Ellis Horwood · Publishers
COMPUTERS AND THEIR APPLICATIONS

The Dartmouth Time Sharing System

G. M. Bull

Computers and Their Applications
Series Editor:
BRIAN MEEK Director, Computer Unit
Queen Elizabeth College, University of London

THE DARTMOUTH TIME SHARING SYSTEM

The Dartmouth Time Sharing System (DTSS) is important both for its historical position and for its influence on time sharing systems throughout the world; and no less importantly for its ease of use both to specialist and non-specialist users. Few systems have the unique advantage of having been designed and developed from grass roots level twice by the same personnel, and few systems can match the efficiency, elegance of design, and the simplicity of the DTSS.

This book documents the system in detail and provides an overview with an introduction to the main software modules and a brief description of the hardware. The coverage of the user interface gives the reader a "feel" for what the software is implementing, and there is a broad view of the file system, including types of files, file security, file systems maintenance, and file sharing. There is a large section on the operating system software, dealing with the first level virtual machine which implements the file system and runs the hardware. All the major aspects are covered thoroughly and clearly, to the final chapter on systems operation, an area which is often left uncovered in existing literature.

The author, who has worked at Dartmouth in the DTSS writing team, has provided here an elementary yet broad text which will fill a need for students studying operating systems and professionals involved in operating system design.

Readership: Undergraduates and post-graduates studying computer operating systems, and all those working in educational establishments, industry and commerce where the DTSS system is in use.

THE DARTMOUTH TIME-SHARING SYSTEM

THE ELLIS HORWOOD SERIES IN COMPUTERS AND THEIR APPLICATIONS

Series Editor: BRIAN MEEK
Computer Unit, Queen Elizabeth College, University of London

The series aims to provide up-to-date and readable texts on the theory and practice of computing, with particular though not exclusive emphasis on computer applications. Preference is given in planning the series to new or developing areas, or to new approaches in established areas.

The books will usually be at the level of introductory or advanced undergraduate courses. In most cases they will be suitable as course texts, with their use in industrial and commercial fields always kept in mind. Together they will provide a valuable nucleus for a computing science library.

Published and in active publication

THE DARTMOUTH TIME SHARING SYSTEM
G. M. BULL, The Hatfield Polytechnic

THE MICROCHIP AS AN APPROPRIATE TECHNOLOGY
Dr. A. BURNS, The Computing Laboratory, Bradford University

INTERACTIVE COMPUTER GRAPHICS IN SCIENCE TEACHING
Edited by J. McKENZIE, University College, London, L. ELTON, University of Surrey, R. LEWIS, Chelsea College, London.

INTRODUCTORY ALGOL 68 PROGRAMMING
D. F. BRAILSFORD and A. N. WALKER, University of Nottingham.

GUIDE TO GOOD PROGRAMMING PRACTICE
Edited by B. L. MEEK, Queen Elizabeth College, London and P. HEATH, Plymouth Polytechnic.

DYNAMIC REGRESSION: Theory and Algorithms
L. J. SLATER, Department of Applied Engineering, Cambridge University and
H. M. PESARAN, Trinity College, Cambridge.

CLUSTER ANALYSIS ALGORITHMS: For Data Reduction and Classification of Objects
H. SPÄTH, Professor of Mathematics, Oldenburg University.

FOUNDATIONS OF PROGRAMMING WITH PASCAL
LAWRIE MOORE, Birkbeck College, London.

RECURSIVE FUNCTIONS IN COMPUTER SCIENCE
R. PETER, formerly Eotvos Lorand University of Budapest.

SOFTWARE ENGINEERING
K. GEWALD, G. HAAKE and W. PFADLER, Siemens AG, Munich

PROGRAMMING LANGUAGE STANDARDISATION
Edited by B. L. MEEK, Queen Elizabeth College, London and I. D. HILL, Clinical Research Centre, Harrow.

FUNDAMENTALS OF COMPUTER LOGIC
D. HUTCHISON, University of Strathclyde.

SYSTEMS ANALYSIS AND DESIGN FOR COMPUTER APPLICATION
D. MILLINGTON, University of Strathclyde.

ADA: A PROGRAMMER'S CONVERSION COURSE
M. J. STRATFORD-COLLINS, U.S.A.

THE DARTMOUTH TIME-SHARING SYSTEM

GORDON M. BULL, F.B.C.S., M.Sc., Ph.D.
Reader in Computer Science
The Hatfield Polytechnic

ELLIS HORWOOD LIMITED
Publishers · Chichester
Halsted Press: a division of
JOHN WILEY & SONS
New York · Chichester · Brisbane · Toronto

First published in 1980 by

ELLIS HORWOOD LIMITED
Market Cross House, Cooper Street, Chichester, West Sussex, PO19 1EB, England

The publisher's colophon is reproduced from James Gillison's drawing of the ancient Market Cross, Chichester.

Distributors:

Australia, New Zealand, South-east Asia:
Jacaranda-Wiley Ltd., Jacaranda Press,
JOHN WILEY & SONS INC.,
G.P.O. Box 859, Brisbane, Queensland 40001, Australia

Canada:
JOHN WILEY & SONS CANADA LIMITED
22 Worcester Road, Rexdale, Ontario, Canada.

Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester, West Sussex, England.

North and South America and the rest of the world:
Halsted Press: a division of
JOHN WILEY & SONS
605 Third Avenue, New York, N.Y. 10016, U.S.A.

British Library Cataloguing in Publication Data
Bull, Gordon M.
The Dartmouth Time-sharing System. –
(Ellis Horwood series in computers and their applications).
1. Dartmouth Time-sharing System
I. Title
001–6'4404 QA76.53 80–41327

ISBN 0–85312–253–9 (Ellis Horwood Ltd., Publishers)
ISBN 0–470–27082–9 (Halsted Press)

Typeset in Press Roman by Ellis Horwood Ltd.
Printed in Great Britain by R. J. Acford Ltd., Chichester

Table of Contents

Chapter 4 – THE FILE SYSTEM

Chapter 5 – MASTER-MODE OPERATING SYSTEM

Chapter 6 – SLAVE-MODE OPERATING SYSTEM

Preface

In 1964 Dartmouth College in Hanover, New Hampshire, U.S.A., developed its time-sharing system on combined GE-235 and Datanet 30 computers. In 1965 General Electric (U.S.A.) marketed the first commercial time-sharing service using the Dartmouth software (the GE-265, Mark I time-sharing system). This world-wide service was responsible in part for the rapid growth of time-sharing. In 1967 Dartmouth College and General Electric combined to produce time-sharing software for the GE-635 computer. Using Dartmouth's multiple on-line debugging system (MOLDS), designed for time-sharing system development, the GE Mark II time-sharing system and the Dartmouth Time-Sharing System emerged as similar but different products. General Electric went on to market Mark II and eventually Mark III time-sharing systems. The Dartmouth Time-Sharing System (DTSS) became operational in 1969 and is now operating at eleven sites in the U.S.A., Canada and Europe.

Although there are many time-sharing systems operational throughout the world today, DTSS is important for a number of reasons: firstly, for its historical position and its influence on time-sharing; secondly, its major design goal was to be easy to use for non-specialist users. Although this design goal is not unique to DTSS, it can be argued that the DTSS user interface meets this goal better than any other system. Thirdly, few systems have had the benefit of being twice designed from the ground up (and by the same personnel). DTSS was designed after the initial GE-265 system had been operational for three to four years both in-house at Dartmouth and in numerous bureaux operated by GE and others throughout the world. Finally, DTSS is a highly efficient system, whose simplicity and elegance of design is a model worthy of study by system designers. Up until now most of the systems available for study have been single-copy, complex, 'state-of-the-art', or experimental, (or all four) which although important give one little guidance on their applicability in the real world.

In writing this book I was reminded of a quote by Professor David Howarth, one of the designers of the Atlas system, which has remained with me since the day I first read it. He said:

> 'Our problem is that we never do the same thing again. We get a lot of experience on our first simple system, and then when it comes to doing the same thing again with a better designed hardware, with all the tools we know we need, we try and produce something which is ten times more complicated and fall into exactly the same trap. We do not stabilise on something nice and simple and say "let's do it again, but do it very well this time".' (Quoted in *Operating Systems Techniques,* Hoare, C. A. R. and Perrott, R. H., Academic Press, 1972.)

DTSS is the product of doing it again and doing it very well and the outcome is a superb software system worthy of serious study.

The first chapter is a historical overview and a statement of the design goals of the system. Chapter 2 is an overview of the system and serves as an introduction to the main software modules and a brief description of the hardware. Chapter 3 is devoted to a discussion of the user interface. This is covered at this point for two reasons. Firstly, because of the importance of this aspect of the system, and secondly because it gives the reader the 'feel' for what the software is implementing. Chapter 4 covers the file system in detail, since the concept of a file is central to the whole system. All aspects of the file system are covered including types of files, the catalog tree structure, file security, file system maintenance, and file sharing. Chapter 5 is the start of the description of the software system. It deals with the software which provides the first level virtual machine which implements the file system and runs the hardware (the Executive). The structures of the Executive and the Job-Executive Interface are explained in detail. Chapter 6 deals with the major elements of the operations system running beneath the Executive. Chapter 7 deals with the communication subsystem, dealing both with the structure and operation of the communication processor software. This deals with the communication between the main system and the Terminal Communication Controller, LOGIN and Monitors, including SIMON, the Monitor which provides the general use interface. Finally, Chapter 8 looks at the system from an operations point of view, an important aspect ignored by most books on operating systems.

Such a book as this cannot hope to be definative in describing such a large and complex system as DTSS, nor can it hope to be an accurate description of the system forever, since like all major software systems DTSS is not a static entity. Nevertheless, what I hope has been captured is a flavour of the system and an accurate description of its underlying philosophies.

Some of the information in the book exists in various forms in one or more Technical Memoranda, the major source of information on DTSS. However, most of the inner details were revealed as a result of discussions with the staff of the Kiewit Computation Center and of DTSS Incorporated. In particular I wish to thank Stan Dunten, for his invaluable help in writing Chapter 7 and Polly Cole, Alex Colvin, Lou Fernandez, Dick Green, Charlie Kaufman and Carl

Pedersen of DTSS Incorporated for the excellent training course I attended and for numerous subsequent discusions. Many others commented on the manuscript and I record my appreciation here. I give thanks to Bob Hargraves for letting me talk him into this project and for his support throughout, and to Paula Lowery and Lynn Sherburns for the typing of the manuscript on to DTSS and making the numerous corrections.

In 1967, I spent six months at Dartmouth College as part of the team engaged writing DTSS. My contribution was small (or even negative) since that was my first brush with a major software system. It was therefore a great personal pleasure to be able to return after ten years and through this book attempt to make known to a wider audience the details of a superb time-sharing system. I make no apologies if my enthusiasm for DTSS shows through.

Gordon M. Bull
Hatfield Polytechnic
England

CHAPTER 1

Introduction

1.1 HISTORICAL DEVELOPMENT

Computing at Dartmouth College started in 1959 when the college acquired a Bendix LGP-30, a 4K drum machine. Out of the work on various software projects for that machine, Professors Kemeny and Kurtz discovered that some undergraduates could become expert programmers and could design and implement highly sophisticated software systems. Thus when the decision was taken to design and implement a time-sharing system, the bulk of the work was done by undergraduate programmers working in their spare time, with only minimal supervision from faculty members.

Time-sharing was first proposed by Strachey in 1959 [1]. The earliest operational systems were the CTSS system at Massachusetts Institute of Technology [2], and the SDC Q-32 system built by the Systems Development Corporation [3]. A preliminary design for the Dartmouth time-sharing system was prepared by an undergraduate junior in July 1962. Work began in earnest during the summer of 1963 when detailed design of the system and some programming was undertaken prior to the delivery of the hardware in February 1964. The hardware chosen – General Electric 235 and Datanet 30 computers – was standard product line equipment at the time and this was intentional. To quote Kurtz [4]: 'The guiding primary purpose was to design a better interface between the student and the computer, and not to engage in computer research. The decision was therefore made to employ known programming techniques and standard hardware equipment.' (What an understatement! Designing a better user interface was, in my opinion, on of the most significant pieces of computer research ever undertaken, influencing many thousands (millions?) of computer users.) Fig. 1.1 shows the hardware configuration.

On 1st May 1964 at 4 am, the first program in BASIC ran successfully [5], barely one and one half months after delivery of the hardware. By fall of 1964, twenty users could use the system simultaneously and it was used to train 85% of the 1964 freshman class. A terminal was also installed in Hanover High School. By summer of 1965, the system could support forty simultaneous users

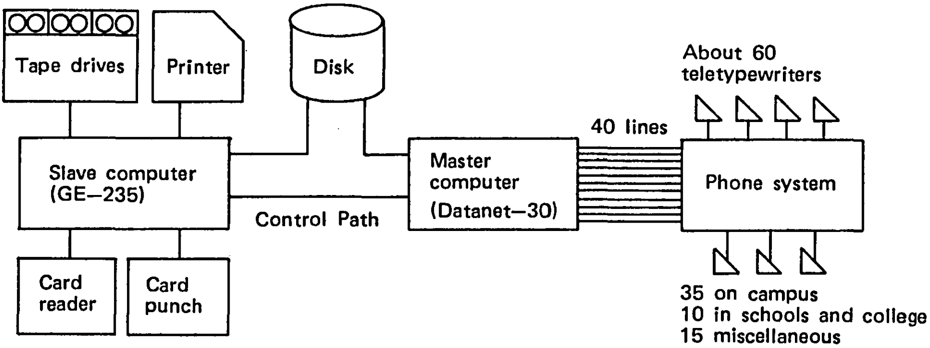


Fig. 1.1 – Hardware and communications for the first DTSS.

and it continued to do so until its departure from Dartmouth in September 1967.

The success of the system at Dartmouth led General Electric to recognise its commercial potential and they renamed the hardware configuration as the GE-265 and marketed the service, later renaming it the Mark I time-sharing system. It can be argued that the early success of the Dartmouth/GE system in the commercial marketplace was directly responsible for the growth of time-sharing throughout the world.

The success of the initial system led to its downfall as more and more people used the system and the need for more computing power was soon felt. As a result of the success of the GE time-sharing business based on the Dartmouth system, an agreement was reached whereby the GE-265 system was replaced by a GE-635 system. This entailed the design of a new software system which would (a) retain the successful, simple user interface, (b) satisfy the demands created by the earlier system, and (c) correct defects found in the first system [6]. The GE-635 system was delivered in November 1966. By October 1967, the GE-635 system was providing a service based on what was known as 'Phase I' software. This was a joint Dartmouth/GE project which GE subsequently marketed as Mark II (and later with the addition of batch and transaction processing as Mark III). The development of Phase I was much aided by a Dartmouth software development system called MOLDS (multiple on-line debugging system), produced early in 1967. In parallel with work on Phase I, Dartmouth embarked in 1967 on the development of 'Phase II'. This project, under the direction of Professor John Kemeny, was undertaken by faculty and students and was the real successor to the GE-265 system. Phase II or the Dartmouth Time-Sharing System, as it is now called, replaced Phase I on 1st April 1969 at Dartmouth. In January 1976 the GE-635 was replaced by a Honeywell 66/40 and with changes to the software went live in February 1976. In order to give a clearer picture of the various stages of development of DTSS, a year-by-year summary of various aspects of the systems is given below:

1962	Initial design for an educational time-sharing system
1963	Program design and coding in preparation for delivery of hardware
1964	GE-235 and Datanet 30 system delivered February 3 user system running in May 20 user system running in September – BASIC only ALGOL added by November
1965	40 user system LISP added
1966	Preparatory work for new system Coding of MOLDS Coding of Phase I (GE Mark II)
1967	Coding of Phase I Design and coding of Phase II (DTSS) Phase I supersedes 265 system in September
1968	ALGOL 60 available First support for graphics – TSP plotters DTSS experimental
1969	1st April – DTSS supersedes Mark II at Dartmouth Background system available Run-off produced – first major word processing system on DTSS Multi-terminal programs
1970	New FORTRAN Compiled BASIC programs saveable
1971	Magnetic tapes in background Second DTSS site at U.S. Naval Academy Subprograms added to BASIC COBOL available under GCOS simulator
1972	DTSS, Inc. established to market software Sublibraries implemented COBOL available under foreground APL SNOBOL 4 Project FIND – first data base system on DTSS Third DTSS site (first commercial DTSS site)
1973	DYNAMO simulation language Type setting system DXPL Telenet 4th and 5th DTSS sites 6000 series executive

1974 716 front end processors
DTSS honoured at Pioneer's Day at the NCC
Perform files
Mail

1975 File archiving to tape
RJE
6th, 7th and 8th DTSS sites

1976 SBASIC
Graphics statements in BASIC
9th, 10th and 11th DTSS sites

1977 New COBOL
PL/I

In order to give some idea of the reliability of the system, statistics from the Dartmouth site are given in Table 1.1. Terminal hours is sum of the hours each terminal was logged on. This number is given, since this is one measure of the usage of the system and it is only through usage that bugs are found. Since DTSS is now running at at least eleven sites, these terminal hours are but a small fraction of the total. The percentage up-time is the percentage that the Dartmouth site was available for use during scheduled time-sharing. The total number of hours is the annual scheduled time-sharing time at Dartmouth in hours.

Table 1.1.

Fiscal year	Terminal hours	Percentage up-time	Scheduled time-sharing hours
1968–69	105 783	98.54	3936
1969–70	180 933	98.62	4385
1970–71	242 042	98.60	5321
1971–72	273 842	98.69	5659
1972–73	309 775	99.01	6258
1973–74	361 284	98.68	6237
1974–75	362 725	97.80	6118
1975–76	386 138	98.97	6002
1976–77†	380 000	98.92	6000

† Extrapolated from 11 months information.

Thus is summary we see that the Dartmouth site has offered a service foı over 50 000 hours over nine years; that the reliability of DTSS averages out at 98.65% available, or put the other way, in the last nine years it has only been down for an average of seventy-five hours per year. During this time, over two and one half million terminal hours of service have been supplied to its users at Dartmouth and elsewhere. These figures are even more remarkable when one considers that Dartmouth is the development site for the software; such sites are traditionally fairly unreliable.

1.2 DTSS IS NOT JUST BASIC

One of the most widely held misconceptions about DTSS is that it only supports BASIC. Of course BASIC played a central role in the development of time-sharing at Dartmouth. It was developed in 1963/64 and was the only language available at first on the GE-265 system. However, by the fall of 1964, ALGOL 60 was available and this was followed by LISP and FORTRAN.

Most implementations of BASIC have been based on the Dartmouth version at the time and have included the command language and the overall user interface. Thus most BASIC systems around today distinguish commands from program text by the absence or presence of a leading line number on each line of text and sort the program text by line numbers, deleting lines with only a line number and replacing lines with duplicate line numbers. This is taken to be 'BASIC' by most users and implementors. In reality, it is the DTSS user interface (provided by SIMON) plus the language BASIC. That is, DTSS provides the above user interface for all its languages, not just BASIC but since this interface has come to be regarded by non-DTSS users as BASIC, the misunderstanding about DTSS supporting only BASIC is easy to understand. The current version of DTSS supports many different languages including BASIC, ALGOL 60, FORTRAN 76, COBOL 74, APL, DXPL, DYNAMO, GMAP, LISP, MIX, PL/I and SNOBOL.

1.3 CURRENT USES

The eleven current sites may be grouped into three broad classes:

(a) **Time-sharing bureax.** Five sites offer a general purpose commercial time-sharing service. Like the Mark I system on the GE-265 where more than fifty copies were in operation, DTSS is highly competitive in this market.

(b) **In-house time-sharing.** Four sites offer a general purpose in-house time-sharing service.

(c) **Academic service.** Two sites use DTSS primarily for educational purposes – teaching, reasearch and college administration.

1.4 REFERENCES

[1] Strachey, C. (June 1969), 'Time Sharing in Large Fast Computers', Proc. International Conference on Information Processing, UNESCO, Paris, France, pp. 336-341.

[2] Corbato, F. J., Merwin-Daggett, M., and Daley, R. C. (May 1962), 'An Experimental Time-sharing System', Proc. AFIPS Fall Joint Computer Conf., pp. 335-344.

