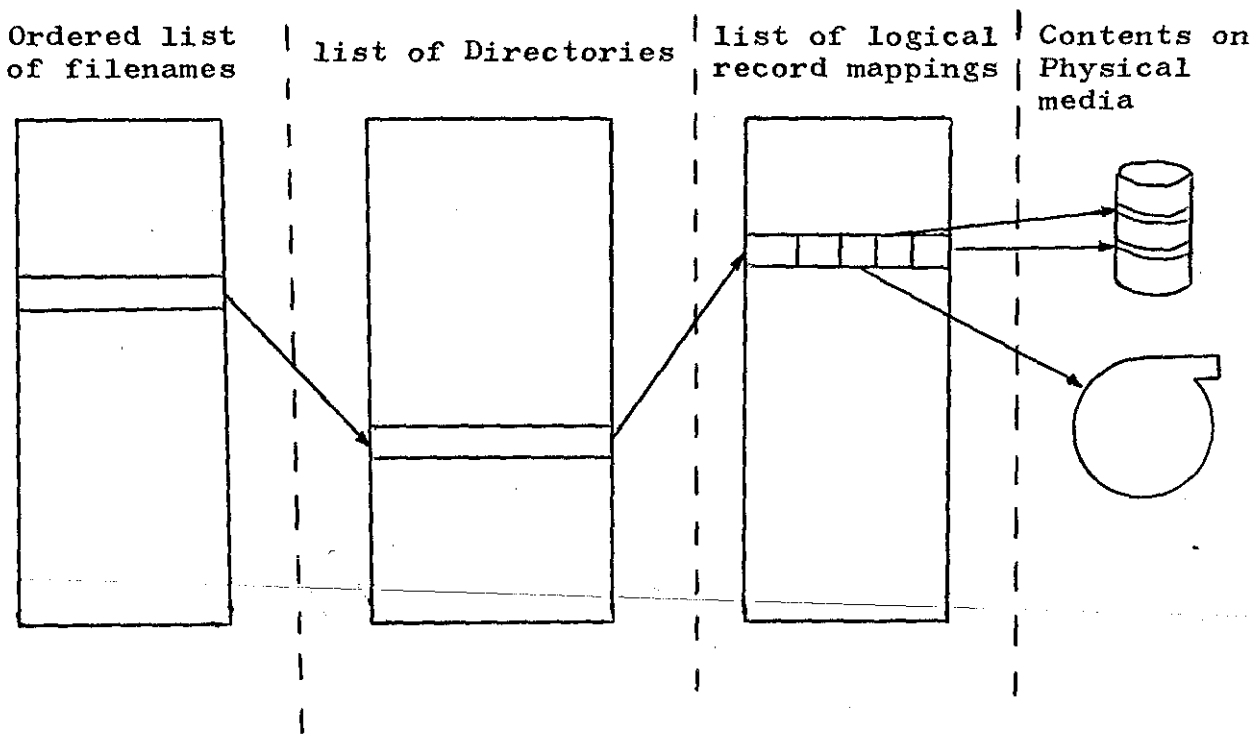


FIGURE 7.2. Organisation of Filestore Space.

2.2. Directory Structure.

Figure 5.2, which showed the file attributes and their structure, forms the basis for constructing a directory space. Some attributes will always require the same amount of directory space, whereas others will vary in size according to the file and its usage. Typically, the attribute denoting the "storage mode" can be accommodated by a single binary digit, but the list of "copy access" users, for example could be any subset of the users in the system. Consequently an external structure for the directory is necessary linking each attribute to the attribute value(s). This is shown in figure 7.3. The variable length attributes are:

List of users with execute access,
copy to contents access,
delete file access,
copy from contents access,
change attribute access,

Execution environment,

History.

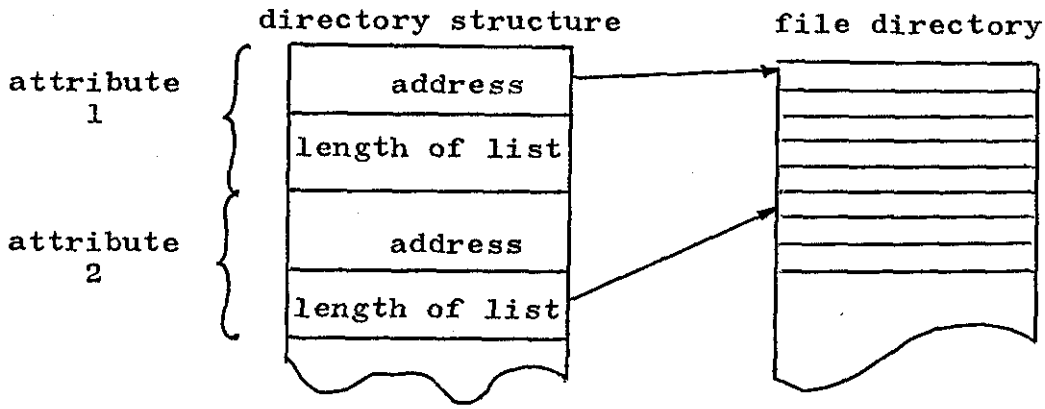


FIGURE 7.3. Directory Structure.

The attributes of fixed size can be represented by integers or bit patterns for the range of values. Media specifications can be represented by integers which are the addresses of the records containing full descriptions. This mechanism reduces the filestore space overhead and permits the operating system to alter the descriptions independently of the filestore system. The media descriptions can be used in user requests to specify the media required. Thus

PRINT⟨filename⟩ ON LINEPRINTER WITH UPPER AND
LOWER CASE CHARACTERS, 160 PRINT
POSITIONS, AT NOTTINGHAM UNIVERSITY.

can be interpreted into the simple primitive function

COPY APPEND⟨filename⟩ONTO 9

where 9 is linked to the appropriate device media within the system. The file(s) of the filestore representing this device have the integer 9 as the media

description in their directories. If this device should be rendered unusable then the integer can be changed (by the operators) to the integer value representing another suitable device.

2.3. Logical Record Mapping.

The form of the contents addresses in the logical record space are largely dependent upon the storage media, thus a disc, drum or magnetic tape can be referenced by the block identifiers, whereas a deck of cards can only sensibly be referenced as an entity. The addressing information in the logical record space of the filestore must be a complete definition of the physical location of the record contents. However, it would often be foolish to duplicate information common to all records so a file whose entire contents are stored on a single disc pack, for instance, can be identified by the track and block location for each record with the pack identifier stored just once for the whole file.