[3] Schwartz, J. I., Coffman, E. G., and Weissman, C. (April 1964), 'A General Purpose Time-sharing System', Proc. AFIPS Spring Joint Computer Conf., pp. 397-411.

[4] Kurtz, T. E. (1971), 'Dartmouth College, Time-Sharing System', Encyclopedia of Library and Information Science, pp. 434-439, Marcel Dekker.

[5] Kemeny, J. G., and Kurtz, T. E. (October 1968), 'Dartmouth Time-sharing', Science, pp. 223-228.

[6] Hargraves, R. F., and Stephenson, A. G. (1969), 'Design consideration for an Educational Time-sharing System', Proc. AFIPS Spring Joint Computer Conf., pp. 657-664.

CHAPTER 2

Overview of the System

2.1 HARDWARE ARCHITECTURE [1]

DTSS was originally written to run on GE-600 series machines, using Datanet 30 communication processors. Subsequent versions use Honeywell 716 communication processors and Honeywell 6000 series and series 60 level 66 mainframes. In terms of the changes in the software this represents a rewrite of the software in the communication front-end processors and a modification to the Executive to deal with the handling of peripherals in the mainframe. Many of the compilers and other subsystems have been modified to utilise the Extended Instruction Set available on the level 66 machines. Fig. 2.1 shows the system architecture of the level 66, together with the communication front-end.

Although it does not make a great deal of sense to consider the absolute minimum hardware required to support DTSS, there are certain key elements of the hardware which are necessary. At least 128K of memory is required in the mainframe along with one tape and at least 24K of 716 memory. Such a system might typically support thirty-two simultaneous users. In terms of maximal systems, the software will support up to seven processors, 16M words of main memory, and multiple 716 front-end processors. A large, multiple processor system is shown in Fig. 2.2.

The hardware components that make up the central system are the central processor (CPU), system control unit (SCU), memory unit, and the input/output multiplexor (IOM). The central processor performs all computations for the system. The system control unit regulates communications between components and services demands upon memory. The input/output multiplexor provides the required level of input/output control between memory and the peripheral and communication subsystems.

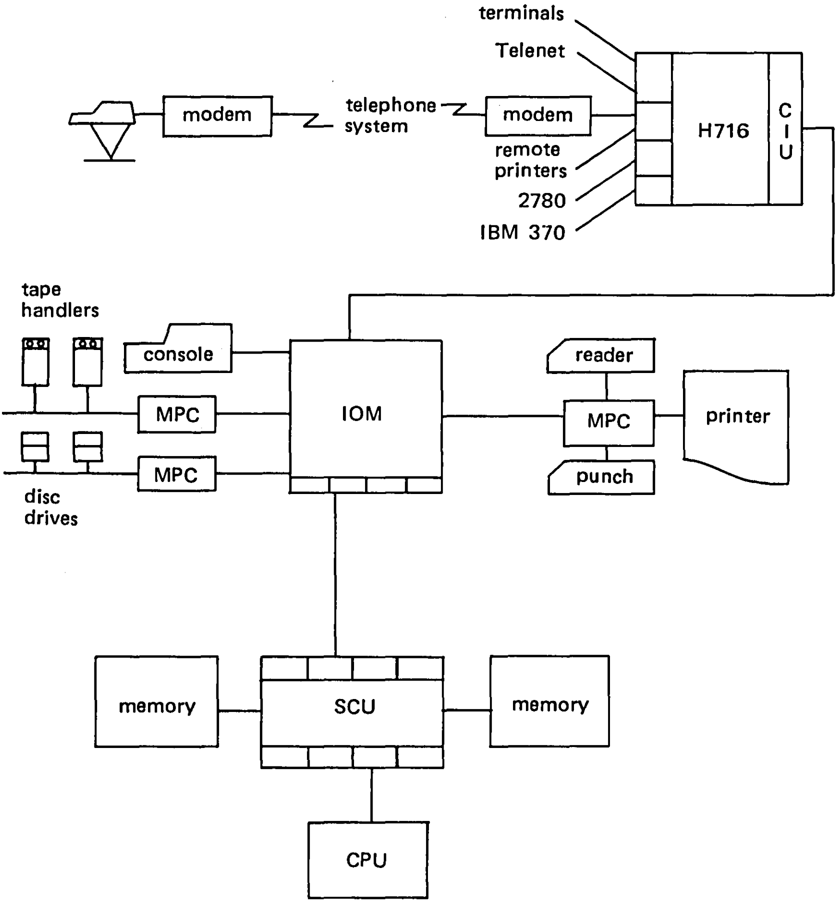


Fig. 2.1 – Honeywell series 60 level 66 architecture.

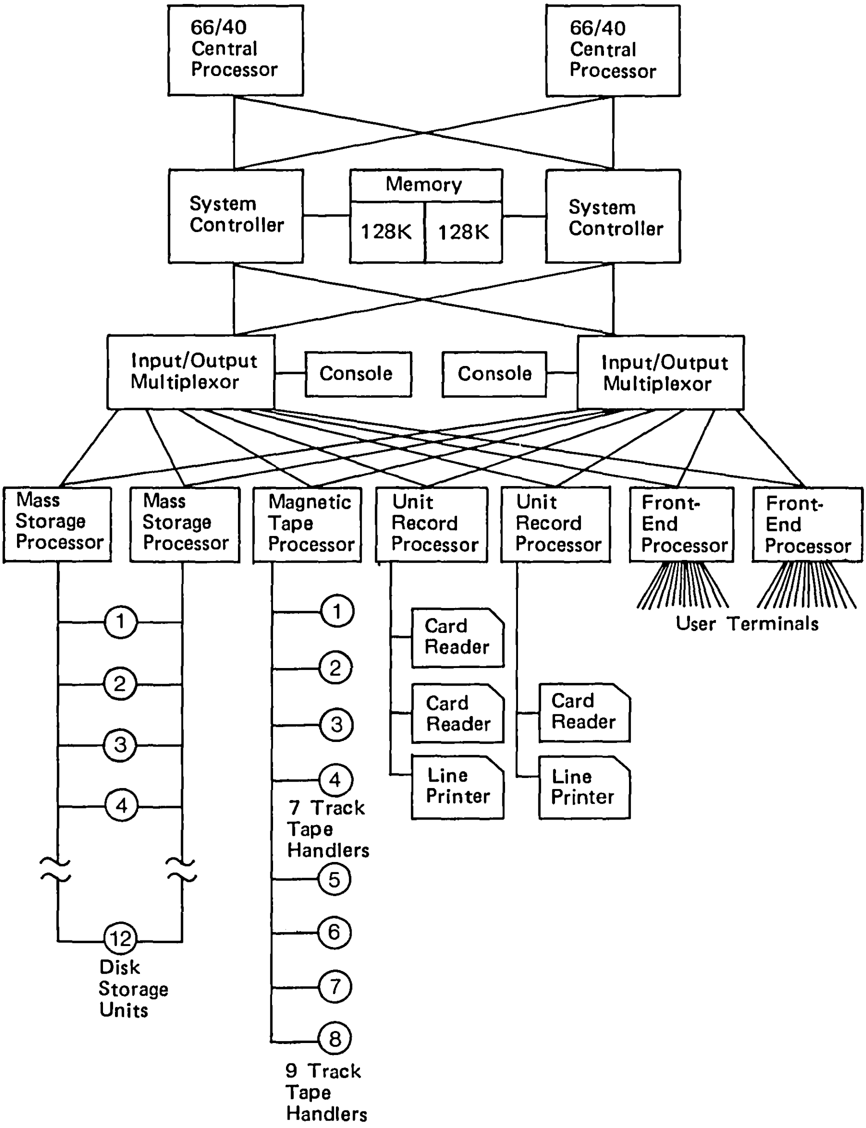


Fig. 2.2 – Large DTSS system to support 200 users.

2.1.1 The processor

The Honeywell level 66 is a 36-bit, word-orientated machine. It has a range of programmable registers including a double length accumulator, eight 18-bit index registers, and exponent register (for floating point operations), an indicator register, and 8 address registers.

The central processor consists of a control unit, an operations unit, and a decimal unit. The operations unit contains the logic to execute binary arithmetic and logical operations. The control unit is the interface between the operations unit and the SCU. It also performs instruction fetching, address preparation, memory protection, data fetching and storing, and overall timing. The decimal unit provides additional registers for address modification, decimal arithmetic, character and bit string operation and formatting. The three units operate with relative independence and varying degrees of overlap.

Most models in the level 66 series have a built-in, high-speed processor cache memory that increases the overall rate of instruction execution. A 2048-word buffer stores copies of information brought into cache on demand by the processor. Subsequent accesses to cache are at a higher speed than main memory access. Cache is controlled by a first-in, first-out replacement algorithm.

The processor operates in one of two modes – master mode or slave mode.

In slave mode, memory references are restricted to an assigned, contiguous address space and are relative to a value held in a register called the base address register. Program execution is limited to a subset of the total processor instruction repertoire such that input/output, setting the base address register, setting the timer and other special registers cannot be executed in slave mode. This mode of operation is used to run user programs and less privileged parts of the DTSS operating system.

In master mode, the entire memory may be accessed, input/output may be initiated, the processor control registers may be set, and the entire instruction set of the processor can be executed. This mode of operation is reserved solely for one section of the DTSS operating system (the Executive).

As noted above, in slave mode, memory addressing is restricted to an assigned continuous address space in memory. Such a facility is necessary if one process is not to interfere with another, totally independent process, either inadvertently or by design. This simple security is provided by the base address register. This 18-bit register contains two pieces of information.

The lower half of the base address register contains the number of 1024-word blocks assigned to the slave process. The upper half of the base address register contains the base of the address space, and is always a multiple of 1024. All addresses generated by a process are in the range zero through n, where n is the largest address in the process' address space. Each time a process running in slave mode generates an address, it is checked to ensure it lies within the address space by comparing it with the lower half of the base register. If it does not lie within the address space, a fault is generated and the process is halted.

If it is within the space, the upper half of the base address register is added to the generated address to give the absolute memory address, and the memory access is performed using this. This is shown diagramatically in Fig. 2.3.

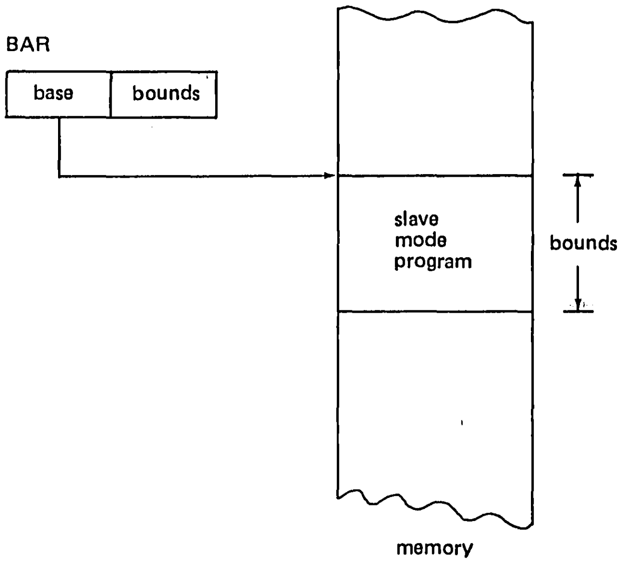


Fig. 2.3 – Base address register.

Processes are not bound to any specific locations in memory since the base address register contains the only specific reference to an actual memory location, and its value may be changed by the Executive as and when a process is relocated in memory.

2.1.2 System controller

The system control unit is the principal interface between all central system components. It provides complete system interrupt control, which regulates communication between components, and services all demands on memory under priority control. The SCU switches control signals, addresses, and data in and out of the memory units, while monitoring data and control paths for accuracy. It also provides memory protection. Processors and IOMs are connected to, and also access memory through, the SCU.

An SCU has eight ports into which are plugged processors and input/output multiplexor. Each system controller can have two independently accessible memory units which can both provide access during a single access period.

System controllers can be operated so that they fetch pairs of words rather than single words in a single access, thus further increasing throughout.

Each input/output multiplexor generates interrupts to indicate such events as completion of requested I/O operations. An interrupt request sets a program interrupt cell in the system control unit. Each IOM sets the interrupt cell it has been assigned. For each interrupt cell, two consecutive locations are provided in memory, establishing an interrupt vector for each IOM.

The processor acknowledges an interrupt request when the interrupt cell is in the SCU. The SCU passes to the CPU the highest priority instruction pair from the interrupt vector, thus initiating the interrupt service.

Each memory unit has separate addressing and memory registers. Two words plus error detection and correction bits are accessed in each memory cycle. The error dectection and correction bits allow single bit errors to be corrected.

2.1.3 Input/output multiplexor (IOM)

The IOM co-ordinates all input/output operations between the SCU, peripheral subsystems, and the 716 communications processor. Data transfers between peripheral devices and memory are handled by the IOM, in parallel with the operation of the CPU. All peripheral operations are initiated by the Executive (in master mode) and controlled by information stored in fixed locations in memory, referred to as IOM mailboxes.

The IOM consists of the IOM central and a variable number of channels. The IOM central controls access to storage for each of the channels and can perform one storage access cycle at a time through the appropriate SCU. Thus the IOM central is time-shared by a number of channels operating concurrently. The IOM provides complete memory protection for all IOM data transfers, simultaneous operation of numerous peripheral subsystems, communication with multiple independent SCUs, scratchpad storage for control words, and a data transfer capability of six million characters/second.

Data channels transfer to all external peripheral devices information controlling the instruction, data and status flow between the IOM and the peripheral control unit. Data channel words are held in the IOM central scratchpad storage to provide higher speed data transfers. The peripheral subsystem interface (PSI) channel allows multiple logical channels for concurrent multiple unit operation. There are a variety of special channels for such things as bootload and hardware diagnostics.

The IOM uses the information passed to it by the Executive in the mailboxes (called data control words – DCWs) to protect memory on all I/O transfers. The Executive passes the contents of the job's base address register to the IOM which uses it to perform all the necessary checks and relocations.

2.1.4 Peripheral processors

The various peripherals are interfaced to the IOM through, and controlled by, a microprogrammed peripheral controller (MPC). There are three basic processors (microprograms) – mass storage processors to control data transfers to and from moving head discs, magnetic tape processors to control tapes, and unit record processors to control printers, card readers and card punches.

2.1.5 Terminal communications controller

All user terminals are interfaced to the system through a Honeywell 716 communications processor.

The 716 is interfaced to the IOM through a special interface called the computer interface unit (CIU) or coupler. The 716 is a 16-bit machine and the function of the coupler is to map 36-bit words into two, 16-bit words and vice versa. This is shown in Fig. 2.4.

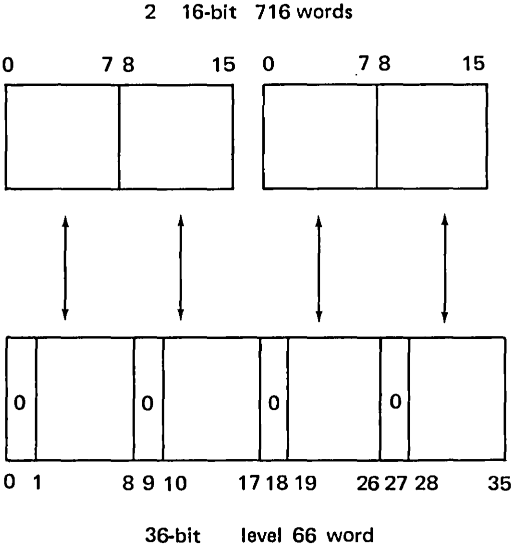


Fig. 2.4 – 716 to level 66 mapping.

The 716 is equipped with real-time clock and watch-dog timer. The universal multiline controller (UMLC) will handle up to sixty-four synchonous and asynchronous lines. The low-speed multiline controller (LSMLC) will handle up to one hundred and twenty-eight asynchronous lines.

2.2 SOFTWARE ARCHITECTURE

The software that makes up the DTSS operating system is both modular and hierarchical. A single module, called the Executive, runs in master mode; all others run in slave mode. The hierarchical structure of the operating system is illustrated in Fig. 2.5. The major modules are briefly described below and a memory map of these modules is shown in Fig. 2.6.

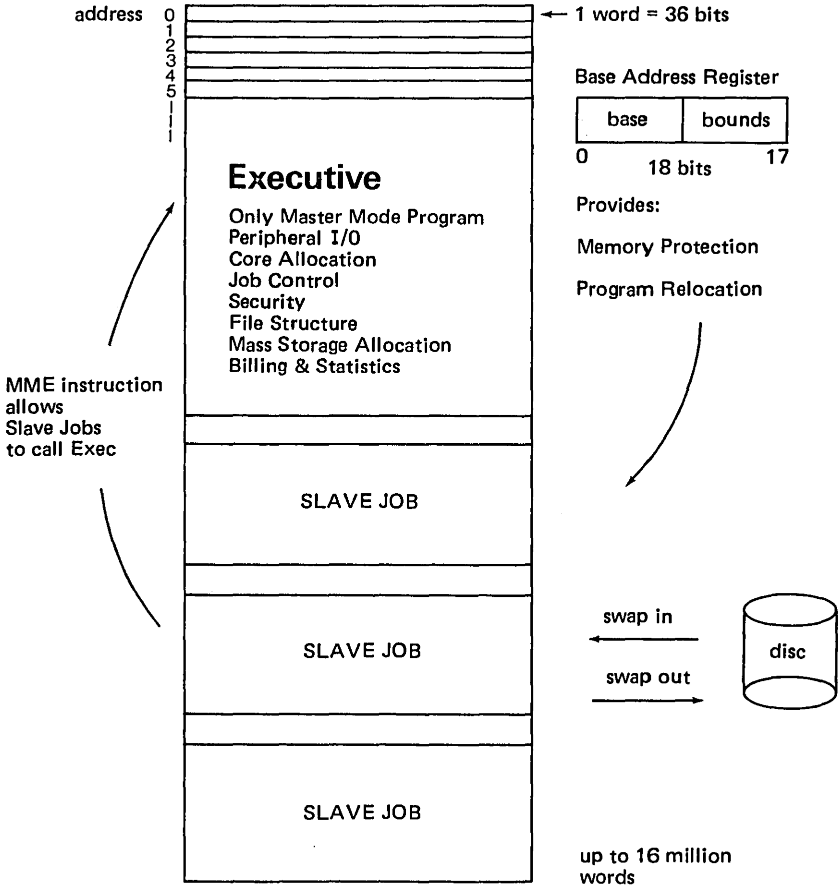


Fig. 2.5 – DTSS job tree.

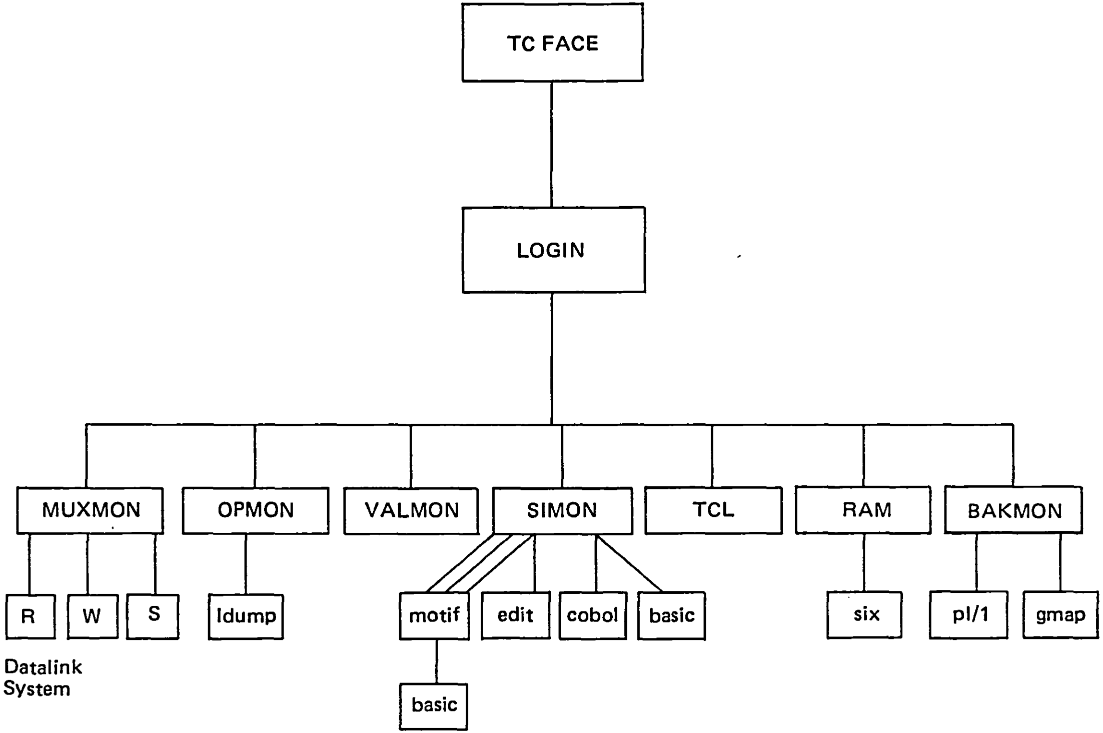


Fig. 2.6 – DTSS memory map.

2.2.1 The Executive

The Executive is the only part of the operating system which runs in master mode. The function of the Executive is to map the physical machine into a virtual machine for each slave mode job. The virtual machine provided has the following characteristics (throughout this text 'slave job' or 'job' will be used rather than 'process' since the entire DTSS literature uses these terms):

(a) A single, slave-mode-only processor.
(b) A single, contiguous area of memory, starting at location zero and up to a maximum of 192K words long.
(c) Access to a number of specific files (files are discussed in Chapter 4).
(d) Access to a clock from which certain timing information can be obtained.
(e) The ability to overlap processing and input/output to many different files.
(f) The ability to create new processes.

In implementing many copies of this virtual machine on a physical machine with fewer total resources than those required by the sum total of all the virtual machines, the Executive carries out the following actions, in a manner that makes them invisible to slave jobs:

(a) Allocates each processor as it becomes available to a slave job which is waiting for it, setting the appropriate value in the base address register and the maximum time for which that job may retain the processor in a timer register.
(b) Swaps jobs between memory and disc and vice versa, allocating memory space and disc space accordingly and calculating the appropriate value for the base address register.
(c) Handles all communication with the IOM, to drive the peripherals.

The Executive is ultimately responsible for system security including such things as controlling file access and inhibiting job interference. Some of the information required for system statistics and for billing is also collected by the Executive.

The mechanism used by a slave job to communicate with the Executive, for example to request input/output, to ask for more memory, to ask for today's date, is a single slave mode instruction called the master mode entry (MME) instruction. The various requests are indicated by different values in the address field of the instruction word and parameters are passed in the accumulators and index registers. The Executive, which is about 35K words long resides from absolute word zero of memory onwards. It is shown diagramatically in Fig. 2.7.

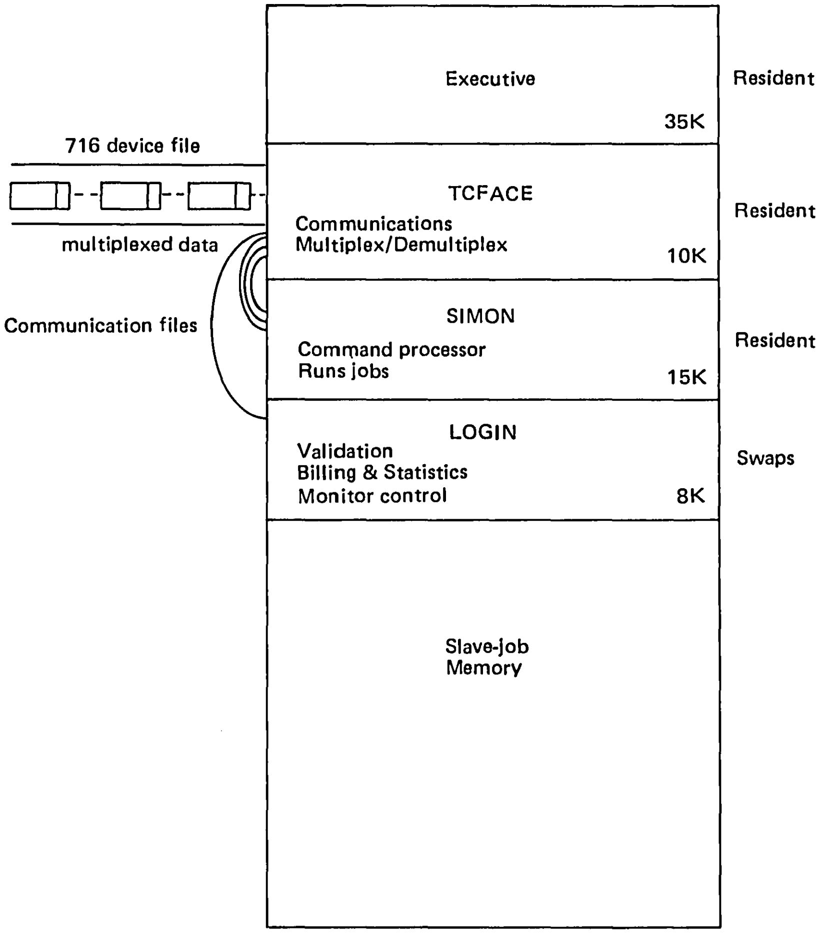


Fig. 2.7 – The DTSS Executive.

2.2.2 Terminal communication controller interface (TCFACE)

The primary function of TCFACE is to interface slave jobs to users at terminals. In doing so it multiplexes and demultiplexes the data stream to and from the communication processors and the various jobs.

TCFACE treats each terminal as a file and as a result most of its work is concerned with reading and writing files. Since TCFACE is only at intermediary in passing data to and from the terminal and the associated job, a special type of file, called a communication file (see section 5.3) is used. TCFACE holds the master end of the communication file for each active terminal on the system.

The communication between TCFACE and the communications processors is carried out in a cyclic manner, blocks of data that are ready for transmission being sent at regular intervals.

TCFACE controls to some extent the action of the software running in the communications processor by sending messages to it to indicate the logical state of each terminal under its control.

2.2.3 LOGIN

This module runs under TCFACE. Its main functions are:

(a) User validation – printing the prompt for the user number and checking that the user number given by the user is valid and corresponds to the given password. Each user of the system has a validation record which contains the password associated with the user's user number. LOGIN reads this validation record to make the check.

(b) When a user first signs on under a newly validated user number, LOGIN creates a user catalog for that number, since no catalog is created when a user is validated (see sections 6.2.1 and 6.3.4).

(c) TCFACE passes LOGIN an end of the communication file representing the terminal. LOGIN passes this end of the communication file together the user's catalog and job permissions (obtained from the validation record) to the monitor which the user is going to run under. This is most often SIMON (see section 2.2.4).

(d) LOGIN records the connect time of each user on the system and is also responsible for collecting other statistics and billing information for each user on the system. This information is collected by the monitor which the user is running under and also by the Executive. Periodically LOGIN asks each monitor and the Executive for all their statistics. LOGIN then writes this information to a file for later processing by other programs.

(e) Since there are many monitors running beneath LOGIN (see Fig. 2.5) and some of them wish to communicate with one another (such as SIMON and BAKMON when a user submits a job from his terminal for later processing by BAKMON), it is simpler to have a single communication path between each monitor and LOGIN than a large number of such paths between each of the monitors. Thus one of LOGIN's functions is to provide this intermonitor communication facility. It uses this same facility to request statistics and billing information form the monitors.

LOGIN, although part of the operating system, is used so infrequently that it is swapped in and out of memory.

2.2.4 Simple monitor (SIMON)

Simple monitor (SIMON) is the monitor that serves as the primary user interface for the Dartmouth Time-Sharing System. After a user has signed on to the system and has been identified, the communication file representing his terminal is passed to SIMON. The user may then start a 'conversation' with the system.

By giving commands to SIMON the user can initiate any of the tasks that the system will perform for the general user. From SIMON's point of view (but not from the user's, who sees a uniform interface) these tasks fall into two categories. First, there are the simple tasks which it can perform unaided; these tasks include elementary file operations such as building, altering, and saving files in the user's catalog. Second, there are more complex tasks which SIMON initiates by running other modules; these tasks include compilation and execution of source language programs, text editing, submission of background jobs, interterminal communication, and so forth.

SIMON serves many users simultaneously (the way it accomplishes this is discussed in Chapter 6).

2.3 REFERENCES

[1] 'Honeywell Series 60 Level 66 Summary Description' (1975), Honeywell Information Systems, document DC64.

[2] Pedersen, L. C. (February 1976), 'How DTSS Works Slide Show', DTSS System Memorandum 256.

[3] Green, R. L. (December 1977), 'Technical Overview of DTSS', DTSS Incorporated System Programmer's Training Course Notes.

CHAPTER 3

The User Interface

One of, if not the most important, aspects of DTSS is the cleanness and simplicity of its user interface. This was one of the primary design goals of the original Dartmouth system and has remained central to the system through its many evolutions and additions. One of the tributes, both to the original design and to subsequent enforcement, is that even after more than ten years of growth the user interface remains clean and simple to use.

This interface is provided by the principal monitor on DTSS – SIMON. Other, more specialised interfaces are provided by other Monitors (see Chapter 6).

Most users of BASIC around the world are in fact familiar with the SIMON interface. Most Basic implementation embed the language in an environment which includes a filing system, a command language, and the concept of a current file. This environment is most often a faithful copy of the original Dartmouth user interface, or a modification (for the worse) of it.

3.1 THE CURRENT FILE

All information on the system is organised into files. One of the most important files is the current file. Many actions on the system which involve a file, use the current file by default. In its simplest form, a current file consists of lines of ASCII text, each line prefixed with a unique line number which serves to order the line within the file. A user may amend the current file by entering line-numbered lines of text. These lines are sorted into the file with the following conventions. If no line previously existed with the line number just entered, the line is entered, in line-number order in the file. If a line with the same line number existed, the newly entered line replaces the old line. If the line just entered consisted solely of a line number followed only by carriage-return, any line with that line number is deleted from the file. Thus without the aid of an explicit editor, the user may build and amend the current file. The current file, or any other file, may contain other than line-numbered text.

3.2 ELEMENTARY COMMANDS [1]

Any line typed by the user when in command mode, that does not start with a line number is taken by the system to be a command requesting some action. In the command descriptions which follow, words in the command language will appear in capital letters. Parameters to be supplied by the user will appear in lowercase, surrounded by angle brackets. All commands, unless noted otherwise, may be abbreviated to the first three characters. When the command has completed (this or any command), the systems outputs the word:

```
    READY
```

to the terminal to signify the user may enter either another command or line-numbered text. This is called the ready-state. all commands are interpreted by SIMON. The user interface is so designed that a user only needs to know a small subset in order to do most things on the system. The philosophy of not requiring the user to know about things not related to the job in hand pervades the whole user interface as well as being incorporated into BASIC. Thus throughout, one is supplied with natural defaults. Only if these defaults are not what is required must the user learn how to override them. The user may enter a command to SIMON whenever the terminal is in ready-state. (The terminal is in ready-state at other times than just following the word READY and before the next command, but this description will suffice for the moment.) The end of a command is signalled by entering carriage-return.

3.2.1 The LIST command

The user may list the contents of the current file on the terminal using the command:

```
    LIST
```

Line-numbered files are listed in line-printer order, with a heading consisting of the name of the current file and the date and time.

3.2.2 The NEW command

The user may create a new, empty current file with a given name with the NEW command, which takes the form:

```
    NEW <filename>
```

The name of the current file becomes <filename>.

3.2.3 The SAVE and REPLACE commands

The user may save a copy of the current file by typing the command:

```
    SAVE
```

which will save a copy of the current file with its current name in the user's catalog. Had a file of the same name already existed in the user's catalog, then attempting to save it in this way would have resulted in the message:

```
    SAVE DENIED--DUPLICATE FILENAME "filename"; TYPE 'EXPLAIN SAVREP'
```

This is to protect the user from inadvertantly overwriting a file which happened to have the same name as the current file. To replace a saved file with a copy of the current file, the user may use the command:

```
    REPLACE
```

Both SAVE and REPLACE may take filenames as parameters, overriding the default which is the name of the current file.

3.2.4 The OLD command

The user may make a saved file the current file, using the command:

```
    OLD <filename>
```

Thus, for example, if a user had previously saved a file (see section 3.2.3) called ABC, to obtain a copy of this file as his current file, he would type:

```
    OLD ABC
```

His current file would be called ABC and the saved version of ABC would be unaffected. On the other hand to obtain a copy of a text formatting program held in the public library, he would type:

```
    OLD RUNOFF***
```

where the three asterisks informs the system that the file should be looked for in the public library.

3.2.5 The UNSAVE command

A user may delete a saved file using the command:

```
    UNSAVE <filename>
```

This command has no effect on the current file. The named file is deleted from the user's calalog. If the filename is omitted, the name of the current file is assumed.

3.2.6 The RUN and SYSTEM commands

If the current file contains text representing the source text of a program, the user may have if compiled and executed using the single command:

```
    RUN
```

The compiler used depends on a number of factors. The default compiler is BASIC but it may be changed to some other language using the command:

```
SYSTEM <systemname>
```

for example:

```
SYS COBOL
```

If the current file was obtained using the OLD command, then part of the information held about the file is the system (language) to be used in compiling the file and this information automatically overrides the default. The system information for a saved file is generated at the time the file is saved and is normally the system current at that time. The RUN command does not change the current file. If the compiler detects any errors it does not go on to the execution stage.

With the above commands: LIST, NEW, OLD, SAVE, REPLACE, UNSAVE, RUN, and SYSTEM, the user can accomplish a great deal of work. Indeed, beginning and unsophisticated users need know no other commands, and if they use only BASIC they do not need to know about the existence of other languages on the system.

Apart from the above commands, there are four special characters/keys which are part of the user interface. Rubout deletes the previous character typed on the current line and echoes a backslash; each rubout deletes one character. Control-X deletes the entire line. When the system is outputting to the terminal, control-X will delete about 250 characters of the output without stopping the outputting job. Control-R will cause the current line to be output. The break-key stops any running job and returns the terminal to the ready-state.

3.3 OTHER COMMANDS [2], [3]

There are over seventy commands available under SIMON, giving sophisticated users the power they require. The user may enter several commands on the same line by beginning the line with a delimiter, separating each command from the next by the delimiter with which the line was begun. Such a line of commands is called a command sequence. The delimiter may be any character other than a letter or a digit. Thus the two sequences (a) and (b) below accomplish the same thing. (The user input is underlined to identify it.)

(a)

```
OLD XYZ
READY
RUN
XYZ              date          time

[input/output associated with the program run]

READY
```

(b) `/OLD XYZ/RUN`

```
XYZ              date             time

[input/output associated with the program run]

2) READY
```

SIMON begins obeying the commands, in the order in which they occur, when the carriage-return at the end of the command sequence is entered. After obeying a command sequence, SIMON prints out the number of commands obeyed followed by READY. If all commands in the sequence were successful, the number is equal to the number of commands in the sequence. If an error occurs, or if the user stopped the execution of the command sequence, the number preceeding the READY message tells how many commands were successfully executed. If an error occurs during execution of a command sequence, an error message is printed together with a number indicating the position of the offending command in the list, and the sequence is terminated at that point. For example, the sequence:

```
/OLD ABC/SAVE/OLD PQR/SAVE
```

assuming ABC was in the user's catalog, would produce the message:

```
2) DUPLICATE FILENAME "ABC"; TYPE 'EXPLAIN SAVREP'
2) SAVE DENIED
```

and stop with ABC as the current file. The sequence:

```
/OLD ABC/REP XYZ/UNS ABX/OLD PQR/LIST
```

assuming ABC and XYZ exist in the user's catalog but that ABX does not would give rise to the message:

```
3) "ABX" NOT SAVED
```

and ABC would be the current file.

In order to give a flavour of the range of facilities available, a few of the more widely used commands, over and above those described in the previous section, are outlined below.

3.3.1 File commands

In this category come LIST, NEW, OLD, REPLACE, SAVE, and UNSAVE. The BUILD command allows the user to type in a file without line numbers. All text is appended directly to the end of the current file. The user exists from build-mode by typing two carriage-returns one after another. The RENAME command allows the user to change the name of the current file. The SCRATCH command allows the user to delete the entire contents of the current file without changing its name.

3.3.2 Program execution commands [4]

In this category comes RUN. The COMPILE command allows the user to compile the current file, replacing the source text in the current file by the resulting object code. The program is not executed and the name of the current file is changed to 'OBJECT'. The object code can subsequently be saved. Object code files can be executed with the RUN command or the EXECUTE command (see below). In either case, the object code file is recognised as such and no compilation phase takes place.

The EXECUTE command allows the user to run a program (compile and execute or just execute depending on the contents of the named file) which is not the current file; this program may use the current file as data. The format of the command is:

```
EXECUTE <filename>
```

The LINK command allows the user to initiate a named, multi-terminal conference. Other users may join the conference by issuing a JOIN command with the same name. An important use of this at Dartmouth is the ability to JOIN CONSULT which puts a user in touch with the programming consultant on duty.

3.3.3 Editing commands [5], [6]

These commands allow the user to use one of the editors on the system. The command name is the name of the editor. In each case the user supplies commands to the editor, either as part of the edit command itself or following a prompt from the editor. Thus if this subsection (3.3.3) were the current file and the user wished to find the line with the text '(3.3.3)' the following sequences (a) and (b) would both cause the line containing '(3.3.3)' to be printed:

```
(a) TEXT LOCATE (3.3.3)

(b) QED
    (/(3.3.3)/
```

where in example (b) the '(' at the beginning of the second line is the prompt from QED telling the user to enter a QED command string. Thus in (a), the edit command forms part of the SIMON command, whilst in (b), the SIMON command calls the editor, which issues a prompt for the edit function required.

3.3.4 Environment commands

In this category comes SYSTEM. The ENTER command allows the user to change his current catalog. When a user signs on, the current catalog is the one associated with the user number; commands such as OLD and SAVE pertain to files in the current catalog. The ENTER command may be used to change to another catalog such as a subcatalog within the user's catalog. To change catalogs may require certain privileges and knowledge of any password associated with the catalog. The user may return to his user number catalog with the command:

```
ENTER *MYCAT
```

3.3.5 Terminal mode commands

The BRIEF command allows the user to suppress the printing of such things as system resource usage messages, filename, date and time headers from listings and runs, and to obtain abbreviated versions of system messages. The word READY is replaced by an *. This mode is not for novice users. The counter command is NBRIEF.

3.3.6 Information commands [7]

The CATALOG command allows the user to obtain a print-out of information describing files saved in a catalog. The simple version of the command gives an alphabetical list of the names of the saved files in the current catalog, together with a header which identifies the catalog and its length. A variety of options are available to obtain additional information and to modify the format of the output.

3.3.7 Miscellaneous commands

The MAIL command allows the user to send mail to another user on the system who has indicated his willingness to receive mail. When the recipient signs on he will be informed of the fact that he has mail. The CALCULATE command provides the user with an on-line adding machine which evaluates arithmetic expressions and prints the value.

The use of some, but not all the above commands is illustrated in section 3.8.

3.4 ERROR MESSAGES

Since all users make mistakes, an important aspect of the user interface is the error reporting system. The philosophy of the system is to provide the user with diagnostic information which is meaningful to him. This is achieved by using words that are as free from jargon as possible. Where words which may be unfamiliar to the user are included in the message, information about how to obtain fuller explanation is given as part of the message. Thus of the fifty or so messages generated as a result of command errors (that is, those error messages generated by SIMON), thirty explicitly tell the user what to do or tell him how to obtain more information pertaining to the problem. For example, following a LINK command, the following error message may be given:

```
KEYWORD "keyword" IS ALREADY IN USE; CHOOSE ANOTHER KEYWORD
```

indicating that someone else has already intitiated a conference using the same keyword (conference name). The error is identified and the remedy given. A similar example, given earlier, is when a SAVE command attempts to save a file with the same name as one already saved:

```
"SAVE DENIED--DUPLICATE FILENAME "filename"; TYPE 'EXPLAIN SAVREP'
```

The EXPLAIN command is dealt with in the next section, but once again the error is clearly identified and in this case further explanatory information is available if the user does not understand. Of those which do not give further information, a typical message is:

```
"filename" IS NOT SAVED
```

which tells the user that the filename he typed with a REPLACE, UNSAVE, SCRATCH or OLD command is not saved.