2.4. Implementation of Semantic Definitions of Primitives.

The primitive semantics as specified in Chapter VI are suitable for an abstract definition but are not in a programmable form. The truth value of the truth functional equation can be determined if the truth values of the items required for the evaluation are known.

If the resultant value is True then the actions associated with the primitive are performed. These actions occur internally resulting in changes to the filename space, directory space, the logical record space or externally resulting in the operating system device controllers transferring file contents.

For each table of Chapter VI it is possible to take each column in turn and transcribe the propositions into a series of logical tests which can be coded in any programming language. This results in a series of tests equal in number to the columns in the truth table. The evaluation could proceed column by column until one series yields the truth value "True" or all the series have been evaluated without producing this truth value. This process is clearly inefficient but Rose [41] shows that it is possible to simplify truth functional equations by re-ordering and reducing the number of evaluations required. Following these procedures the algorithms are simplified and the resulting programs are more efficient. When the evaluation process produces a truth value of "False" for a conjunct of the truth functional expression, a suitable message can be returned and the evaluation terminated. Although further tests may still remain, the truth value of the expression will be unaltered once a subexpression connected by AND takes the truth value "False".

2.5 Input/Output for the Filestore.

In a functional system the filestore and operating systems co-exist; the operating system driving devices etc. while the filestore system manipulates files in response to users job control. Some commands will be requests to input or output files through the real devices controlled by the operating system. Files designated for input/output form links between hardware and software. A user request to transfer a file to another medium is effected by appending the file record addresses to the logical record space of the system file associated by the filestore system with a device that copies to the specified medium. The filestore system, independently of the request, examines files of this type (known as system input/output files) and performs the transput as part of its normal operation.

Similarly, files are input either explicitly by requests forming part of a user job or implcitly by the input being presented at a physical device. In either case the input file is copied by the operating system (not the filestore system) into the system file associated with the device and/or medium. This system file is accessible to the filestore system.

The effect of this technique is equivalent to streaming input/output. However, it is believed that the concepts have been clarified and are consistent with the requirements of the machine independent filestore.

All manipulations of file contents are from one storage medium to another performed by the operating system in response to requests from the filestore system. For input streams the contents are conceptually transferred to the appropriate file within the filestore by transferring the logical record mappings from the system file to the record space of the user file. For output streams the contents are actually transferred to real devices via the buffers of the operating system.

2.6. Limitations of the Prototype System.

Figure 7.1 showed a suitable structure for the filestore system and the related semantics of the filestore primitives. Due to restricted resources the actual system implemented is a curtailed version whose structure is shown in figure 7.4.

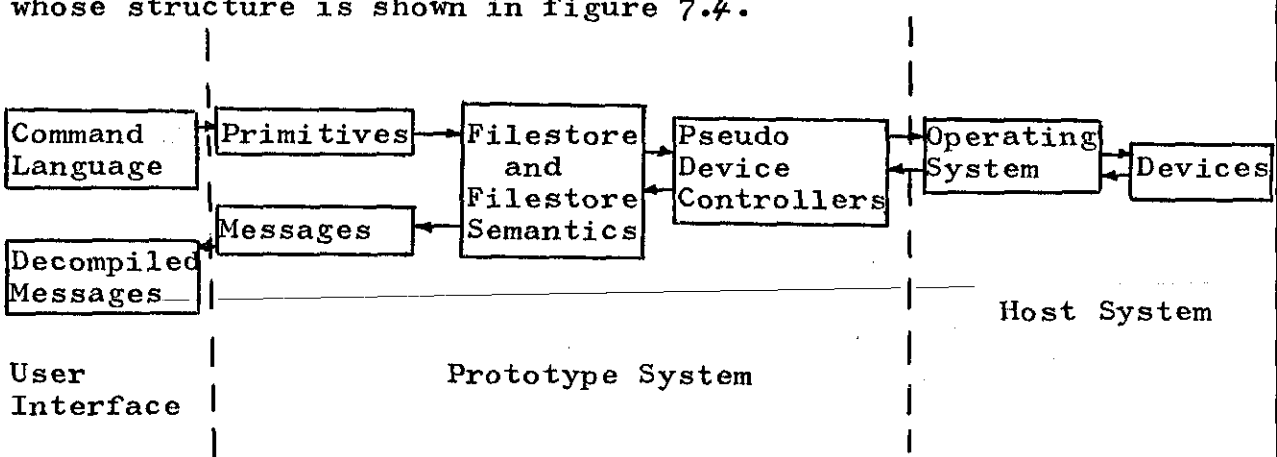


FIGURE 7.4. Structure and Extent of Prototype System.

It will be observed that the prototype system does not interface directly to the device control routines of the host operating system because an intermediate level has been introduced. This level of pseudo device controllers has been found necessary for two reasons. In the limited time available it was considered impractical to attempt to prove the viability of the system using techniques which would involve interfacing directly to the internal routines of the host operating system. Also it is obviously impractical to integrate a demonstration system into an existing operating system on any machine providing a user service. Since the machine used for hosting the prototype system was a main University service machine the demonstration had to be implemented and run as a normal user program. However, this had the advantage that the user facilities provided by the service machine were available for testing the programs.

If figures 7.1 and 7.4 are again compared it will be seen that the implemented system incorporates the user interface whereas the prototype system does not extend beyond the primitives and system messages.

Time constraints have not permitted:

either 1) the primitives to be interfaced to an existing command language,

or 2) a new command language to be designed and implemented,

with 3) the design and implementation of a system

message handler.

Since it is known from the results of Chapter VI that the primitives form a complete definition of the filestore operations it is sufficient to indicate the feasibility of the prototype system as shown in figure 7.4. From this it can be seen that there is no requirement to provide a tailored user interface as part of the demonstration system.