3.5 EXPLAIN FILES

There are over 750 000 characters of information available to the user through the EXPLAIN command. The form of the command is:

```
EXPLAIN <list of topics>
```

Thus, for example, if the user were to type:

```
EXPLAIN NOTAVAIL
```

the following output would be given:

```
"NOTAVAIL"

When you receive the message

"filename" IS NOT AVAILABLE; TYPE 'EXPLAIN NOTAVAIL'

you have typed a command of the form

OLD *<user number>:<filename>

and the file specified by "<filename>" is unavailable for one of
the following reasons:

1. The file is not saved in the specified user catalog.

2. The file is saved in the specified user catalog, but is not
saved with group, public or fetch permission.  For further
information on permissions for saved files, type

     EXPLAIN ACCESS
```

```
3. The file is saved in the specified user catalog with group,
public or fetch permission available provided the proper password
is specified.  You either did not specify a password in your OLD
command, or you specified an incorrect password.

4. The user who owns the file is currently modifying it via a
program or via a REPLACE, SCRATCH, or UNSAVE command.

5. The specified user catalog contains more than twice the
amount of file storage that it should.

READY
```

The information within the explain system is structured, thus for example, the CATALOG command has many parameters and it would be inappropriate to get all the information by typing 'EXPLAIN CATALOG' since there is over 35 000 characters of information on the catalog command. The user is guided down the tree if he doesn't know the explain system. Thus, a user might start with:

```
EXPLAIN CATALOG
```

This is a fairly lengthy explanation, but within it are listed the various options on the CATALOG command and a note which reads:

```
For a more complete explanation of any option type

EXPLAIN CATALOG <option>
```

Thus the user might then type:

```
EXP CAT CURRENT
```

wishing to have the current option on the CATALOG command explained. This would produce the following output.

```
"CURRENT"

The CURRENT option causes information that would
normally be printed at the user's terminal to be
written into the user's current file.  Since this
information replaces whatever was previously in the
current file and the current file name is changed to
.RESULT., the user should be careful to preserve his
current file via a SAVE or REPLACE command before
specifying the CURRENT option.
```

A similar set of explanations exist for the editors and their commands. Thus typing:

```
EXP EDIT LOCATE
```

will explain the LOCATE command of the Dartmouth Editor. The total explain database is implemented as a catalog tree, thus, exploiting the file structure on DTSS. EXPLAIN files exist for most of the major applications systems also. Thus, information about the FIND database system may be obtained by typing:

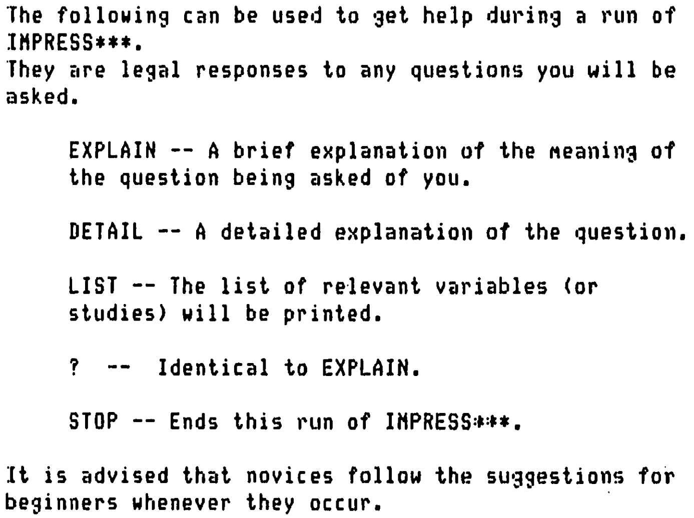
```
EXPLAIN <list of topics>
```

to FIND. Thus:

```
OLD FIND***
READY
RUN

FIND           13 DEC 1977    11:07

FIND HERE!
USAGE DATA IS BEING RECORDED

? EXP DATABASE

DATABASE

A database is simply a collection of information stored
in a specialised format which is acceptable to FIND.
The demonstration database DEMO is available to all
DTSS users and may be accessed by typing

     BASE DEMO

when FIND requests a command.

DONE
```

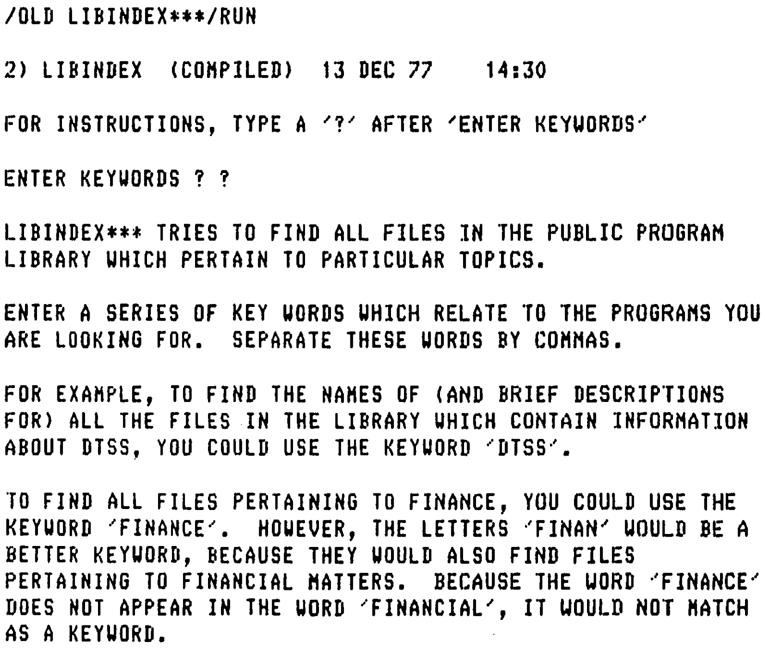
IMPRESS, the social science data retrieval and analysis package, on the other hand, takes a more interactive approach to providing information, as shown below:

```
/OLD IMPRESS***/RUN

2) IMPRESS   (COMPILED)   30 JAN 78    11:11

XIMPRESS, last modified 26JAN78.  ***Privacy warning***
Please report any difficulty to project IMPRESS
staff.
Thanks.

New IMPNEWS*** (12/14/77)

Do you want user's instructions? YES
```



```
The following can be used to get help during a run of
IMPRESS***.
They are legal responses to any questions you will be
asked.

     EXPLAIN -- A brief explanation of the meaning of
     the question being asked of you.

     DETAIL -- A detailed explanation of the question.

     LIST -- The list of relevant variables (or
     studies) will be printed.

     ?  --  Identical to EXPLAIN.

     STOP -- Ends this run of IMPRESS***.

It is advised that novices follow the suggestions for
beginners whenever they occur.
```

Another type of information made available to users is the location of programs or packages relevant to their needs. A run of the public library program called LIBINDEX*** is shown below:



```
/OLD LIBINDEX***/RUN

2) LIBINDEX  (COMPILED)  13 DEC 77    14:30

FOR INSTRUCTIONS, TYPE A '?' AFTER 'ENTER KEYWORDS'

ENTER KEYWORDS ? ?

LIBINDEX*** TRIES TO FIND ALL FILES IN THE PUBLIC PROGRAM
LIBRARY WHICH PERTAIN TO PARTICULAR TOPICS.

ENTER A SERIES OF KEY WORDS WHICH RELATE TO THE PROGRAMS YOU
ARE LOOKING FOR.  SEPARATE THESE WORDS BY COMMAS.

FOR EXAMPLE, TO FIND THE NAMES OF (AND BRIEF DESCRIPTIONS
FOR) ALL THE FILES IN THE LIBRARY WHICH CONTAIN INFORMATION
ABOUT DTSS, YOU COULD USE THE KEYWORD 'DTSS'.

TO FIND ALL FILES PERTAINING TO FINANCE, YOU COULD USE THE
KEYWORD 'FINANCE'.  HOWEVER, THE LETTERS 'FINAN' WOULD BE A
BETTER KEYWORD, BECAUSE THEY WOULD ALSO FIND FILES
PERTAINING TO FINANCIAL MATTERS.  BECAUSE THE WORD 'FINANCE'
DOES NOT APPEAR IN THE WORD 'FINANCIAL', IT WOULD NOT MATCH
AS A KEYWORD.
```

```
THEREFORE, IN GENERAL, THE LESS SPECIFIC THE KEYWORD, THE
MORE ENTRIES WILL MATCH IT.  HOWEVER, THIS CAN BE CARRIED TO
EXTREMES; THE USE OF THE SINGLE LETTER 'A' AS A KEYWORD WILL
PRODUCE REFERENCES TO NEARLY EVERY FILE IN THE LIBRARY!

TO STOP THIS PROGRAM, PUSH THE 'RETURN KEY' WHEN 'ENTER
KEYWORDS' IS TYPED.

ENTER KEYWORDS ?
```

An example of the use of this program is given in the section of examples at the end of this chapter. A similar program called TMINDEX*** performs the same function on the DTSS Technical Memoranda. The file DOCUMENT*** lists the various documents available pertaining to the system or programs on the system. Various other information files are available. CCNEWS*** gives news about changes and improvements to the system. 'NEWS' files in various libraries give news relevant to that library (for example, PLOTIB***:NEWS). The file SUBLIBS*** gives a list of sublibraries together with any recent additions. The file NEWFILE*** records new additions to the program library.

Program files in the library contain information about themselves, informing the user exactly what the program does, and how to use it.

Thus we see many levels of information available easily and quickly for the user, ensuring that he is able to recover from errors, obtain information about all aspects of the system and make full use of the library and documentation.

3.6 FILE SYSTEM AND LIBRARIES

A file is a block of information stored on DTSS. Filenames are from one to eight characters long, composed of letters, digits, periods, and hyphens. A catalog is a file which contains structured information about other files. Associated with every user number there is a catalog; each file saved whilst using a given user number becomes part of that number's catalog (although this can be overriden). We shall see (Chapter 4) that the catalogs are organised in a tree structure. All commands which refer to the filing system using simple filenames use the current catalog as the place to look for the named files. Most users sign on under a user number and that catalog becomes and remains their current catalog. Only a few users change their current catalog using the ENTER command. Thus the commands such as OLD, SAVE, REPLACE, UNSAVE when issued with a simple filename look for that filename in the current catalog.

An important concept is the distinction between a current file and a saved file: these are always two separate entities. The current file, created by the command NEW or OLD, existes only temporarily as the file presently being used. Using a saved file, the current file is a temporary copy of the saved file. As information is typed into the current file, that file is the only copy of the new information (until saved or replaced). As soon an another file is called up or the user signs-off the system, the current file disappears; it is no longer in existence.

A saved file is a file preserved in a catalog. When SAVE or REPLACE is typed, a saved file is created which is a copy of the current file as it exists at the moment. Once saved, the preserved version of the file remains unchanged until a command to REPLACE it or to UNSAVE it is given. (The command SCRATCH, as explained below, and certain programs which write information into a saved file can also affect the contents of a saved file.) A current file called via the command OLD is not the saved file itself but merely a temporary copy of it; the saved file is still preserved and unchanged. Thus any changes typed into a current file do not affect the saved version of the file in any way until REPLACE is typed. Then and only then is the stored file replaced by the revised current file. The editors work only on the current file and do not change the saved version.

The command UNSAVE operates only on the saved version of a file. Typing UNSAVE destroys the saved version of what is then the current file but does not affect the current file itself. As long as the current file is not changed with a NEW or OLD command, the temporary, current version of the file still exists. The command form UNSAVE <filename>, specifying a file other than the current one, allows the user to destroy the named stored file without affecting the current file (or its stored version if any) in any way.

The SCRATCH command operates on the contents of either the current or saved file depending on which of its two forms is used. Typing SCRATCH erases the contents of the current file only; the saved version (if one exists) is not affected. In the form SCRATCH <filename>, the command destroys the contents of the saved file specified; the file remains saved in the catalog but its length is set to zero. The current file is unaffected in this case.

File stored in public libraries are saved in such a way that users may make access to them without having the ability to change the saved version in the library. The user can make programs such as LIBINDEX*** his current file and can then run it. Indeed, the user can save library files in his own catalog if this proves to be useful. Programs in the DTSS library may be in source code or object code. For those held in object code the source versions are in the catalog SCODELIB*** and may be copied from there, modified if necessary and then saved in the users catalog.

The use of sublibraries allows large libraries to be structured in a meaningful way, making the indexing of files easier and subsequently making it easier for users to find and use them.

3.7 PERFORM FILES

A further dimension is given to the user interface through the use of perform files.

Perform files allow the creation of specialised user interfaces by allowing a series of commands, and other terminal input to be replaced by a single PERFORM command naming the perform file. The following description of the perform system is that obtained by typing EXP PERFORM.

```
PERFORM <filename>
PERFORM <filename>;<argument>;<argument>;...
```

The PERFORM command causes a PERFORM file to be executed. A PERFORM file simulates the typing that you would do at the terminal to complete a task. PERFORM files can contain commands to Simple Monitor, line-numbered lines, input data, and special commands to the PERFORM system.

When a PERFORM command is typed, Simple Monitor begins executing the commands in the PERFORM file. Simple Monitor commands in the file are printed on the terminal as they are executed, along with any terminal output they cause (such as 'READY', time and input/output messages). When the last command in the PERFORM file is completed Simple Monitor prints the total time and an extra 'READY'.

If there is a file named .SIGNON. saved in a user catalog, it is PERFORMed automatically whenever the user signs on.

NUMBERED LINES AND THE CURRENT FILE

The contents of the current file may change as the commands in the PERFORM file are executed. Any numbered lines in the PERFORM file will affect the current file when the PERFORM file is executed, just as they would affect the current file if they had been typed directly at the terminal. Thus a command to Simple Monitor which is accidentally typed with a line number will not be recognised as a command, but will modify the current file.

COMMANDS TO THE PERFORM SYSTEM

A PERFORM file can also contain commands to the PERFORM system. These commands must begin with $. Possible commands are:

```
$REMARK (the rest of the line is ignored)
$PRINT (the rest of the line is printed on the terminal)
$OUTPUT <treename> (terminal output will be diverted to the
specified file)
$NO OUTPUT (terminal output will be suppressed)
```

$OUTPUT with no <tree name> can be used to turn the output back on, after a $NO OUTPUT. $OUTPUT *TTY will redirect output to the terminal after it has been diverted to a file.

Commands to the PERFORM system are not printed on the terminal during the PERFORM, although the message is the $PRINT command is printed. Commands may be abbreviated to their first three letters.

Any dollar sign in the PERFORM file that does not designate a command to the PERFORM system or an argument (see below) must be preceded by another dollar sign.

INPUT FROM THE TERMINAL

If a PERFORM file runs a program which requests input from the terminal, a question mark will be printed on the terminal at the appropriate time and input can be entered as usual.

If output has been diverted from the terminal with a $OUTPUT command, or suppressed by a $NO OUTPUT command, and input is requested from the terminal by a program, PERFORM will print the message ***SUPPLY INPUT on the terminal. A question mark will then be printed, at which time input can be entered from the terminal as usual.

Input which a program requests from the terminal can alternatively be entered from the PERFORM file. Lines in the file which follow a RUN command and begin with a question mark will be intercepted as inout to the program that is being run. A sample PERFORM file which contains such an input line follows:

```
OLD ADDUP

RUN
? 5286.5, 3572.43
```

The input line will be printed when the file is PERFORMed.

USING BUILD

If the PERFORM file contains a BUILD command, BUILD mode is entered. Input to BUILD may be entered from the terminal or from the PERFORM file, as in program input. BUILD can be terminated by a line in the file containing only a question mark.

PASSING ARGUMENTS TO THE PERFORM FILE

The PERFORM command can pass arguments to the PERFORM file. Arguments are separated from <filename> and from each other by semicolons. If an argument contains a dollar sign, it must be preceded by another dollar sign to show it is literal.

In the PERFORM file a digit preceded by a dollar sign indicates that an argument is expected; $1 is the first argument in the command, $2 is the second, and so on to $9. Identical results will be obtained by the following two procedures; the one on the right uses an argument:

```
PERFORM command:             PERFORM command:
   PERFORM RUNPROG              PERFORM RUNPROG; ADDUP

PERFORM file RUNPROG         PERFORM file RUNPROG
   OLD ADDUP                    OLD $1
   RUN                          RUN
```

THE CURRENT FILE AS THE PERFORM FILE

If no filename is specified in the PERFORM command, the current file will be assumed. You should use the current file as a PERFORM file with caution, since the contents are likely to be destroyed.

NESTING PERFORM COMMANDS

The PERFORM command can appear in PERFORM files; it is is the last line in the PERFORM file, a chain (much as a CHAIN in BASIC) is produced. Also, PERFORMs can be nested; this nesting can be about fifteen levels deep (until PERFORM runs out of space).

3.8 EXAMPLES

3.8.1 Use of library programs

Suppose a user has written an SBASIC program and wishes to have its structure displayed through indentation. (The content of the program is unimportant.) Knowing that such a program exists on the library he uses LIBINDEX*** to find it and then uses it. Thus, his program might be:

```
OLD XYZ
READY

LIST
XYZ             19 DEC 77   16:22

100 'Program to replace consecutive lines
110 'with form-feeds in them by a single line
120 'with a form-feed in it.
130
140 'vector for change statement
150 DIM V(200)
160
165 'open a scratch file
170 FILE #2: "*"
180 MARGIN #2: 132
190
195 'prompt for a file name
200 PRINT "FILE  NAME";
210 INPUT F$
220 FILE #1: F$
230
240 'P notes if previous line was form-feed
250 LET P = 0
260
270 DO WHILE MORE #1
280 LINPUT #1: N$     'get next line of file
290 CHANGE N$ TO V
```

```
300 IF V(0)>0 AND V(1)=ASC(FF)   'check for form feed
310 THEN
320 IF P=0
330 THEN
340 LET P=1   'form-feed found
350 PRINT #2: N$  'this is first one so copy it
360 CONTINUE
370 ELSE
380 LET P=0   'unset marker
390 PRINT #2: N$    'and copy line
400 CONTINUE
410 LOOP
420
430 'transfer back to original file
440
450 RESET #2
460 SCRATCH #1
470 DO WHILE MORE #2
480 LINPUT #2: A$
490 PRINT #1: A$
500 LOOP
510
520 END

READY
```

The user then has to call up LIBINDEX*** to find the indenting program:

```
OLD LIBINDEX***
READY
RUN

LIBINDEX (COMPILED)   19 DEC 77     16:23

FOR INSTRUCTIONS TYPE A '?' AFTER 'ENTER KEYWORDS'

ENTER KEYWORDS ? INDENT

"INDENT"

BKINDENT*** EMPHASISES THE STRUCTURE OF BACKGROUND PROGRAMS
FINDENT*** ALLIGNS AND INDENTS INSTRUCTIONS IN FORTRAN
SOURCE CODE
INDENT*** BRINGS OUT STRUCTURAL ELEMENTS OF A BASIC PROGRAM
SBINDENT*** BRINGS OUT THE STRUCTURE OF SBASIC PROGRAMS BY
INDENTATION

ENTER KEYWORDS ? STOP

READY
```