It is again an obvious consequence of the results of Chapter VI that when a primitive operates on its subsystem table the resulting state tables may become inconsistent. Several changes in the sub-system table may occur corresponding to the operation of the primitive. These are implied to take place simultaneously because each primitive is an indivisible unit, yet in practice it is sufficient that changes are completed before the next primitive in the input stream operates on the system table. It is not possible to demonstrate user request consistency using the prototype system because the highest level of user interaction is only the primitive stream. For the development of the theory it was not necessary to consider either the size of the file directory or the space required by the individual attributes. Any fixed size is invariably arbitrary and ideally the attribute fields would be unlimited. (User abuse of this facility can be prevented by

charging for the file space used). The constraints imposed on the prototype require that the space used by the demonstration system for representing the filestore on the host machine is limited, thus the numbers of the file identifiers, directories and logical records are bounded. Each directory has been limited to a maximum of 128 24 bit words although individual field lengths within each directory are permitted to vary in length provided the upper bound of the directory is not exceeded. Similarly each logical record has been limited to a maximum of 128 24 bit words.

The file names are limited to a maximum of any combination of twelve characters, excluding spaces. The name must be different from any other currently in the system, this being checked before the name is accepted. As the prototype system is small, any algorithmic procedure for locating file names is considered to be of little value and so names are located by a serial search.

The user identifiers are limited to a maximum of eight characters excluding spaces which, again, are prohibited. For convenience a special user identifier SYSTEM has been created having automatic access to all files and attributes. This greatly facilitated recovery from inconsistent states while program testing but would not necessarily form part of an actual implementation.

The historic information is generally generated by the user request interpreter which inserts the necessary

primitive functions into its output stream as the commands are decoded. Thus, because the history does not relate directly to the user it has not been incorporated into the prototype system.

In earlier chapters it was suggested that the contents of a file could be used by more than one user. This is possible if each logical record can be marked with its used status and users access the file at the logical record level. For the prototype system access to files is at the file name level only permitting a single user access to the file regardless of the mode of access.

One of the most desirable features which cannot be included in the prototype is a multiuser environment. Provision has been made in the program code for the checks necessary to permit several users but these are superfluous in the prototype. The main difficulty in achieving this type of environment is the protection mechanism of the host system GEORGE III which only allows one job to have access to any file.

Although the theoretical system permits selective access to each attribute of a file, the prototype allows access to all of the attributes to users whose identifiers are in the attribute access list as this is far easier to implement.

Details of the prototype system as implemented are described in the following section.

§3. Implementation of Prototype System.

3.1. System Files.

In the prototype system the filespace is represented by a number of "data files". (To avoid confusion with the files in the filestore the term "data files" will apply to the files forming the prototype system.)

A serial data file's records are each formed by a filename and an associated integer. These records are unordered, but each integer indicates the record number of a second data file which contains the corresponding file directory. The records of the directory data file are restricted to a maximum of 128 24-bit words although the internal fields can vary in size provided the maximum length of the total directory is not exceeded. Files which possess records also have entries in a third data file representing the filestore contents space. The logical records are linked to the contents data file by integers in the directory data file which are the indexes corresponding to the location of the records in the contents data file. These records are also limited to a maximum length of 128 24-bit words. To utilise fully the space in the directory data file real device descriptions, where appropriate, are represented by integers. These integers are indexes of a further data file containing textual descriptions of the devices.

Subsidiary data files are necessary for the primitive command stream, file records awaiting transfer to the filestore, and file records awaiting output to

physical devices.

3.2. The Programs and their Functions.

Portability demands that the programs are written in a standard high level programming language. The routines for the filestore semantics and the pseudo device controllers are written in Fortran IV because the language is readily available and possesses a reasonable degree of machine independence. It is envisaged that no great difficulty should be experienced in transporting these routines to other computer systems.

The main routine analyses the primitive input stream, placing the parameters into global variables for ease of manipulation. For each primitive there is a corresponding subroutine which performs the necessary checks to ensure the request is valid and if so, changes the data files accordingly. If changes are required in the structure of a file directory a further routine is used for these manipulations.

The checks in the routines which represent the primitives have been arranged so as to be intuitively efficient without aiming for an optimal strategy. The first check encountered which invalidates the primitive request terminates the routine and the next, if any primitive is then processed. A suitable system message is produced reporting the reason why the request could not be performed. (A certain amount of forward checking for further faults is feasible and the degree of help provided by the system can perhaps be determined

by the user scenario. This facility was not considered for the prototype system.)

The command stream is translated into a standard format for input to the portable component of the system by an Algol 68 program. Normally the input to this program would be user requests but for the prototype system the user interface is the primitive functions.

As the command translator is not part of the portable system the language most suited to the task has been chosen. As the program has to deal with text strings to check the syntax of the commands, comprehensive string handling facilities are desirable which precluded Fortran IV.

The input/output of the prototype system are dealt with by two further Fortran routines described in the next subsection.

3.3. File Input/Output in the Prototype System.

In the prototype system the operating system function is mimiced by initialisation and close-down phases.

In addition to storing the current states of the file directories into disc files which form the re-start data for the next run, the close-down routine also checks the output file streams. Any output file that is not empty is copied to the appropriate pseudo device (pseudo because in reality the operating system would copy directly to the device). For the demonstration system the operating system in the guise of the close-

down routine, copies the file contents, as denoted by the record identifiers in the streams to a single file to be printed later by the host system (GEORGE III). Each stream is headed by a file media/device description and several files may be in any single stream.

The close-down routine also clears the output and input file streams so the next run of the demonstration begins with no outstanding output and different input files. The input of files from file streams does not occur until the relevant primitive is obeyed. The appropriate file stream is searched for the required file name. If the name is present and the user identifier in the request and file stream are the same the file records are transferred to the file contents space of the filestore. If the name is absent from the stream an external request must be made (to the operators) to load the file into the appropriate device. If the file does not exist the filestore aborts the request. In the prototype system only files in the input stream can be transferred to the filespace.

3.4. Input of User Requests.

The user requests have to be input as primitive functions because the prototype system does not incorporate a command interface. Each interaction commences with a user identifier (necessary to check access to files) and terminates with either another user identifier, or the symbol END signifying no further primitives are in the stream. Each request is terminated

by a semi-colon which is superfluous while the primitives are syntactically correct. However, if the primitive currently being processed has syntactic errors the input stream is searched for the next semi-colon which defines the point at which the analysis restarts.

The user requests are input to the Algol 68 validation program which checks the syntax and some semantics. It is not possible to verify existence of files or user access to files because at this stage the requests are only partially decoded. Neither does the translator have access to the data files of the filestore.