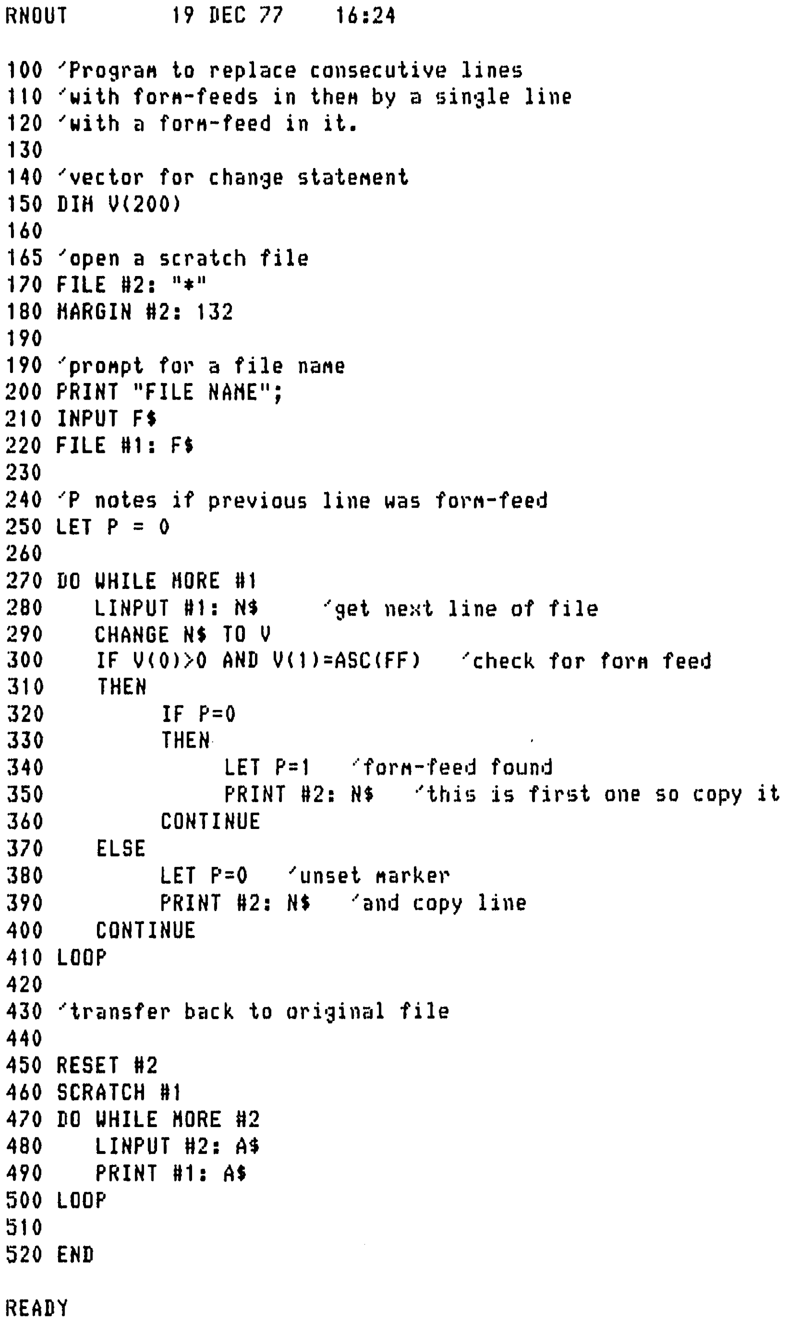
```
/OLD SBINDENT***/RUN
***SYSTEM CHANGED TO SBASIC

SBINDENT  (COMPILED)    19 DEC 77    16:23

Source file, output file--?  XYZ, RNOUT

Scratch output file RNOUT before running SBINDENT***.

STOP
READY
```

Not knowing what scratch means, the user asks for an explanation from the system:

```
EXP SCRATCH

"SCRATCH"

The SCRATCH command erases the current version of your file.
After you type SCRATCH, both your current file and alter
file contain no information.  Saved files are not affected
by this command and the current file name is not changed.
Another form of the SCRATCH command, which should be used
with caution is:

     SCRATCH <filename>,<password>

This form of the SCRATCH command is used to discard the
contents of a file saved in your catalog; the file remains
saved but its length is set to zero.  The current file is
not affected.  If a password  is required in order to allow
the saved file to be scratched and the password is not
supplied, a request for it will be made.

READY
```

The user now understands he must scratch RNOUT as requested and proceeds:

```
SCR RNOUT
READY

RUN

SBINDENT (COMPILED)     19 DEC 77   16:24

Source file, output file--? XYZ,RNOUT
READY

/OLD RNOUT/LIS
```



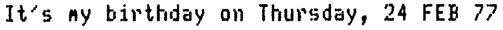
```
RNOUT        19 DEC 77    16:24

100 'Program to replace consecutive lines
110 'with form-feeds in them by a single line
120 'with a form-feed in it.
130
140 'vector for change statement
150 DIM V(200)
160
165 'open a scratch file
170 FILE #2: "*"
180 MARGIN #2: 132
190
190 'prompt for a file name
200 PRINT "FILE NAME";
210 INPUT F$
220 FILE #1: F$
230
240 'P notes if previous line was form-feed
250 LET P = 0
260
270 DO WHILE MORE #1
280    LINPUT #1: N$     'get next line of file
290    CHANGE N$ TO V
300    IF V(0)>0 AND V(1)=ASC(FF)   'check for form feed
310    THEN
320         IF P=0
330         THEN
340              LET P=1   'form-feed found
350              PRINT #2: N$   'this is first one so copy it
360         CONTINUE
370    ELSE
380         LET P=0   'unset marker
390         PRINT #2: N$   'and copy line
400    CONTINUE
410 LOOP
420
430 'transfer back to original file
440
450 RESET #2
460 SCRATCH #1
470 DO WHILE MORE #2
480    LINPUT #2: A$
490    PRINT #1: A$
500 LOOP
510
520 END

READY
```

3.8.2 Use of library subprogram

Suppose a user wishes to write a date reminder program by processing a file whose records consist of three fields – the first and second are four-digit fields representing dates with the first two digits denoting the month and the second two the day of the month, the third field is a free format text string. The processing consists of checking to see if today's date lies between the first and second date of each record and if so printing the text field together with the second date field, reformatted. Thus a record might contain:

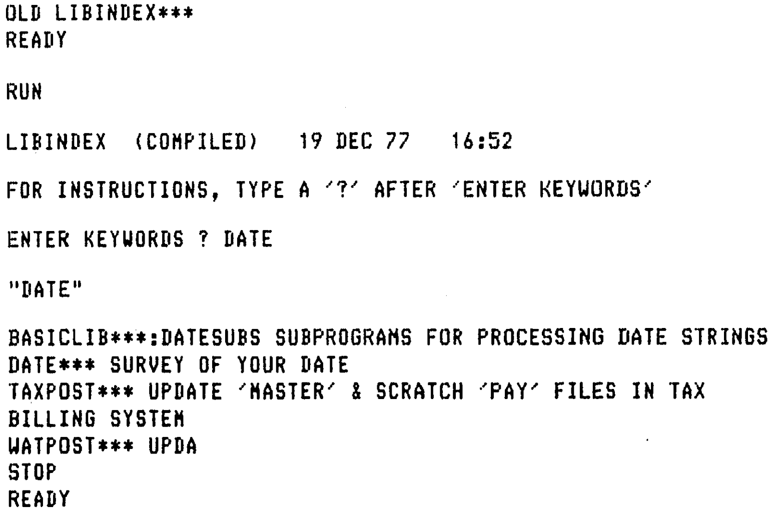
0217, 0224, my birthday

and if the program were run between the 17th and 24th February 1977 the output would be:



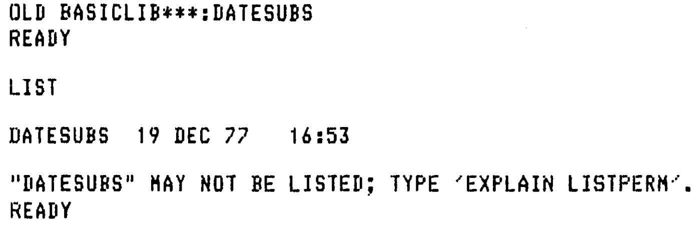
```
It's my birthday on Thursday, 24 FEB 77
```

Before starting the user turns to LIBINDEX*** to help locate any data processing routines.



```
OLD LIBINDEX***
READY

RUN

LIBINDEX  (COMPILED)    19 DEC 77    16:52

FOR INSTRUCTIONS, TYPE A '?' AFTER 'ENTER KEYWORDS'

ENTER KEYWORDS ? DATE

"DATE"

BASICLIB***:DATESUBS SUBPROGRAMS FOR PROCESSING DATE STRINGS
DATE*** SURVEY OF YOUR DATE
TAXPOST*** UPDATE 'MASTER' & SCRATCH 'PAY' FILES IN TAX
BILLING SYSTEM
WATPOST*** UPDA
STOP
READY
```

Stopping the output when he realises that first entry is the one he is looking for. The user then calls up the file and attempts to list it.



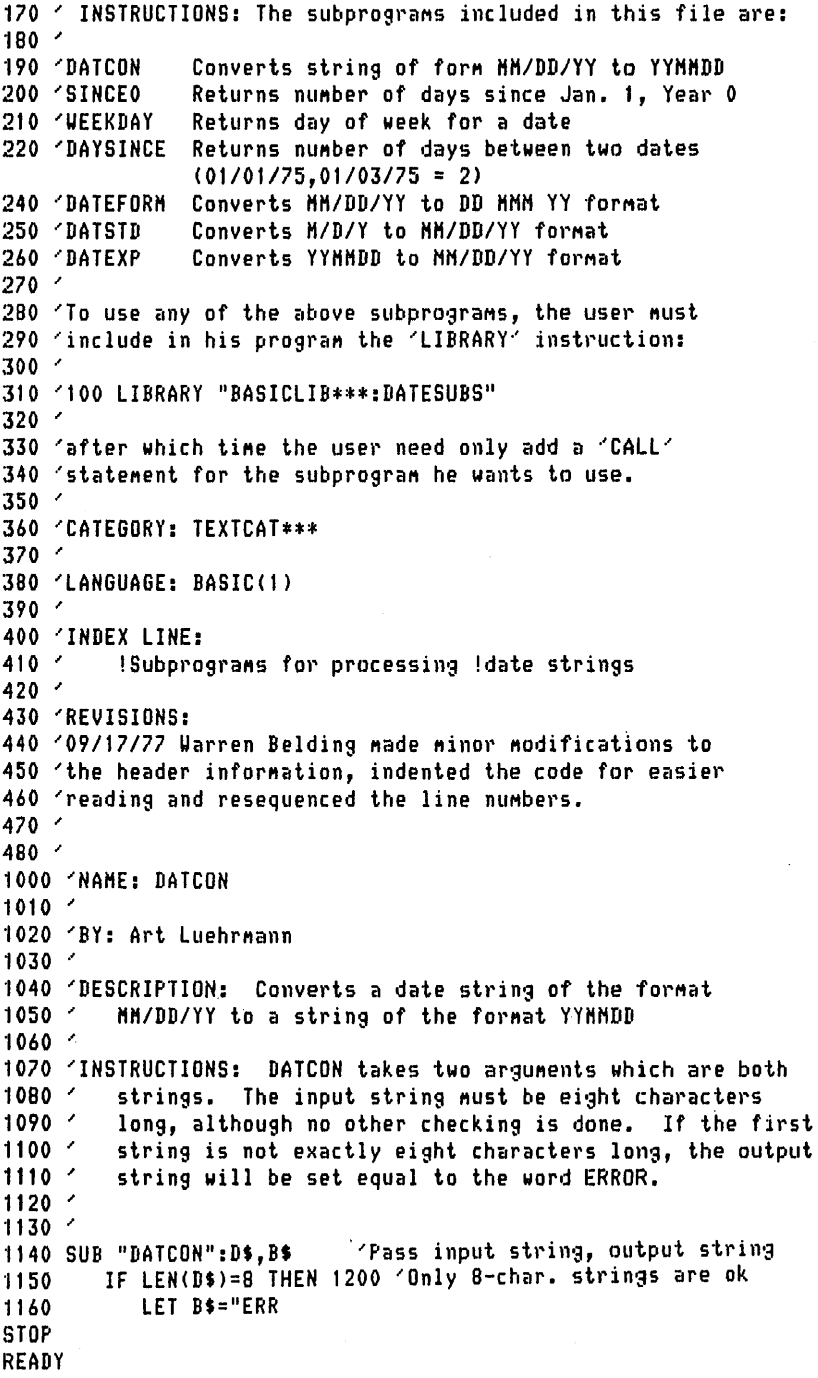
```
OLD BASICLIB***:DATESUBS
READY

LIST

DATESUBS  19 DEC 77   16:53

"DATESUBS" MAY NOT BE LISTED; TYPE 'EXPLAIN LISTPERM'.
READY
```

Not knowing why it is not listable, he does as suggested. Note that even misspelling the word works: Explain allows abbreviations, tells you about possible misspellings, and logs all misspellings so system administrators can decide what explanations should be added.

```
EXP LISTPRM

"LISTPERM"

To list a file, LIST access permission must be available on
the file.  As you referenced it in your OLD command, LIST
permission was not granted.  (You may have forgotten to
supply a password in your OLD command.)  For a complete
explanation of LIST permission, type

     EXPLAIN ACCESS

Also note that it is not possible to LIST a compiled file.
If your file is compiled, and you called it up from the main
program library (you typed something like OLD PROGRAM***) or
from a Kiewit-maintained sublibrary (you typed something
like OLD SUBLIB***:PROGRAM), you may list the non-compiled
version of the file, which is saved in sublibrary
SCODELIB***. For example, to obtain the listable source
code for SUGGEST***, type

     OLD SCODELIB***:SUGGEST

and when the computer types READY, type

LIST.

READY
```

So he gets a copy from SCODELIB*** and lists that:

```
OLD SCODELIB***:DATESUBS
READY

LIST

DATESUBS    19 DEC 77   16:54

100 ' NAME: BASICLIB***:DATESUBS
110 '
120 'BY: Pearson, Smith, Loomis, and Luermann
130 '
140 'DESCRIPTION: A BASIC library file of subprograms for
150 '    handling strings which represent dates.
160 '
```



```
170 ' INSTRUCTIONS: The subprograms included in this file are:
180 '
190 'DATCON     Converts string of form MM/DD/YY to YYMMDD
200 'SINCE0     Returns number of days since Jan. 1, Year 0
210 'WEEKDAY    Returns day of week for a date
220 'DAYSINCE   Returns number of days between two dates
                (01/01/75,01/03/75 = 2)
240 'DATEFORM   Converts MM/DD/YY to DD MMM YY format
250 'DATSTD     Converts M/D/Y to MM/DD/YY format
260 'DATEXP     Converts YYMMDD to MM/DD/YY format
270 '
280 'To use any of the above subprograms, the user must
290 'include in his program the 'LIBRARY' instruction:
300 '
310 '100 LIBRARY "BASICLIB***:DATESUBS"
320 '
330 'after which time the user need only add a 'CALL'
340 'statement for the subprogram he wants to use.
350 '
360 'CATEGORY: TEXTCAT***
370 '
380 'LANGUAGE: BASIC(1)
390 '
400 'INDEX LINE:
410 '    !Subprograms for processing !date strings
420 '
430 'REVISIONS:
440 '09/17/77 Warren Belding made minor modifications to
450 'the header information, indented the code for easier
460 'reading and resequenced the line numbers.
470 '
480 '
1000 'NAME: DATCON
1010 '
1020 'BY: Art Luehrmann
1030 '
1040 'DESCRIPTION:  Converts a date string of the format
1050 '   MM/DD/YY to a string of the format YYMMDD
1060 '
1070 'INSTRUCTIONS:  DATCON takes two arguments which are both
1080 '   strings.  The input string must be eight characters
1090 '   long, although no other checking is done.  If the first
1100 '   string is not exactly eight characters long, the output
1110 '   string will be set equal to the word ERROR.
1120 '
1130 '
1140 SUB "DATCON":D$,B$     'Pass input string, output string
1150    IF LEN(D$)=8 THEN 1200 'Only 8-char. strings are ok
1160       LET B$="ERR
STOP
READY
```

The user realises the output may be a bit long so stops the listing and uses the editor to find the highest line number in the file (the EDIT LIST command lists the current file in descending line-number order).

(Note the format of the comments at the beginning of the library file which is standard for all library files.)

```
EDI LIS

3620 SUBEND
3610    C
STOP
READY
```

He decides it would be better to get a listing on the local line-printer, so he creates a background job file, saves it and submits it with the BACK command for background processing:

```
NEW REM1
READY

10 PRINT SCODELIB***:DATESUBS
20    USING REM1
30 END
SAVE
READY

BAC
***JOB ACCEPTED
READY
```

From the information he determines he needs to use DATCON, DATEXP, DATEFORM, and WEEKDAY and writes the following program, using SBINDENT*** to format it:

```
NEW DATES
READY

10 file #1: "dates"            'list of dates to check
15 library "basiclib***:datesubs"
16 file #2: "datesl"
18 input #2: t$                'get date last used
20 let d$ = dat$               'get todays date
22 if t$<>d$                   'skip if used today
24 then
30    call "datcon": d$,f$     'convert to YYMMDD form
40    let d1$ = seg$(f$,3,6)   'get rid if YY
45
50    do while more #1
```

```
60         input #1: a$,b$,c$        'day-7,day,message
70         if a$<=d1$ and b$>=d1$ 'check for todays date in range
80         then perform convert date
90              print "It's ';c$;" on ";d2$;", ";e$
100        continue
110     loop
112
114     scratch #2
114     print #2: d$                  'write todays date
117
118 continue
119
120 define convert date
130
140     let k$ = seg(d$,7,8) & b$ 'put year on actual date
150     call "datexp": k$,m$      'convert YYMMDD to MM/DD/YY
160     call "dateform": m$,e$    'convert MM/DD/YY to DD MM YY
165     call "weekday": m$,d2$    'get day of week of actual date
170
180 defend
190
200 end
```

3.8.3 Command errors and the EXPLAIN command

Below are listed a few examples of user errors and the information available to recover from them:

(a) Trying to save a file already saved:

```
OLD XYZ
READY

SAV
SAVE DENIED--DUPLICATE FILENAME "XYZ"; TYPE 'EXPLAIN SAVREP'.
READY

EXP SAVREP

"SAVREP"

There is a file saved permanently under your user number having
the same name as that of your current file.  If you want to
replace this saved file with your current file, type "REPLACE".
Alternatively, you may change the name of your current file by
typing "RENAME" and then save it with this new name by typing
"SAVE".
```

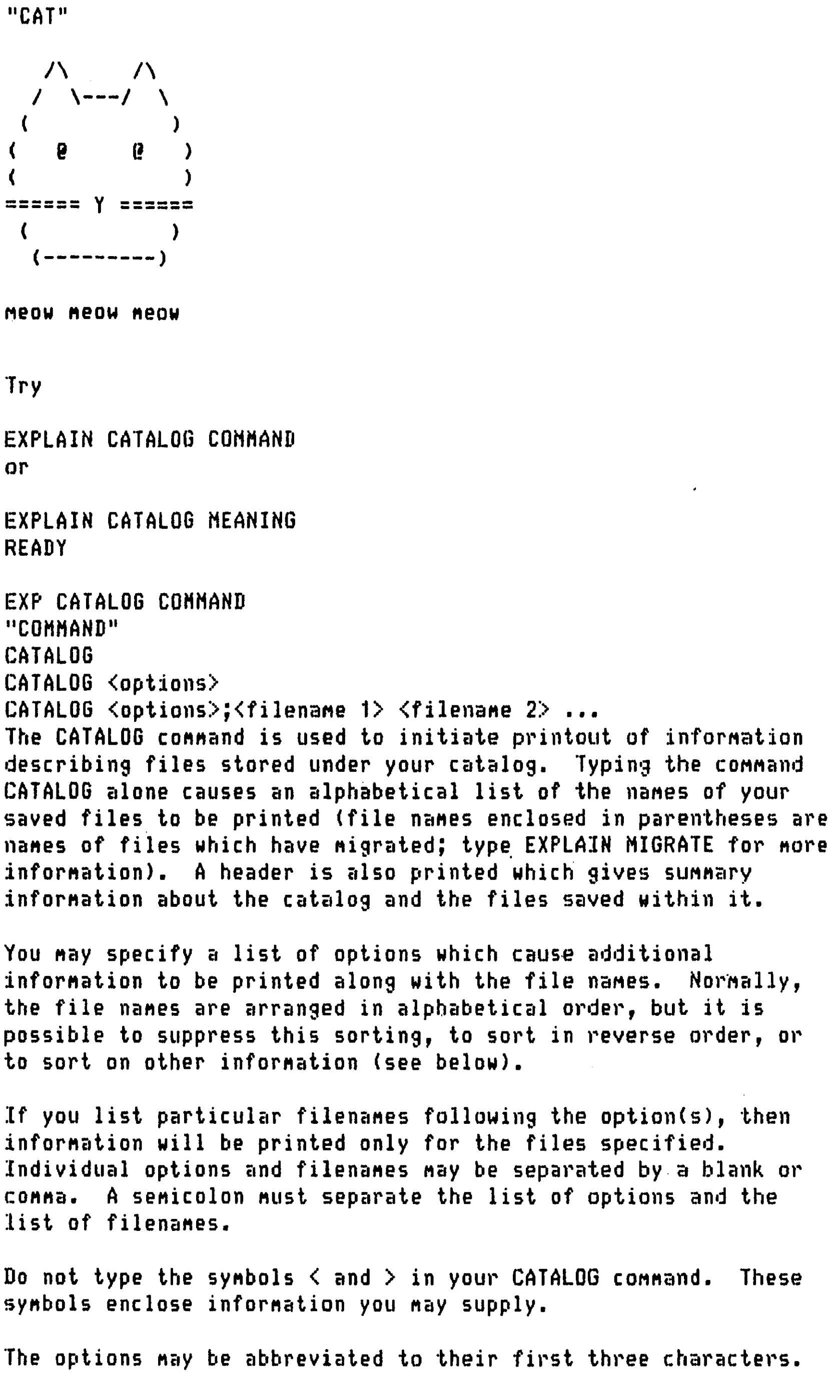
```
Other alternatives include:

    1. typing "REPALCE <filename>" which replaces the saved
       copy of <filename> with your current file and does not
       change the name of your current file.

    2. typing "SAVE <filename>" which saves a copy of your
       current file with the name <filename> but does not
       change the name of your current file.

READY
```

(b) Giving an unknown EDIT command:

```
EDI REF
Function not available. Type: EDIT HELP

STOP

READY

EDI HELP

Available functions are: APPEND, DELETE, DESEQUENCE, EXTRACT,
INSERT, JOIN, LIST, LOCATE, MOVE, PAGE, RESEQUENCE, PRINT,
SEQUENCE, DUPLICATE, MERGE, EXPLAIN, HELP AND STRING.

For and explanation of a function, type:
EXPLAIN EDIT NAME

where "name" is the name of an EDIT function.

READY
```

(c) Giving an illegal option to the CATALOG command. Most users would simply type CAT, but more sophisticated users may wish to find out more information than just the names of the files. The CATALOG command is the most complex command on the system and this explains the length of the explanation:

```
CAT XYZ

Illegal option: XYZ

STOP
READY

EXP CAT
```



```
"CAT"

   /\     /\
  /  \---/  \
 (           )
(   @     @   )
(             )
====== Y ======
 (           )
  (---------)

meow meow meow


Try

EXPLAIN CATALOG COMMAND
or

EXPLAIN CATALOG MEANING
READY

EXP CATALOG COMMAND
"COMMAND"
CATALOG
CATALOG <options>
CATALOG <options>;<filename 1> <filename 2> ...
The CATALOG command is used to initiate printout of information
describing files stored under your catalog.  Typing the command
CATALOG alone causes an alphabetical list of the names of your
saved files to be printed (file names enclosed in parentheses are
names of files which have migrated; type EXPLAIN MIGRATE for more
information).  A header is also printed which gives summary
information about the catalog and the files saved within it.

You may specify a list of options which cause additional
information to be printed along with the file names.  Normally,
the file names are arranged in alphabetical order, but it is
possible to suppress this sorting, to sort in reverse order, or
to sort on other information (see below).

If you list particular filenames following the option(s), then
information will be printed only for the files specified.
Individual options and filenames may be separated by a blank or
comma.  A semicolon must separate the list of options and the
list of filenames.

Do not type the symbols < and > in your CATALOG command.  These
symbols enclose information you may supply.

The options may be abbreviated to their first three characters.
```

The following options are the ones most useful to the typical user.

Option	Explanation
ACCESS	Access permissions; these determine how file may be used by system commands or programs
ALL	NAME, LEN, ACC, DLM, DLU, and USE
DATES	DLU and DLM
DLM	Date file was last modified
DLU	Date file was last used
EXPLAIN	Explains options specified on same line
LENGTH	Length in words of files and catalog (a word is 4 characters in ASCII files)
MIGRATE	Prints information about only files which have migrated
NAME	Prints names of files stored in your catalog
NMIGRATE	Suppresses printing of information about files which have migrated
OF <catalog name>	Prints information about files in the catalog specified; for example, CATALOG LENGTH OF SCODELIB***:BRIDGE would print information about the file BRIDGE in sublibrary SCODELIB*** of the public program library; for a list of sublibraries you may reference in the CATALOG command, type EXPLAIN SUBLIBS
RATIO	% of days file has been used since DLM
RSORT <argument>	Sorts information in descending order according to the argument specified; arguments are same as those for the SORT option
SORT <argument>	Sorts files before printing them according to the argument specified rather than by name; allowable arguments are ANAME, APASSWORD, DLM, DLU, LENGTH, NAME, OLENGTH, ONAME, PASSWORD, and USE; NAME is the default argument
SYSTEM	System under which the file was saved
USE	Gives number of different days a file was used since DLM

The options described below are not generally of interest to the novice DTSS user.

Option	Explanation
ANAME	Prints ASCII name; no octal conversion
CLIMB	Supplies information about subcatalogs
CURRENT	Information requested becomes your current file and is not printed at your terminal .RESULT. is the new current file name

EVERYTHING	NAME, LEN, OLE, ACC, OAC, DAT, USE, CLI, STO, OST, RAT, PRE, and SYS
FIND	Searches catalog and subcatalogs for specified files; type CAT EXP FIND
NFILES	Prints catalog information only
NHEADER	Suppresses header
NSORT	Suppresses sorting of catalog
OACCESSES	Octal representations of ACCESS
OLENGTH	Octal representations of LENGTH
ONAME	Prints octal representations of file names
OSTORAGE	Octal representations of STORAGE
PREFERENCE	Prints the file preference numbers; type CAT EXP PREFERENCE
PRINTER	Prints 120 characters per line; not 75
STORAGE	Total length of all files in catalog printed in header

Unless you have special privileges, the next options will work only for a subcatalog saved in your user number or for a subcatalog (such as a sublibrary of the public program library) for which you know the password. Note in particular that these options will not work on files saved in your user number.

Option	Explanation
APASSWORD	Prints ASCII password of file
PASSWORD	Prints password of file in octal

For a more complete explanation of any option type

```
EXPLAIN CATALOG <option>
```

For example, for an explanation of the option LENGTH, type

```
EXPLAIN CATALOG LENGTH
```

A discussion of all CATALOG options is available in Technical Memo TM041, "The CATALOG Command", available from the Documents Centre, Keiwit Computation Center, Hanover, NH 03755.

If a file name contains control characters, the file name will appear when you type CATALOG as a slash followed by 24 digits. These digits, taken in groups of three, are the octal numbers which correspond to the 8 characters of the file name (if your file name is less than 8 characters long the remainder is filled with spaces, octal number 040). You can determine the characters in the file name by referring to the list of ASCII code numbers in section 8.3 of TM075, "BASIC", or by running the library program CHARS***.

In your CATALOG command, you may specify a file or catalog name in octal as a slash followed by 24 digits that are the

```
octal representations of the ASCII characters in the name.  Eg,
CAT ALL;/101102103040040040040040 is equivalent to CAT;ABC.

This information was last updated 7/14/75.

READY
```

3.8.4 Perform files

Perform files may be used for a number of things. In the most general sense it may be used to change the user interface, but in practice it is primarily used to save typing time. I regularly use three perform files and these will serve as examples of their application.

I use DTSS for a great deal of word processing using a text formatting program called RUNOFF. I have a standard output file called RNOUT to which I direct all output from RUNOFF. I then print this file on a dual-case line-printer. The only parameter which varies is the name of the file to be processed. Thus I have a perform file called UT-RNO which contains the following:

```
$NO OUTPUT
EXE RUNOFF***
?$1,RNOUT,P
NEW LOWER
10 PRINT RNOUT
20    USING LOWER
30    FORMAT NSL
40 END
REP
$OUTPUT
BAC
```

which is called as follows:

```
PER UT-RNO;AGENDA
```

produces the following output:

```
BAC
***JOB ACCEPTED
READY

READY
```

and processes the file AGENDA. Another perform file I use is one to produce multiple copies of a file with a single banner page on either the dual-case line-printer or a local line-printer called REM1. This file, called QNH, contains:

```
EXE UT-C-NHD
PER XYZ
```

This expands into a sequence of events as shown below:

```
PER QNH
EXE UT-C-NHD
REM1 (Y OR N)? Y
NSL (Y OR N)/ N
FILENAME, COPIES? MAJOR,2
ANY MORE FILES (Y OR N)? Y
FILENAME, COPIES? CACM,4
ANY MORE FILES (Y OR N)? N
READY

PER XYZ

EXE UT-C-1FF
FILE NAME? GMB-NH
READY

OLD REM1
READY

BAC
***JOB ACCEPTED
READY

READY
```

First, the program UT-C-NHD is executed. This asks about the printer and printing options and the names and number of copies of each file to be printed. When all the files have been specified UT-C-NHD creates a file GMB-NH containing the material to be printed, a background file REM1 (or LOWER) containing a variant of:

```
10 PRINT GMB-NH
20    USING REM1
30    FORMAT NSL
40 END
```

and a perform file XYZ containing:

```
EXE UT-C-1FF; GMB-NH
OLD REM1 (or OLD LOWER)
BAC
```

The program UT-C-1FF deletes multiple form-feeds from the file GMB-NH. The whole sequence is somewhat clearer if the perform file is performed in brief mode:

```
PER QNH
EXE UT-C-NHD
```

```
REM1 (Y OR N)? N
NSL (Y OR N)? N
FILENAME, COPIES? MAJOR,2
ANY MORE FILES (Y OR N)? Y
FILE NAME, COPIES? CACM,4
ANY MORE FILES (Y OR N)? N
*PER XYZ
*EXE UT-C-1FF
FILE NAME? GMB-NH
*OLD REM1
*BAC
***JOB ACCEPTED
*
```

The last example of the use of perform files is my .SIGNON. file, which is automatically performed at sign-on time. The contents of the file is:

```
/BRI/MAX/USE/EXE MYDATES
$NO OUTPUT
/OLDUPDATE/EDI EXT 10-30/REP/BAC
```

which turns the terminal into brief mode, sets the memory limit to the maximum allowed in my validation record, prints out the number of users on the system, and then executes the program MYDATES which is a date reminder program. The output to the terminal is then shut off and a background job called UPDATE is submitted which manipulates the file containing the date reminder. A typical output from signing on is given below:

```
DARTMOUTH TIME-SHARING
LINE 2/0215 ON AT 14:29  20 DEC 77, 112 USERS
DTSS UNTIL 03:00 TODAY. LIST CCNEWS*** 11/22/77

USER NUMBER--B34360
/BRI/MAX/USE/EXE MYDATES

3)112 USERS
It's Harry's birthday on WEDNESDAY, 21 DEC 77
```

3.9 BACKGROUND JOB PROCESSING [8], [9]

The background job processing system has two components, the background systems, which converts job requests into background job descriptions, and the background monitor, BAKMON, (see section 6.3.3) which processes each job as resources become available. This section focuses on the background system.

The background system is convenient for running long programs, listing large files and making use of magnetic tapes.

Background programs describing activities to be performed are written in a line-oriented background language. Each line in a program begins with a line number consisting of a maximum of four digits. Messages appended to the program by BAKMON are prefaced by consecutive five-digit line numbers beginning with 10000 and refer to line numbers in the user's background program.

Each background program must contain an END statement. Activities described before the END statement are compiled into the background job description; lines occurring after the END statement are ignored.

The current version of a background program must be saved in the user's catalog. If it is saved with password protection, the background system asks the user for the password (overprinting the password as it is entered) before accepting the background job.

The background system determines from a user's validation information which activities that user is allowed to perform in background. It also computes limitations on run time and peripheral output. Users may specify, via the MAX command, smaller upper limits for certain activities. This feature is designed to allow a user to protect himself from a runaway program and also to supply information which may aid BAKMON in scheduling activities to be performed. When no MAX is specified for an activity and no limits are provided with the RUN command, small defaults are used; that is, a MAX of 32 seconds run time and 16K core. A user may find out what limitations are placed on his user number by typing RUN PER. BAKMON will allow four times the run time allowed in foreground.

All files named in a background program must have been saved prior to submission of the program. Neither the background system nor BAKMON saves a file for a user. When a saved file is protected by a password, the user has the option of identifying the file in his background program by typing:

<filename>,<password>

or

<filename>,

if he wishes to protect his password. In the latter case the background system requests the user to enter the password for the file and then overprints it for protection. For files without a password, the file name alone is supplied, as usual.

File names ending in *** denote library files, with the following seven exceptions:

(i) PRINTER*** denotes a zero-length scratch file which is provided by Bakmon for the given activity and which is listed on the high-speed printer upon completion of the activity.

(ii) REMn***, where 'n' is an integer between 0 and 9, denotes a zero-length scratch file which is provided by Bakmon for the given activity and which is listed on remote printer n upon completion of the activity.

(iii) SCRATCH*** denotes a zero-length scratch file which is provided by the background monitor for the given activity and which is discarded after completion of the activity.

(iv) TAPE*** denotes a scratch tape which is provided by Backmon for a given activity and which is discarded after completion of the activity.

All four of the preceding special files may appear more than once in a background activity.

(v) MYCAT*** denotes the user's catalog and is used to make files in the catalog available to Bakmon.

(vi) SCRATCHn***, where 'n' is an integer between 0 and 9 inclusive, denotes a scratch file to be provided by Bakmon which may be used by more than one activity in a background program. This file, known as a system scratch file, is created as a zero-length scratch file the first time it is referenced. In all subsequent activities, the file contains whatever data the user has written into it. The file is discarded after completion of the last activity in which it is referenced. A particular system scratch file may appear only once in an activity.

(vii) TAPEn***, where 'n' is an integer between 1 and 6, inclusive, denotes a scratch tape to be provided by Bakmon which may be referenced by more than one activity in a background program. This tape, known as a system scratch tape, is initially mounted with a write-permit ring and is assumed to contain no information the first time it is referenced. In all subsequent activities the tape contains whatever data the user has written onto it. Upon completion of the last activity in which it appears, the tape is released from the background job and made available to other background programs. A particular system scratch tape may appear only once in an activity.

Due to the nature of the background language, commands appear in a group with their related specifiers following them, forming blocks. A library program BKINDENT***, formats background programs in a manner which makes the command blocks easy to see. The complete list of commands and specifiers are given in Tables 3.1 and 3.2.

Table 3.1 – Complete list of commands.

Execution commands	Peripheral commands	Miscellaneous commands
DEBUG	COPY	ABORT
ORUN	PRINT	END
RUN	PUNCH	ERROR
TEST	READ	NUM
	SCRATCH	REMARK
	WRITE	STATUS
		WAIT

Table 3.2 – Complete list of specifiers.

Execution specifiers	Peripheral specifiers
FILES	AND
INPUT	BANNER
LINPUT	CTAB
MAX	FORMAT
OLD	FROM (FILE and TAPE)
OUTPUT	IDENT
TAPE	MODE
	POSITION
	REPORT
	TO (FILE, PRINTER, PUNCH, and TAPE)
	USING

A few of the more common commands are shown below as a way of illustrating the ease-of-use and power of the facility. Most programs are self explanatory. The simplest is the PRINT command, used to list a file on a printer.

```
100 REM PRINT 3 FILES WITH 10 PAGE MAXIMUMS
110
120 PRINT ALG1;ALG2;ALG3
130     MAX 10 PAGES
140     BANNER ALGOL60    'ADD BANNER
150 END
```

Other specifiers to the PRINT command will be shown in other examples. The RUN command is used to run the specified system.

When any of the specifiers INPUT, LINPUT, OLD, or OUTPUT is used in conjunction with a background RUN command, Bakmon performs a foreground simulation. When a foreground simulation is performed, FORGSIM is the module in charge of execution. The program runs in a manner paralleling that of a foreground run.

In terms of implementation, this means that several conventions are followed. First of all, Bakmon must be informed that a program is to be RUN by a statement of the form:

RJN [<system name>] [<runtime limit>] [<core limit>K]

where all bracketed material is optional.

Next, the user must specify the program which is to be run using the statement:

OLD <filename>

When a program is run in foreground, it may communicate with a user's terminal. In a foreground simulation, the terminal is simulated by specifying files from which input is to come (as when typing it) and to which output is to go (as printed on the terminal). To specify an input file, the user specifies:

INPUT <filename>

Similarly, to specify an output file, the user uses a statement of the form:

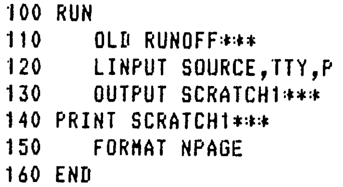
OUTPUT <filename>

In addition, the user may specify a line of input by using the LINPUT specifier, which is useful if only a small amount of input in necessary during the program's execution. Its format is:

LINPUT <date>

where <data> contains the information as would be typed at the terminal (including commas or other punctuation, if they would normally be necessary). Note that if either INPUT or LINPUT is used, OUTPUT must also be used.

The following example runs RUNOFF*** in background, specifying as the source the file SOURCE. It then prints the output on the high-speed printer:



```
100 RUN
110     OLD RUNOFF***
120     LINPUT SOURCE,TTY,P
130     OUTPUT SCRATCH1***
140 PRINT SCRATCH1***
150     FORMAT NPAGE
160 END
```

More examples of the RUN command are given below:

```
100 REM RUN A FORTRAN PROGRAM, FORTPROG
110 REM SPECIFYING A TIME LIMIT OF 64 SECONDS
120 REM AND MEMORY OF 40K WORDS
130 REM USING THE EXPERIMENTAL VERSION OF FORTRAN
140 RUN XFORTRAN 64 40K
150    OLD FORTPROG
160    INPUT DATA
170    OUTPUT OUTFILE
180    FILES DATA1;DATA2
190 END
```

```
100 REM RUN BACKJOB, PRINT THE OUTPUT,
110 REM AND SCRATCH THE OUTPUT FILE
120 RUN
130    OLD PROGRAM
140    OUTPUT OUTFILE
150 PRINT OUTFILE
160    FORMAT NPAGE
170 SCRATCH OUTFILE
180 END
```

```
100 REM PASS THE FILES DATA1, DATA2, AND
110 REM DATA3 AS #1, #2, AND #3 FOR
120 REM THE PROGRAM AVERAGER WHEN IT IS RUN
130 REM THEN PRINT OUTPUT WITHOUT FORMATTED PAGES
140 RUN
150    OLD AVERAGER
160    OUTPUT SCRATCH1***
170    FILES DATA1; DATA2; DATA3
180 PRINT SCRATCH1***
190    FORMAT NSLEW
200 END
```

The following examples all use magnetic tapes and serve to illustrate the various ways they may be used:

```
100 REM THIS PROGRAM MAY BE USED TO SORT IN BACKGROUND
110 REM RUNS SYSTEM SORT PASSING ERROR FILE ERMES,
120 REM CONTROL FILE CONTRL, INPUT TAPE Q-000 (THE
130 REM FILE TO BE SORTED), AND OUTPUT TAPE Q-001
140 RUN SORT
150    FILES ERMES;CONTRL
160    TAPE Q-000
170       POSITION 10
180    TAPE Q-001
190       POSITION 5
200 END
```

```
100 REM RUN SYSTEM GMAP SPECIFYING FILES
110 REM SOURCE, BINARY, AND SYMTABLE.  THE
120 REM LISTING IS TO GO TO TAPE1***
130 REM MYCAT*** IS PASSED TO MAKE OTHER
140 REM FILES IN THE CATALOG ACCESSIBLE
150 REM FORMAT NPAGE IS USED IN PRINTING BECAUSE
160 REM GMAP CREATES ITS OWN FORMAT CONTROL CHARACTERS.
170
180 RUN GMAP 64 32K
190    FILES MYCAT***;SOURCE
200    TAPE TAPE1***
210       POSITION 1
220    FILES BINARY;PRINTER***;SYMTABLE
230 COPY
240    FROM TAPE1***
250    TO PRINTER
260       FORMAT NPAGE
270 END
```

```
100 REM CREATE A TAPE WHICH CAN BE READ
110 REM AT IBM INSTALLATIONS WITH 7-TRACK HANDLERS
120 REM OUTPUT IS 10 CARDS PER BLOCK
130 REM IN THE IBMEL SET.
140 REM NOTE THAT THE LABEL IS OVERWRITTEN.
150 COPY
160    FROM FILE XYZ
170       FORMAT ASCII
180    TO TAPE 02389,FOREIGNTAPE
190       FORMAT NLABEL;NSERIAL;NRCW
200       FORMAT IBMEL;BLKSIZE 140;RECSIZE 14;FILREC
210       POSITION 1
220 END
```

The WAIT command is used to delay execution of a background program. Background does not check the availability of resources requested until execution time:

```
100 REM CAUSE RUN TO WAIT UNTIL LATE AT NIGHT
110 REM WHEN MORE RESOURCES ARE AVAILABLE
120 WAIT UNTIL 01:00
130 RUN
140    OLD PROGRAM
150    INPUT DATAFILE
160    OUTPUT PRINTER***
170    MAX 256 SECS
180    MAX 32K CORE
190 END
```

The ABORT command halts the execution of the background job(s) specified in the list. The job which contains the ABORT command will not be aborted even if its name is contained in the list, whereas other background programs of the same name will be aborted:

```
110 REM FORGOT POSITION ON INPUT TAPE

120 ABORT TAPEJOB
130 READ MYFILE;DATA
140    FROM TAPE Q-000
150       POSITION 5
160 END
```

An important aspect of background system is the checking of both the syntax of the background program and the availability of the required sources. This feature is designed to eliminate the frustration and delay caused by simple errors in the job control file not being detected until the job is run at a time when the user is not present. Nearly fifty errors are trapped, thus ensuring that any job submitted to Bakmon has a high probability of running. It does not of course guarantee anything about the programs which will be run nor the integrity of any data used.