The output from the translator is a list of numeric codes, each primitive function denoted by a unique number, followed by the parameters for each primitive in a predefined format.

3.5. Garbage Collection of the System Disc Files.

When a file is deleted from the filestore its name is removed by compacting the list of filenames. The directory entry remains in the disc file although the filename which did previously index it is no longer in the system. However, the list of directory indexes is modified by clearing the index number which was associated with the filename. This indicates that a current directory entry is no longer stored in the disc record. When a subsequent file is created the list of used indexes is searched and the first one

that is free (denoted by a clear location) is used to store the new file directory.

The file contents space is treated somewhat differently. When a new set of records have to be stored in the filespace, the index for the contents space is scanned for empty cells. These denote unused record locations and each may be used to store a new record. If a file is deleted no change occurs in the index or the filespace because other files may index the records of the deleted file. However, when the index becomes full, more space is generated by checking each index against the current file records in the filestore. If a record is not in any file the index can be used for a new record. This procedure is only used when space is required for new records although in an actual implementation it could be a garbage collection routine used at regular intervals.

3.6. Example of Protogype System

A typical sequence of primitive functions constituting a user interaction is shown below. (The line numbers in brackets are merely for the convenience of the reader to relate later items).

```
CREATE FILE1;                                          (1)
USER USER3;                                            (2)
CREATE FILE2;                                          (3)
USER SYSTEM;                                           (4)
CREATE ATTRIBUTE 2 VALUE USER3 OF FILE2;               (5)
CREATE ATTRIBUTE 4 VALUE USER3 OF FILE2;               (6)
CREATE ATTRIBUTE 3 VALUE USER3 OF FILE2;               (7)
CRETE FILE3 ;                                          (8)
COPY FILE2 FROM CARD-READER;                           (9)
USER USER2;                                            (10)
DELETE FILE2;                                          (11)
CREATE FILE3;                                          (12)
USER SYSTEM;                                           (13)
CREATE ATTRIBUTE 2 VALUE USER2 OF FILE3;               (14)
CREATE ATTRIBUTE 3 VALUE USER2 OF FILE3;               (15)
CREATE ATTRIBUTE 4 VALUE USER3 OF FILE3;               (16)
CREATE ATTRIBUTE 4 VALUE USER2 OF FILE3;               (17)
COPY FILE3 FROM TAPE-READER;                           (18)
DELETE ATTRIBUTE 4 VALUE USER2 OF FILE3;               (19)
USER USER3;                                            (20)
COPY FILE2 TO FILE3;                                   (21)
USER USER2;                                            (22)
COPY FILE3 TO LINE-PRINTER;                            (23)
END;                                                   (24)
```

The corresponding messages returned to the users from the Algol 68 program interpreting these commands are as follows:

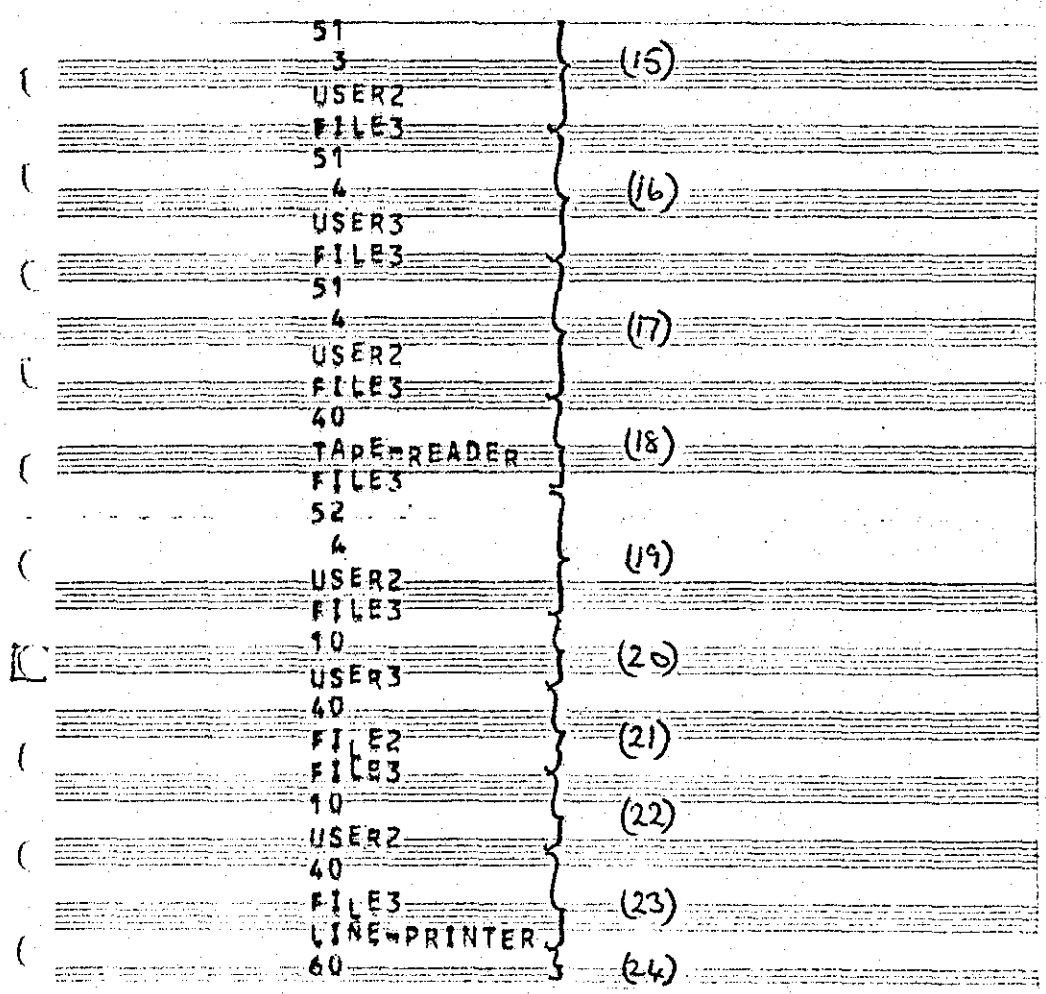
```
SEQUENCING ERROR-USER IDENTIFER REQUIRED
BEFORE ANY COMMANDS.                          (1)
CREATE FILE1 IGNORED.

CRETE FILE3 UNINTELIGABLE.                    (8)

END OF INTERPRETATION.                        (24)
```

The interpreted commands are stored in an intermediate file which contains the following information:

```
10
USER3                                         (2)
20
FILE2                                         (3)
10
SYSTEM                                        (4)
51
2
USER3                                         (5)
FILE2
51
4                                             (6)
USER3
FILE2
51
3                                             (7)
USER3
FILE2
40
CARD-READER                                   (9)
FILE2
10                                            (10)
USER2
30                                            (11)
FILE2
20                                            (12)
FILE3
10                                            (13)
SYSTEM
51
2                                             (14)
USER2
FILE3
```



The numeric codes correspond to the commands and the parameters appear in a predetermined order following the command identifier. Only one item is placed on each line of the file to facilitate ease of input to the Fortran program.

Assuming the filestore consists of the three system stream files CARD-READER, TAPE-PRINTER and LINE-PRINTER only, the following messages are produced as the commands are obeyed.

```
FILE2          CREATED                                    (3)
USER3    ADDED TO ATTRIBUTE  2 OF FILE2                   (5)
USER3    ADDED TO ATTRIBUTE  4 OF FILE2                   (6)
USER3    ADDED TO ATTRIBUTE  3 OF FILE2                   (7)
CARD-READER  STREAM COPIED TO FILE2                       (9)
USER2    DOES NOT HAVE DELETE ACCESS TO FILE2            (11)
FILE3          CREATED                                   (12)
USER2    ADDED TO ATTRIBUTE  2 OF FILE3                  (14)
USER2    ADDED TO ATTRIBUTE  3 OF FILE3                  (15)
USER3    ADDED TO ATTRIBUTE  4 OF FILE3                  (16)
USER2    ADDED TO ATTRIBUTE  4 OF FILE3                  (17)
TAPE-READER  STREAM COPIED TO FILE3                      (18)
USER2    REMOVED FROM ATTRIBUTE  2 OF FILE3              (19)
FILE2          COPIED TO FILE3                           (21)
FILE3          COPIED TO LINE-PRINTER STREAM             (23)
END OF RUN                                               (24)
```

To demonstrate the resultant state of the filestore it will be assumed that the Card Reader stream contains three records viz

THIS IS

THE

CARD INPUT

and the Tape reader stream contains two records viz

TAPE INPUT

STREAM

Also attributes 2, 3 and 4 are assumed to be change attribute, copy from and copy to access respectively.

The printout of the filestore at the end of the command processing would be as follows:

```
                    CONTENTS OF FILESTORE

FILENAME=FILE2
ACCESS PERMITS ARE AS FOLLOWS:
DELETE FILE= NO USERS
CHANGE ATTRIBUTES= USER3
COPY FROM= USER3
COPY TO= USER3
EMPTY CONTENTS= NO USERS
EXECUTE= NO USERS
FILE CONTENTS ARE TEXT AND SERIAL
RECORD NUMBERS ARE:
  1  2  3
CONTENTS ARE:
     THIS IS
     THE
     CARD INPUT


FILENAME FILE3
ACCESS PERMITS ARE AS FOLLOWS:
DELETE FILE= NO USERS
CHANGE ATTRIBUTES= USER2
COPY FROM= USER2
COPY TO= USER3
EMPTY CONTENTS= NO USERS
EXECUTE= NO USERS
FILE CONTENTS ARE TEXT AND SERIAL
RECORD NUMBERS ARE:
  4  5  1  2  3
CONTENTS ARE:
     TAPE INPUT
     STREAM
     THIS IS
     THE
     CARD INPUT

END OF FILESTORE
```

The line printer output produced is as follows:

```
LINE=PRINTER OUTPUT STREAM=NUMBER OF FILES= 1

TAPE INPUT
STREAM
THIS IS
THE
CARD INPUT


END OF STREAM
```

CHAPTER VIII

APPLICATIONS

§1. Introduction.

The main objective of this thesis has been the development of a portable basis for command languages. This has been realised through the definition of primitive functions which satisfy the preconditions specified in the early chapters of this thesis.

In conjunction with the primitives it was found necessary to formulate concepts concerning the machine independence of the operands associated with the primitive functions. This led to the development of a machine independent filestore.

It has been possible to identify applications of these developments which are outlined in the following sections.

§2. Database Security.

Data security can be maintained by various physical, hardware and authorisation procedures. The physical checks are not a matter of data management controls but the others could form part of the computer system.

Hardware protection can take the form of locks on main and auxilary stores by assigning protection keys to each block, and similar keys to each program. The operating system will only perform transfers if the program and data keys agree.

Files can be treated in a similar manner with keys in the job control which must match the keys stored as part of the file.

A further method which can be employed is to supply each user with a password, and the system can associate each password with a predefined limited access to files, programs, or data entry points. It is also necessary to prevent a user self declaring his authority to access any given file.

File security, specifically in a database environment, can be maintained more readily if the concepts of the logical filespace are applied. Each non-empty file in the filespace contains information stored on physical media. Users can access a file if they are included in the appropriate permission list of the file attributes. The actual information can only be accessed through system routines which use the logical record mapping retained in the directory. Thus a user can be provided access to a particular

subset of the database by giving him permission to access the appropriate files. Any subset can be constructed from the given data set without need to replicate the information as the logical record space provides the necessary mapping. The database manager must construct the file profiles with the appropriate access permission for each user. Clearly more file directories will be required but it is believed that the housekeeping necessary is reduced.

§3. Database Integrity.

Ideally any system should provide the user with protection against corruption of the data stored on his behalf caused by hardware error or interaction with other users. The only method of safeguarding against hardware error is to provide sufficient redundancy ensuring that the original contents can be restored in the event of a particular failure. Since keeping extra information increases the cost of storage and furthermore no system can provide a 100% guarantee of information integrity the actual protection provided usually represents a compromise. As was noted in Chapter V protection of all files represents a waste of resources in many cases while if the user has to copy files explicitly this is a waste of user time. The filestore concepts described herein permit the degree of protection provided to be determined by the user at the file level and then implemented automatically by the system. This minimises cost by allowing the appropriate users to be charged directly for this facility.