3.10 REFERENCES

[1] Hardy, S., and Mather, D. (1977), 'DTSS User Primer', DTSS Technical Memorandum 022.

[2] Noyce, W., and Roberts, S. (March 1978), 'Simple Monitor', DTSS Technical Memorandum 060.

[3] Mather, D., and Oden, T. (1975), 'System Command Reference Manual', DTSS Technical Memorandum 085.

[4] Smith, L. (1973), 'Programming for the EXECUTE Command', DTSS Technical Memorandum 046.

[5] Mather, D., Pedersen, C., and Thompson, C. (1974), 'DTSS TEXT Editor Reference Manual', DTSS Technical Memorandum 004.

[6] Dunten, S. (1976), 'DTSS QED Editor Reference Manual', DTSS Technical Memorandum 036.

[7] Fernandez, G., Mather, D., and Smith, L. (1977), 'The CATALOG Command', DTSS Technical Memorandum 041.

[8] Mather, D. (March 1977), 'Introduction to Background', DTSS Technical Memorandum 053.

[9] Fernandez, G., and Ligett, S. (September 1977), 'DTSS Background Reference Manual', DTSS Technical Memorandum 088.

CHAPTER 4

The File System

4.1 TYPES OF FILE [1]

The concept of a file is central to the design of DTSS. A job communicates with everything outside its own memory space as a file. There are a variety of types of files which can be distinguished.

4.1.1 Data files

Data files are what the user typically calls 'files'. They are held on disc or magnetic tape. Such files can contain any information whatsoever. The space occupied by data files is meaured in words. Data files can be further categorised in a number of ways. Thus they may be catalogued or scratch files, that is, have an entry in a catalog or not; migrated or not, that is, have moved from disc to tape storage or not; and may be of a variety of structures according to the language being used to process it (see Table 4.1), such structures are not indicated in the catalog entry for the files but rather are determined by the run-time package of the language.

Table 4.1 – File formats supported under DTSS.

File format	BASIC	FORTRAN 76	COBOL	PL/I
Terminal format	●	●	●	●
Random string	●			●
Random numeric	●			●
Indexed sequential			●	●
Stream		●		
Literal	●	●	●	
Binary		●		
Formatted		●		
Unformatted		●		

4.1.2 Catalogs

A catalog is a collection of descriptors. Each descriptor describes some sort of file; the file may be a disc file, a catalog, a peripheral device, a core file, or one of several other things. For each file about which the catalog contains information, such things as the name of the file, its length, its password, access control information, and the location of the file are recorded. A catalog may contain entries for both data files and other catalogs. If there exists an entry in a catalog for a given file then that file is called a catalogued file. A scratch file differs from a catalogued file by having no entry in any catalog corresponding to it. A catalog with an entry in another catalog is called a catalogued catalog. Catalog entries are made by users with commands such as SAVE; entries are changed with commands such as REPLACE: and entries are deleted with commands such as UNSAVE. In all cases, changes are only made in the catalog if the appropriate permissions are available to the user/job attempting to make the changes.

4.1.3 Device files

All the non-allocatable devices on the system such as line-printers, card readers, and magnetic tape drives are catalogued in the catalog PERCAT. The entries in PERCAT are made by the Executive and are special catalogued files in that they are permanent and unalterable; thus no job may change the contents of PERCAT. That is, the files cannot be unsaved, appending data to them does not change the length of the file, only one job may have append permission on these or any non-catalog file to avoid two jobs appending at the same time. All the non-allocatable devices on the system are (special) catalogued files and appear to jobs as files. Thus information is written to a line-printer in the same way as information is written to any other catalogued file on the system. Some non-peripheral files are made to look like peripherals so as to have the properties of special catalogued files.

4.1.4 Communication files

Jobs on the system need to communicate with other jobs. Thus any job which needs to transfer information to and from the user's terminal must communicate with TCFACE. Inter-job communication is accomplished using communication files (explained in detail in section 5.3) which are a super set of the message system described by Brinch Hansen [2]. Communication files are not catalogued.

4.1.5 Job files

When a job is created by another job (say by SIMON) it is created as a running job file. Such files have various properties associated with them such as the maximum amount of memory the job is allowed, and the time allotted to the job.

4.2 A JOB'S VIEW OF FILES

Slave jobs on DTSS manipulate files by issuing calls to the Executive. These calls are made by executing a special instruction called Master Mode Entry (MME) with the address field coded to indicate what service is required by the slave job. In almost all cases, the instruction word itself is insufficient to indicate what is required and other information must be passed to the Executive. This information is loaded by the job in various registers, using a predefined protocol, prior to executing the MME instruction. When the MME instruction is executed, control is transferred from the slave job to the Executive. The Executive analyses the information passed to it and if it decides the operation is valid, initiates the service. Control is then returned to the slave job at the location immediately following the MME, the service may or may not have been completed by that time. A more detailed description of the various MME instructions is given in Chapter 5. All file manipulations are specified by slave jobs using MME instructions. The way in which the slave job indicates which file is being referenced is to specify a file reference number as part of the parameters for the MME. Each file available to a job has a unique file reference number for that job (that is, unique for each job but not unique across the entire system; thus a given file may be file reference number six for one job and file reference eight for another job). File reference numbers are obtained by a job in a number of ways. Files may be passed to the job by other jobs. For example, when a user types RUN to SIMON, SIMON runs a new job and passes three files to the new job with fixed file reference numbers (FRNs):

FRN 1:	The user's terminal (communications file)
FRN 2:	The current catalog
FRN 3:	The current file

When an editor is run then five files are passed to the job:

FRN 1:	The user's terminal (communications file)
FRN 2:	The current catalog
FRN 3:	The current file
FRN 4:	The output file
FRN 5:	The command-line file

For example, the command:

```
EDI JOIN ABC; XYZ
```

appends the file XYZ to the end of ABC and makes the result the current file. SIMON carries out this command by running a job which:

Reads command lines from #5
Looks up ABC in #2
Copies to #4

Looks up XYZ in #2
Appends to #4
Makes #4 the current file
Closes #3

File reference numbers may be obtained by issuing a MME to:

Open a catalogued file or catalogued catalog.
Create a scratch file or scratch catalog.
Create a communications file with another job.

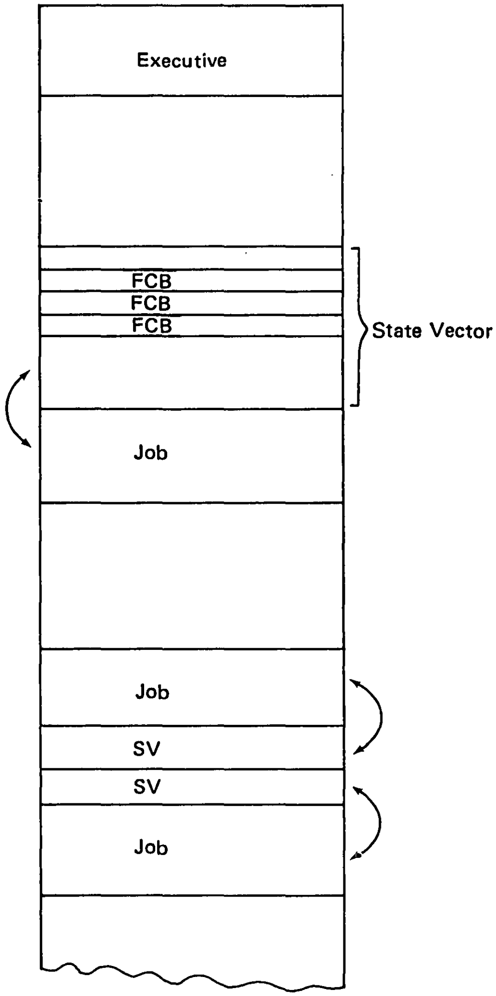


Fig. 4.1 – Jobs and state vectors.

A file reference number available to a slave job becomes invalid the slave job issues a MME to close the file. Data written to a scratch file is lost when that file is closed. Data written to a catalogued file is preserved after that file is closed and may be recovered by re-opening the file. A scratch file may be changed to a catalogued file by issuing a CATALOG MME. Part of the information supplied with a CATALOG MME is the name of the file, the file reference number of the scratch file to be catalogued and the file reference number of the catalog in which the entry is to be made. A job obtains a file reference number for a catalogued file by issuing an OPEN MME together with the name of the file and the file reference number of the catalog in which the file is catalogued. The UNCATALOG MME reverses the effect of the CATALOG MME deleting the file from the catalog and making the file a scratch file. The parameter of the UNCATALOG MME is the file reference number.

The file reference number is created by the Executive and is used by the Executive in processing file MMEs to locate a block of information relating to the file. This block of information is called a file control block (FCB). All the file control blocks of the files which a job has open at a given time are grouped together in a structure called a state vector. The state vector for a job may be in memory or swapped out; it is not part of the slave job's memory and cannot be manipulated directly by a slave job. Fig. 4.1 shows a number of jobs in memory and their corresponding state vectors. When a job is swapped in, its state vector is placed either just before it (that is, in lower memory), or just after it, but the base address register is set up to exclude it from the job's memory.

Table 4.2 – File types.

0 –	Scratch file
1 –	Scratch catalog
2 –	Catalogued file
3 –	Catalogued catalog
4 –	Special scratch file
5 –	Not used
6 –	Special catalogued file (typically a peripheral)
7 –	Not used
10 –	Communications slave end
11 –	Communications master end
12 –	Running job
13 –	Communications detached
14 –	Migrated
15 –	Not used
16 –	File open in shared mode
17 –	Not used

A state vector consists of three logical areas. The first area is a fixed length block of sixty (octal) words which is used by the Executive to record information about the job. For example, the jobs registers are stored here if the Executive suspends the job. There are also entries which are used to control the internal storage of the state vector. The second area is a variable length block of pointers to the file control blocks which make up the third block. Thus a pointer to file reference number three is always found in word sixty-three (60+3) of the

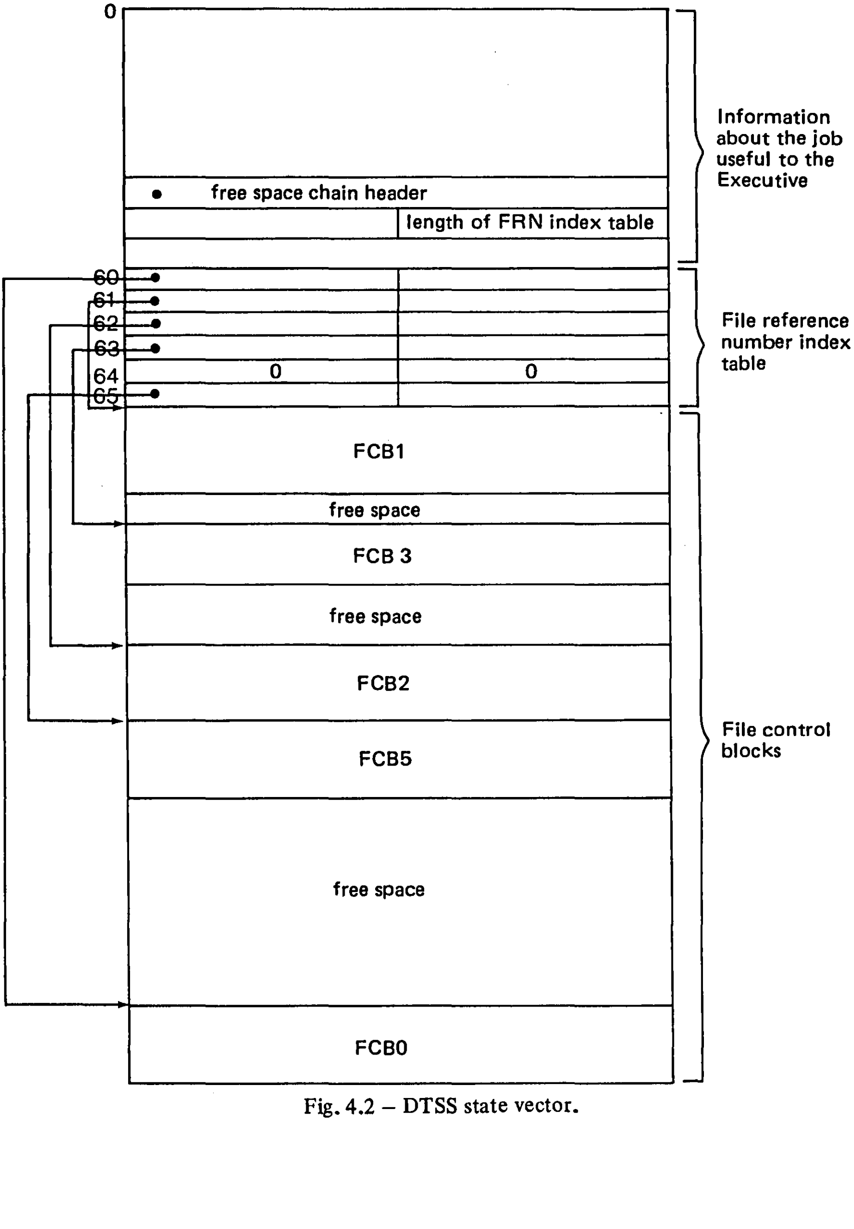


Fig. 4.2 – DTSS state vector.

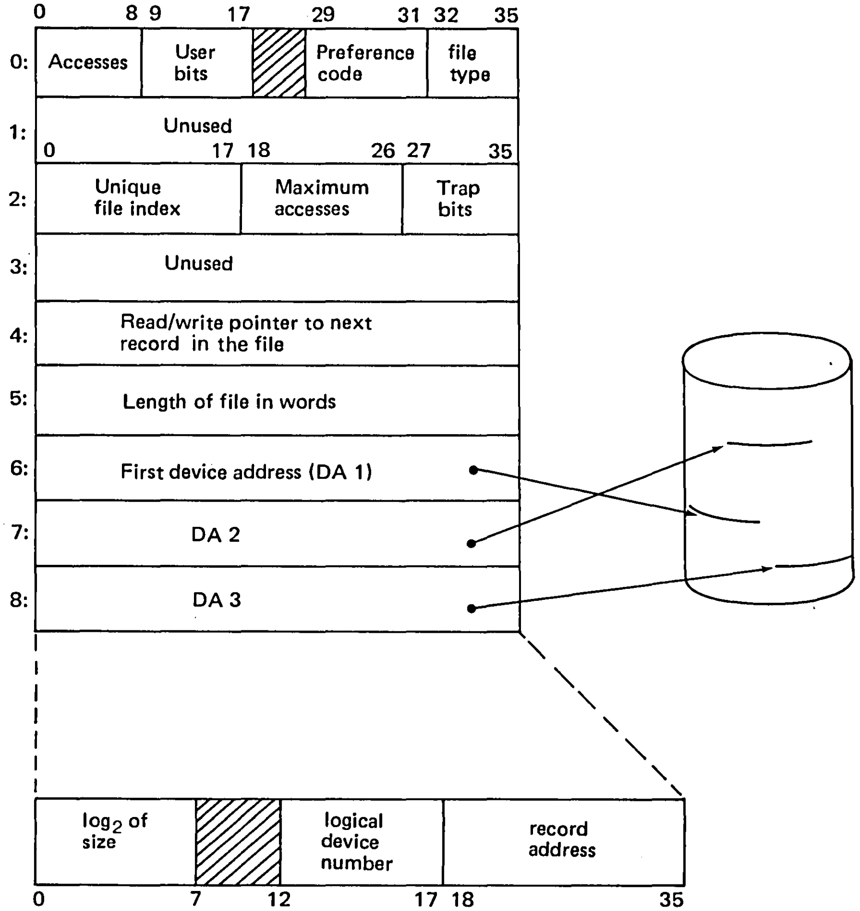


Fig. 4.3 – File control block.

state vector. Each of the index words consists of two fields, a pointer to the file control block and the length of the file control block is words. A state vector is shown in Fig. 4.2.

A file control block is between six and thirty-two words long. Fig. 4.3 shows a typical file control block. The first two fields in word zero determine the actions that can be performed on the file at present and are discussed in more detail in section 4.5. The preference code is used to determine the type of storage to be used for the file on disc. In particular, it indicates whether it is to be stored on the center tracks of a moving head disc (which have the lowest average seek latency) or on the inner or outer tracks (which have the highest average seek letency). The file type is one of the types listed earlier and the coding is given in Table 4.2. Word two contains a unique file index used by the Executive to ensure two jobs don't try to write to the same file, together with access information (see section 4.5). Word four contains a pointer to the next word to be read or written. A file may be open for reading and for writing by a given job, but the Executive maintains only a single pointer. Word five contains the length of the file in words. Words six and higher contains the device addresses of the various segments of the file. If the file is stored in one contiguous area on a simple device then only one device address word is needed.

File reference number zero is special and does not conform to the normal FCB format.

4.3 CATALOG ENTRIES [3]

Each entry in a catalog is a descriptor. The information held for each file in its descriptor is made up of the information held in a state vector file control block. Catalog descriptors are created by copying information from the state vector file control block and appending three extra fields:

(i) The name of the file (specified in the CATALOG MME).
(ii) The password associated with the file, used to control access to the file (also specified in the CATALOG MME), together with other access information.
(iii) Usage information generated by the Executive at the time of creation and modified by the Executive on usage.

Each descriptor may be broken into two halves: one half contains logical (device-independent) information, and the other specifies where the object is stored by a list of device addresses. In general, the logical information may be viewed and/or directly modified by slave programs, while the device addresses may not be.

There are only two types of descriptors: catalog descriptors and 'file' descriptors. Both types have eight words of logical information. Catalog descriptors include exactly one word for device addresses, while file descriptors may have from zero to twenty-seven words for device addresses.

The device address (DA) in a catalog descriptor points to the first part of the catalog it describes; this first part contains a catalog header which describes storage for that catalog. A catalog header consists of eight words of logical information and twelve DA's.

A DA is full word quantity specifying where part or all of an object can be found. The format of a DA is shown in Fig. 4.4.

The device code and record number are internally mapped by the Executive into the true physical addresses.

The device code (DC) is the most important part of a device address, for it specifies what kind of peripheral the object is. Most objects are disc files/catalogs, so the device code reflects which disc drive their data are stored on, Peripherals such as printers, tape drives, card readers, and so on, all have their own device codes. The device code is the only part of an object's description which specifies the physical type of the file. Since most slave jobs cannot look at device addresses they cannot determine an object's physical type (whether it is a peripheral file, etc.) from its description. They must open it and perform operations on it to deduce its type.