A single logical file can be mapped into several identical physical copies which can be stored on different media to reduce cost or increase security. This corresponds to the dumping concept in GEORGE III and also to the user copy in systems with no automatic protection. (The various protection techniques practised have been described by Davenport [15]).

Automatic switching between copies can occur if a particular copy is found to be corrupt or missing from the system (e.g. sum check failure or loss of disc pack).

Protection against inconsistencies arising from two or more users updating information simultaneously is provided by the execution envelope which ensures that all the files required are available for use before processing starts. This also prevents deadly embrace.

A further advantage of the execution envelope is that all operations during processing are performed on a logical copy of the file and not on the filestore copy. This is only changed after the execution is successfully completed and thus ensures against loss of integrity arising from programs terminating part-way through a file update. It does, however, require that the filestore copies which will be written to at the end of the process are locked while execution proceeds. In most cases different applications will use different subsets of the data, thus different files (and contents). Therefore locking a file will not necessarily impede other applications.

Finally, the system also provides for protection against user error since it is possible for the file contents to be associated with the history of the file. As has already been stated in Chapter V, the update of a record is actually the creation of a new record and the deletion of the old one. A history of the file can therefore be maintained if the old records are not deleted but each logical record has times and dates

of its validity. Thus the file contents at any time and date can be reconstructed and since the protection is at the file level the physical protection of multiple copies etc. can still be applied.

§4. Checkpointing.

The problem of effective and efficient checkpointing has been the subject of some discussion (e.g. [8]) without any clear solution being reached.

If the execution envelope as a whole is retained in the file directory then a simple extension to the filestore system described provides a complete solution. It has been stated that a file is defined by its name, which in turn acts as a pointer to the file directory containing the file attributes. The directory of an executable file which is currently being executed is no exception. Thus the directory contains all the information known about the file while it is executing. Hence it is only necessary to record the contents of the file directory to obtain a checkpoint mechanism. This will contain:

1) The attached filenames (including the location descriptor of each such file),
2) The current positions in each of the attached files (conceptually to a copy of the file in logical file space),

and 3) The current address within the execution (which is a special case of 2)).

If a program has to be restarted the checkpoint record can be used to regenerate the execution at which ever checkpoint is chosen.

This principle may be used recursively. The ultimate

stage is to have the whole system defined by a single checkpoint record which could be used to restart the machine when the system has malfunctioned. A special purpose boot-strap program would operate on this first checkpoint record and the system could be progressively rebuilt from other checkpoint directories.

§5. Networks of Computers.

The logical filestore is particularly applicable to a loosely coupled network of machines all of which can operate independently if the network is severed. The network should appear to each user as if it is a single system, being indistinguishable from a self-consistent extension of facilities on each of the stand alone computers. The user may require facilities which are only available at a particular site which is not his own, but information about the site would normally be deduced by the system from a study of the facilities requested rather than by requiring the machine to be specified explicitly.

The conventional approach to networks only allows on-line files to be accessible to each user. Each file that is required, but not available in the filestore associated with the users local machine has to be explicitly fetched. This creates problems of renaming to meet the uniqueness criterion, ensuring the security of duplicate copies, and the integrity of the files when updated. This is because the files are explicitly manipulated across system boundaries.

If the boundaries are removed then the conceptual problems of the user largely disappear and the task of the system implementor can be greatly reduced.

The logical filestore approach permits the user to see the concatenation of the individual filestores

with no explicit boundaries. The user accesses the file he requires "directly" (the implementor provides different mapping functions to connect the user request to the physical instance of the file). For the simple minded user he will only "see" the filestore of the machine which he uses.

The files are accessed through a directory which provides the unique filename composed of user identifier and user filename. This points to a conceptual file, which in turn, through the mapping function provides the physical location, or locations, of the file which could be on any machine. The filename provides the link between the directory and the logical filespace. For each directory entry there is one, and only one, entry in the logical filespace. The mapping function links the filename to the physical media. There may be none, one, or several entries in the physical filespace depending on the file being a name only, a single copy, or many copies (for security or because it is being used on several machine).

The name of the user automatically links the search to the relevant part of the file directory. This may mean that the search is performed on the physical media attached to the local machine or may require a request to the machine associated with the user name.

Files stored on the physical media of a remote machine can be accessed from any other machine in the network by specifying the file name. This provides

the file contents currently held on the remote machine's segment of the filestore. Consequently the user always obtains the latest version of the file because the updated contents are available to the network automatically. However, if the user does not wish to access updated versions, for example he may not want to use a partially tested program, then he must explicitly copy the file to his own machine's segment of the filestore. It is this copy that will then be used regardless of changes to the original file contents.

The principles of the logical filespace are currently being used in connection with the SRC contract B:RG:7010 awarded to the Computer Studies department of Loughborough University. The grant has been awarded for an investigation into the effective use of multiprocessor configurations as described by Evans and Newman [18].

§6. Command Language for a Network of Computers.

Proposed network command languages take several forms. Chupin [9] believes that each machine can have a different command language but the network operations are performed by a network command language common to all the machines. LE/1, the SOC network language described by du Masle [28] is rather like IBM's OS/360 JCL and requires the user to have explicit information of the network structure.

Many of the difficulties associated with the networks disappear if the machines involved have command languages that are based upon the same set of primitive functions (although the individual command languages can be different).

This permits each machine to accept jobs and decode the requests into a series of primitive activities. (This would be the usual process and is independentof the network environment). The resources required by the activities are determined by the operating system resource and scheduling routines and a decision is made either to:

1) perform all of the job locally,

or 2) direct some of the job steps to other machines in the network,

or 3) direct the whole job to the network.

It is a simple matter for the job (or job step) to be transferred around the network in the form of a series of primitive actions. These can be interpreted (without

any additional software requirement) by all the machines just as though the job had been submitted locally. This removes the need to have machine to machine interpreters or a separate network command language. For the user, the main advantage is that he can access the network facilities without knowing that he is doing so . Neither does he need to learn any additional information.

It is possible to design a command language model for the network applications. The model must reflect all the points made earlier in this thesis.

The structure is shown below. As the user progresses down the levels the facilities become increasingly detailed. Of the four levels shown below, the top-most level (0) does not involve the user in any knowledge of the network at all and corresponds to the inner-most ring(s) of figure 4.2. (By hypothesis this satisfies most of the user-uses of the system.) Although the user does not know about the network this does not imply that he will not be using it. If a user requests a particular package and it is not available on the machine at which his job was input, then the job will be transmitted to a machine which does run the package and the results will be returned to the user's local machine. The network may also help him if the machine he is connected to is overloaded and cannot handle his job within the timescale he has specified. (Provided

he gets his output back in time it is immaterial to him which machine processed his job.)

The three levels of network usage, working progressively down, can be thought of as:

1) the user exercising a choice between facilities,
2) user knowledge of the existence of subsystems, and 3) user knowledge of the detailed structure of the network.

Hence the model is as shown in figure 8.1

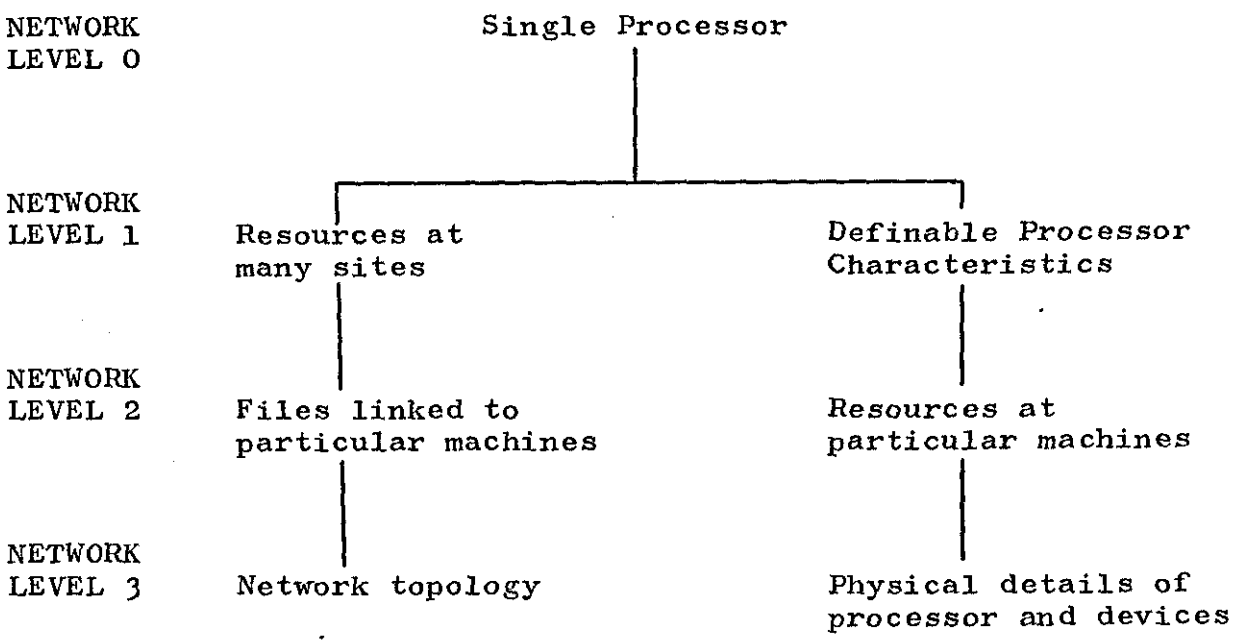


FIGURE 8.1. LEVELS OF NETWORK USAGE.

The parallel structure at each level indicates components to be independent yet requiring approximately the same degree of expertise. The serial structure, on the other hand, implies that knowledge of any level is more

complex than preceeding levels, and knowledge of the upper levels is generally a necessary condition for use of a lower level.

The ideas expressed above have been expanded to provide suggestions as to a possible user command interface for a network in a paper presented at Online 75 [33].

CONCLUSIONS

Design of Operating Systems.

In the early pages of this thesis Barron was quoted as saying that ideally the operating system should be the implementation of the command language [3]. This thesis has derived primitives which are user and machine independent but are intended to be interfaced to existing operating systems. This is necessary at present due to the time and cost involved in interfacing the primitives directly to the computer hardware. Consequently, the primitives are mapped onto existing command languages and a suitable user interface maps onto the primitives, this being a method of achieving portability.

However, if in the future a new computer is developed, it is believed that the primitives would form a suitable abstract description for the operating system provided the machine independent filestore is also used.

The advantages are twofold:

1) The structure of the operating system designed from the primitives is expected to be clearer, as it would be hierarchical, reflecting the default structure progressively until the hardware is precisely defined. Systems so designed would be efficient as the user requests are mapped into an operating system which is structured from the primitive functions.

2) The user interface can be built upon the primitives independently of the machine; the system designed has the potential to give an interface to suit the individual user types.

Thus, this thesis represents the first stage of a clearly defined schedule. The next two stages are identified to be:

1) Linking existing command languages to the portable basis,

and 2) Designing new user interfaces using the portable basis.

Linking Existing Command Languages to the Primitives.

At present command languages are constructed to function with a particular machine operating system and are consequently machine dependent and unstructured. To obtain portability the user requests in the jobs must be expressed in machine independent terms. This is possible if the requests can be mapped onto primitive functions which are themselves mapped onto the existing command language. Chapter VII has indicated that this is a feasible proposition when GEORGE III is the host language. For portability to become a reality it is necessary to show that the primitive functions with the machine independent filestore can be mapped into other existing command languages. It seems that this is a feasible proposition since it has been proved possible for both UNIQUE and GCL.

Design of New Command Languages.

Having shown that the primitive functions can represent the real machine (either through mapping onto an existing command language or to a new system based on the primitives) the next logical stage is to build new command languages onto the primitive set. These languages would necessarily exibit the criteria specified in Chapters III and IV.

There would be no need to limit the number of languages as each user group could require different facilities. However, it is thought that generally each language will be a recognisable dialect of a general purpose language.

Provided the languages mapped into the primitive set portability would be retained and the design of the languages, limited by the preconditions, should ensure their usability.

An important part of a new language would be the definition of a suitable default structure. Clearly the default values would differ between machines as individual installations would have their own limitations on job times, printer output etc. It would seem sensible to associate the default values with the system utilities provided at each site where this is possible.

REFERENCES

1. Anscombe, F.J. Rectifying Inspection of a Continuous Output. J. Amer. Statist. Ass. 1958.

2. Barron, D.W. Computer Operating Systems. Chapman and Hall, 1971.

3. Barron, D.W. Job Control Languages and Job Control Programs. Computer Journal, Vol. 17 No. 3 August, 1974.

4. Barron, D.W. Job Control on the ICL 2900 Series. Computer Bulletin. Series 2 No. 7 March 1976.

5. Barron, D.W. and Jackson, I.R. The Evolution of Job Control Languages. Software - Practice and Experience. Vol. 2, 1972.

6. Beech, D. Command Languages edited Unger C. North-Holland 1975.

7. Bredt, Thomas H. Command Language Processing in Formally Described Operating Systems. Command Languages edited Unger C. North-Holland 1975.

8. Chapin, N. Programming Computers for Business Applications. McGraw-Hill, 1961.

9. Chupin, J.C. Command Languages and Heterogeneous Networks. Command Languages edited Unger C. North-Holland, 1975.

10. Cheetham, C.J. and Wickham, R. Cafeteria Systems. IUCC Newsletter Vol. 2, No. 1.

11. Cowan, Richard M. Burroughs B6700/B7700 Work Flow Language. Command Languages edited Unger C. North-Holland, 1975.

12. Cox, D.W.J. Job Control Language - the way forward. Computer Bulletin. Series 2, No. 3, March, 1975.

13. Dakin, R.J. A General Control Interface for Satellite Systems. Command Languages edited Unger C. North-Holland, 1975.

14. Dakin, R.J. Preliminary GCL User Manual. Culham Laboratory, 1974.

15. Davenport, R.A. Database Integrity. Computer Journal. Vol. 19, No. 2, May, 1976.

16. Enslow, P.H. Summary of the Working Conference. Command Languages edited Unger C. North-Holland 1975.

17. Enslow, P.H. Operating System Command Languages, A Brief History of their Study. Command Languages edited Unger C. North-Holland, 1975.

18. Evans, D.J. and Newman, I.A. Usage Considerations in Multiple Processor Systems. Proceedings Software 1974.

19. Frank, G.R. Job Control in the MU5 Operating System. Computer Journal Vol. 19 No. 2 May, 1976.

20. Green, J.A. Sets and Groups. Library of Mathematics Series. Routledge and Kegan Paul Ltd.

21. Gilbert, W.S. The Mikado. Chappell and Co. Ltd.

22. IBM System/360 Operating System: Job Control Language. IBM Manual L28-6539-7.

23. James, S.K. and Newman I.A. The Incidence of Compiler Detected Errors in Algol and Fortran on a 1906A. Department of Computer Studies Report, University of Technology, Loughborough.

24. Jensen, Jørn and Lauesen Søren. Programming Language Extensions which render Job Control Languages Superfluous. Command Languages edited Unger C. North-Holland 1975.

25. Krayl, H. Unger C. and Weller T. Portability of JCL Programs. Command Languages edited Unger C. North-Holland 1975.

26. Lucas, P. and Walk, K. On the Formal Description of PL/1. Annual Review in Automatic Programming Vol. 6 Part 3, 1969.

27. Machine Independent Command Language. Computer Bulletin. Series 2 No. 4, June 1975.

28. Masle, J. du. An Evaluation of the LE/1 Network Command Language Designed for the SOC Network. Command Languages edited Unger C. North-Holland 1975.

29. Morris, D. Job Control on the ATLAS and MU5. Job Control Languages edited Simpson, D. NCC 1974.

30. Newell, G.B. The Family of Operating Systems called GEORGE. Job Control Languages edited Simpson, D. NCC 1974.

31. Newman, I.A. Machine Specific Facilities in a Machine Independent Command Language. Command Languages edited Unger, C. North-Holland, 1975.

32. Newman, I.A. User Oriented Message Handling, a new approach to error message presentation. Database Technology. Online 1976.

33. Newman, I.A. and Fitzhugh, N.S. Command Language Design for Networks of Processors. Communications Networks. Online 75.

34. Newman, I.A. and McConachie, M.A. The UNIQUE Machine Independent Command Language. Job Control Languages edited Simpson, D. NCC 1974.

35. Nicholls, J.E. Job Control in IBM System/360 and 370. Job Control Languages edited Simpson, D. NCC 1974.

36. Niggemann, N. A Method for the Semantic Description of Command Languages. Command Languages edited Unger C. North-Holland 1975.

37. Operating Systems GEORGE III and IV. ICL manual 4345 (1.76).

38. Orwell, G. Animal Farm. Penguin Books.

39. Parsons, I.T. A High-Level Job Control Language. Software - Practice and Experience Vol. 5 1975.

40. Rayner, D. Recent Developments in Machine Independent Job Control Languages. Software - Practice and Experience Vol. 5 1975.

41. Rose, A. Computer Logic. Wiley, 1971.

42. Shearing, B.H. Job Control Languages - As they are and as they might be. Job Control Languages edited Simpson D. NCC 1974.

43. Sibley, E.H. The Operating Systems Command Language Task Group. Technical Report. Command Languages edited Unger C. North-Holland, 1975.

44. Stoy, J. and Strachey, C. OS6: An operating system for a small computer. Computer Journal Vol. 15 1972.

45. Tozer, E.E. Preservation of Consistency of CODASYL-type Databases. Database Technology. Online 1976.

46. Turing, A.M. On Computable Numbers. Proc. Lond. Math. Soc. II, 42, 1936.

47. Wada, Eiiti. On the Possibility of the Unification of Command and Programming Languages. Command Languages edited Unger C. North-Holland 1975.

48. Weegenaar, H.J. and Wielenga, D.K. Towards a Generalised Command Lanaguage for Job Control. Command Languages edited Unger C. North-Holland 1975.

49. Weller, T. On the User Oriented Semantics of Command Languages. Command Languages edited Unger C. North-Holland, 1975.

50. Appel, K. Reduced Control Language for non-professional users. Command Languages edited Unger C. North-Holland 1975.

51. Brandon, J.P. Job Control Languages MAXIMOP and CAFE. Job Control Languages edited Simpson, D. N.C.C.

52. Simpson, D. editor, Job Control Languages, N.C.C. 1974.

53. Unger, C. editor. Command Languages. North-Holland 1975.