0 7	8 11	12 17	18 35
EXP	X	DC	RN

EXP: log2 of the number of contiguous records in DA.
X: (not used).
DC: logical Device Code (logical device number in ENV deck, see section 8.1.4).
RN: Record Number of first record in DA.

Fig. 4.4 – Device address word.

The EXP (size) field is only used for disc files/catalogs. It contains the log base two of the number of records in this device address, some or all of which are used to hold a section of the file catalog. When the Executive allocates room for a file/catalog on a disc, it finds areas of storage big enough to hold all the data. The number of words in each area is a power of two. The device address list, if read in order, contains the addresses of each section of the file in the proper order.

If data is appended to the end of the file/catalog, the remaining unused space in the area pointed to by the last DA is filled. When the area pointed to by the last DA is completely filled, a new area is allocated and its DA appended to the end of the list in the FCB. Then data is written into the area associated with that DA.

Note that since FCB's may hold no more than twenty-seven device addresses, files' sizes may be limited even though there is enough disc space to hold more data. The Executive's algorithm for allocating various sized areas is designed to minimise this problem, but has the side effect of reserving too much space for files. This space is periodically regained by the Shuffle module which opens files which are not optimally allocated, moves them into preallocated scratch files, and then replaces the old files with the scratch files. When Shuffle opens each scratch file, it can tell the Executive how long the scratch fille will be so that the Executive may allocate device addresses optimally for it and avoid wasted space.

Catalogs are always allocated on dics, and the twelve DAs point to the various sections of the catalog. (Naturally, not all twelve need be used.) Two special catalogs, the MFD and SMCORCAT, are allocated in memory.

Storage management within catalogs works as follows. Assume the picture of a catalog as shown in Fig. 4.5.

'Holes' are created whenever objects are unsaved from catalogs. Holes are simply unused descriptors. When a new descriptor is added to a catalog, the

Executive searches from the beginning of the catalog to find the first hole big enough to put the descriptor in. If there are no holes big enough, the descriptor is added to the end of the catalog and the catalog's length is updated.

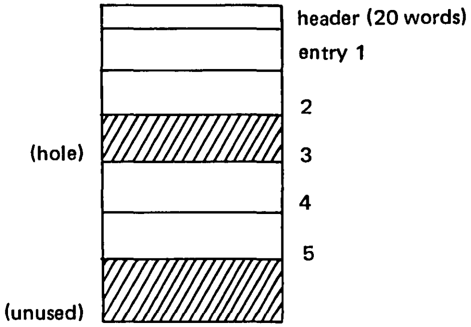


Fig. 4.5 – Typical catalog.

Essentially the same thing happens when a file expands and needs a new DA in its descriptor. The descriptor is moved to the first hole big enough, or is added to the end of the catalog. Adjacent holes are never coalesced in a new, bigger hole. This is because the Executive maintains a table of open files, and this table contains the entry number of each file in its catalog. Entry numbers must include holes, or removing a descriptor from a catalog would change the entry numbers of an unknown number of open files in this table. Note that entries (holes and descriptors) are essentially maintained on a sequential list. Thus searches for a descriptor must proceed linearly through all entries until it is found.

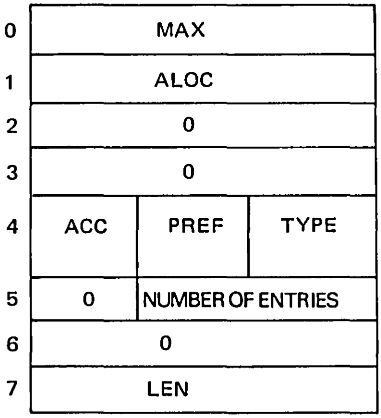
Finally, due to the field size allocated for entry numbers in the Executive's table of all open files, catalogs may contain no more than 4095 entries (holes and descriptors).

Subcatalogs may be nested within catalogs an arbitrary number of levels deep (subject only to rules on storage quotas). However, entries more than 32 levels deep are not preserved by logical dump/logical load. (See section 4.8.)

Slave jobs may view a portion of each descriptor in a catalog, and may obtain information about the catalog itself. The information they may see includes information which governs access to the file, file identifiers, and so forth. This information is summarised below, in the forms in which it can be obtained by the READ CATALOG MME (see Chapter 5).

Information about all objects saved in a catalog may be gained by applying a READ CATALOG MME to the catalog. The MME returns an eight word block for each descriptor; this descriptor is not in the same format as the Executive maintains it internally, and does not provide the device addresses in each descriptor.

The zeroth entry returned on a READ CATALOG MME contains information about the catalog itself. This information is shown in Fig. 4.6.



pref: bits 26–31;
type: bits 32–35

Fig. 4.6 – Catalog header information.

MAX

The maximum length which the catalog may hold. This maximum and its effects are described below.

ALOC

The current length of all objects contained in the catalog plus twenty words for the catalog header, plus twelve words for each file descriptor in the catalog. (Twelve is somewhat arbitrary, since file descriptors may range from eight to thirty-five words long. It is chosen as a good average length.)

Every catalog contains a twenty word header describing the catalog, so the minimum ALOC for a catalog is twenty words.

The length of each file in the catalog is added into the ALOC; but each subcatalog has its MAX added into the ALOC. This means that changes to the lengths of files in subcatalogs do not affect the ALOCs of all catalogs containing the subcatalog.

ACC

The accesses with which the slave job has this catalog open.

PREF

The catalog's preference. See the discussion of preferences below.

TYPE

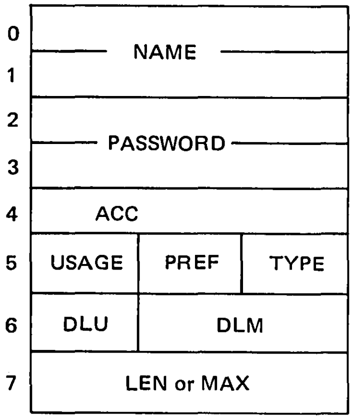
The file's type. Described in Table 4.2 and below.

#ENTRIES
The number of entries (descriptors and holes) in the catalog.

LEN
The length of the catalog itself: twenty word header, descriptors, and holes.

The device addresses may be obtained by opening each object and applying a PROVIDE DEVICE ADDRESS MME to the object. Jobs may only issue this MME if they are Load-Dump enabled (see section 4.8), and so most jobs may not see an object's device addresses. Since device addresses are the only way to determine peripheral types, most jobs may not readily discover whether or not a given catalog contains peripherals.

The descriptor is shown in Fig. 4.7.



pref: bits 26–31;
type: bits 32–35

Fig. 4.7 – File descriptor.

NAME
The name is a unique identifier for this descriptor, and hence for the object it represents. No two descriptors in a catalog may have the same name, and the name is the only identification for a descriptor.

PASSWORD
If a job supplies a password when trying to open a file, the password is compared with this field, and no accesses are allowed if the password does not match. If the access word has the slave-trapped bit set, then this field contains the filename of a slave trap program (in the same catalog) to be run whenever some job attempts to open this file. (See section 4.5.3 and 4.5.5.)

If the file is master-trapped, the password field is irrelevant. (See section 4.5.5.)

ACC

This word contains access bits, which are used to determine how various jobs may handle the object associated with this descriptor. (See section 4.5.2.)

USAGE

The usage field contains a count of how many different days the object associated with the descriptor has been used since it was last modified. A file is 'used' when it is opened. The day when the object was modified is not included in this count.

PREF

An object's preference determines which storage class its storage is allocated from. In general, lower preference files are saved so they are faster to access, but this is not an inherent property of preferences. They are simply a means of partitioning storage. The various preferences and how they are used are:

0 – Allocated in memory
1 – Swap storage
2 – Monitor scratch files, and files in :SMCORCAT
3 – Catalogs
4 – Normal scratch files
5 – Special data base
6 – Normal saved files
7 – Permanent data bases

In general, if a file needs storage of some preference and none is available, it will be allocated storage from the next higher peference which has room.

Most catalogs are preference three. MFD and SMCORCAT are preference three to zero. However, jobs will always get preference three when they open a scratch catalog.

TYPE

An object's type to a large extent determines what can be done to it.

The difference between a type six file and types zero through three files/catalogs is that the device addresses for type zero through three files/catalogs are drawn from a system-wide pool, and storage for these files forms a part of the DTSS file system. Special catalogued files have their own preallocated device addresses which point to storage which is dedicated to these files.

Peripherals are saved as special catalogued files, since their device addresses point to the devices themselves. The Executive makes a special check for special catalogued files, and does not update their lengths when data is written to them. Also, special catalogued files may not be unsaved or erased.

Migrated file descriptions contain no device addresses since the files they represent have moved to tape.

DLM

This field contains the coded form of the date the object was last modified. An object is modified when it it catalogued, its length changes, or when data is written to it. The DLM of a catalog is modified when its MAX changes.

DLU

This field contains a coded form of the date last used. An object is 'used' when it is opened, replaced, etc.

LENGTH (or MAX)

If the descriptor is for a file, this field contains the file's length in words. For a catalog, this field contains the catalog's MAX as described earlier.

Associated with every catalog is a catalog maximum, which is specified when the catalog is first created as a scratch catalog, and a catalog ALOC. The catalog ALOC is the total length of all files in the catalog plus a small overhead for the catalog itself. When a catalog is catalogued, the catalog maximum of the subcatalog is added to the ALOC of the supracatalog.

A catalog's quotas are determined by the MAX word in its header. A catalog's MAX does not place any upper bound on its ALOC (length of its included files and subcatalogs); rather it limits the accesses which jobs may obtain upon opening a file when the ALOC>MAX. That is, if a job can open a file with append permission, it can write as much data into the file as it wants, limited only by the job's quotas (discussed in the next section).

Three cases are notable for the relations between ALOC and MAX:

(i) ALOC⩽MAX. A file may not be replaced if its new length would push the allocated length over the catalog's maximum. But if a job can open a file with append permission, the amount of data which it can append is limited only by its job quota.

(ii) ALOC>MAX. Jobs may never get append permission on any objects within the catalog. Objects, however, may be replaced with objects of smaller lengths, or erased.

(iii) ALOC>(2*MAX). Files may not be opened. They may, however, be erased or replaced with smaller files.

The catalog MAX also determines whether a user may catalog new files in the catalog. Once a catalog exceeds its MAX, no further files may be catalogued in that catalog.

Jobs with special permissions may open scratch catalogs with infinite MAX's. Infinite MAX's are flagged with the sign bit in the MAX field set; quota checks are suspended for infinite MAX catalogs. Such catalogs may only be catalogued within other infinite MAX catalogs, which then have the sign bit set in the ALOC field to signal an infinite length catalog.

Note that scratch catalogs with infinite MAX's may be catalogued within themselves, but may not then be catalogued elsewhere (since already catalogued).

To prevent jobs from running amok and filling all system storage with unnecessary data by appending to files, each job has a quota of how many words it may add to catalogued or scratch storage. For user's these quotas are set by the monitor they are running under according to information from validation files passed by LOGIN. These job quotas are equal to the user number catalog quotas. (See section 4.5.1.)

The CATALOG command to SIMON [4] allows the user to display in a variety of formats the information held in the descriptors in the catalog. Thus, the command:

```
CAT ALL
```

might produce the following output:

```
HDK52500  27 Dec 77  16:44

Length       54268      Max       81920

.MAIL.                0  RWAL       RWAL
12/27/77  12/27/77      0    6     SBASIC

.SIGNON.             20  RWAL       RWALP
12/27/77  11/29/77     16    6     BASIC

C-UPDATE            826  RWAXCP     RWAXCP
12/27/77  11/23/77     16    6     SBASIC

CACM1              1498  RWAL       RWAL
12/23/77  12/22/77      1    6     BASIC
             .
             .
             .
             .

RECIPES            4000  COSRWA     OSRWA
12/23/77  12/23/77      0    0     ****

Storage used    51290

READY
```

where the entries for each file are as follows:

1. File name
2. Length in decimal
3. Letter code accesses available without a password
4. Letter code accesses available with a password

5. Date last used
6. Date last modified
7. Number of days used since date last modified
8. Preference
9. System under which the file was saved

The READ CATALOG MME is the Executive call which provides this information.

4.4 THE CATALOG TREE

Since data files and catalogs may be catalogued in other catalogs, which may in turn be catalogued in further catalogs, the filing system as a whole has the structure of a tree, and its known as the catalog tree. The root of the tree is the master file directory (MFD) and this file is supplied to all jobs as file reference number zero. In general, the closer a file is to the root the more likely a file is to be a system related file, and, as such, likely to be available for public perusal. The main exception is the Master User Directory (MUD), in which groups of user numbers and user catalogs are saved. Here, since most user files are protected against other users, it is usually impossible to get files from other user catalogs without special arrangements (see Section 4.6). Fig. 4.8 shows a schematic overview of the structure of the DTSS file system. Some of the more important catalogs in the MFD are as follows:

The catalog DLIBRARY contains all files in the program library, including sublibraries which are subcatalogs of DLIBRARY. PLOTLIB*** is an example of a subcatalog within DLIBRARY.

The catalog SMSYSCAT contains the systems that may be run by Simple Monitor for a user. The compilers for BASIC, FORTRAN, COBOL, APL, and others are found here.

XSYSCAT is the counterpart of SMSYSCAT and contains experimental systems. These versions of the SMSYSCAT systems are under development and often contain new and useful features. There is usually an 'X' system for each system in SMSYSCAT (for example, XBASIC and XFORTRAN).

PERCAT is the catalog in which the peripherals such as the card punch, the line-printer and the magnetic tape handlers are catalogued.

The Master User Directory (MUD) contains catalogs called super groups. Super group names are formed from the first three characters of the eight character user number. The catalogs within a super group are called groups; their names consist of the first five characters of a user number. Each group may contain up to a thousand individual user catalogs. Each user catalog may, of course, contain its own files and/or catalogs. Other catalogs contain such files as billing files (CATBV), background-related files (BAKCAT), and monitors (MON).

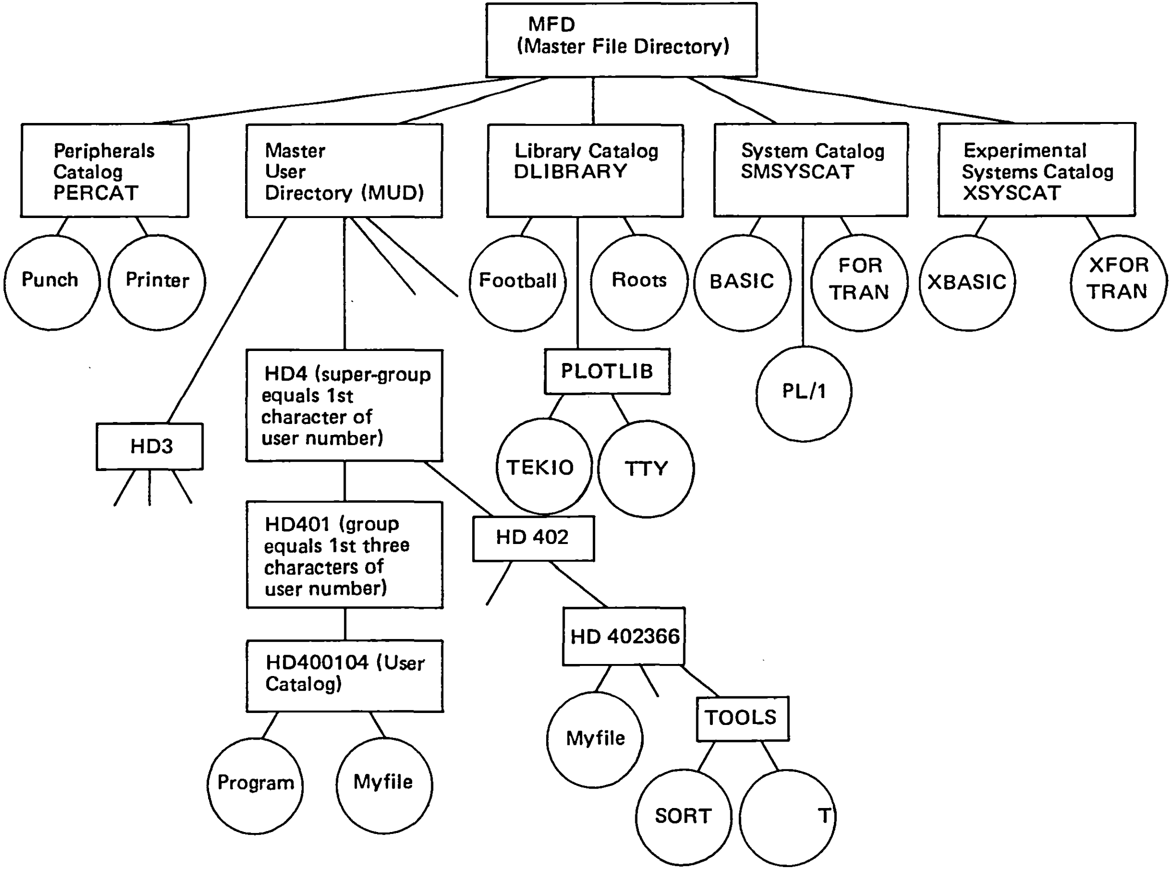


Fig. 4.8 – DTSS file system.

A filename is part of the treename, since a filename shows the location of the file in the catalog tree. A treename lists the catalogs which contain the file, in the order in which the catalogs are encountered as you move down the tree to reach the file, followed by the name of the file. Each section of the treename is preceded by a colon. Thus the treenames of some of the files in Fig. 4.8 are as follows:

```
:MFD:DLIBRARY:TEACHLIB:LIBCAT
:MFD:SMSYSCAT:BASIC
```

Since typing such long files is awkward, many parts of the system accept certain abbreviations:

MYFILE The name of a file in the current catalog.

:MFD MFD is assumed to be the first catalog in any tree name and thus may be omitted from a tree name. For example:

:SMSYSCAT:BASIC is short for:

:MFD:SMSYSCAT:BASIC

*** For files and catalogues in DLIBRARY. Three asterisks following a file name indicate that the file is part of the Program Library, DLIBRARY. Thus:

LIBINDEX*** is short for:

:DLIBRARY:LIBINDEX

TEACHLIB***:LIBCAT is short for:

:DLIBRARY:TEACHLIB:LIBCAT

For user catalogs in the Master User Directory. Instead of typing the complete treename of a file saved in another user's catalog, the user number may be preceded with an asterisk. For example:

*41734J:SURVEY is short for:

:MUD:HD4:HD417:HD41734J:SURVEY

There is also a SIMON convention for referring to the catalog corresponding to the user number under which the user is currently signed on, namely:

*MYCAT

4.5 SECURITY SYSTEM [5]

The objective of a security system is to allow user to do useful, legal work in an easy and straightforward manner whilst maintaining the integrity of the system and the privacy of non-puplic information. Since everything is a file, control of what a user can do to DTSS or on DTSS comes down to control of what a user can do to files and in particular, what a user can do to specific files.

Because file protection needs vary greatly, there are several levels of security with which files may be protected. Thus some protection can guard against accidental destruction (by the owner or other users), while others can withstand casual attempts to access or change a file, and still others can resist determined efforts to access the file. The security system regulates the actions users may perform on catalogs and files and controls which catalogs and files they may use. Thus user catalogs are private in that other users may not normally retrieve files saved in another user's catalog. Most non-user catalogs on the system are public and users may enter them and call up files saved in them. Certain catalogs are protected when confidentiality is required. Thus most users are unable to look into the catalogs in which running background jobs are saved and cannot look in sublibraries which have been protected at the request of that sublibrarian.

There are broadly five levels of protection of increasing power and these are described below.

4.5.1 User number protection [6]

As stated above, files in user catalogs are not normally available to other users and are only available to a user signed on under the given user number. If this is to have any meaning in terms of security, the user numbers themselves must be protected.

Each DTSS user has an eight-character identification code called a user number. An arbitrary sequence of up to eight characters may be associated with each user number; they must be supplied whenever the user attempts to sign on to the system. This sequence is called a password; it provides an additional level of security. At sign on, the user is asked for a user number and a password. The password is either overtyped or entered on top of an overtyped area on the line depending on how the user responds. This method of overprinting was chosen, rather than echo suppressing, to deal with half duplex terminals. Since it is possible to respond in such a way that the password is not obliterated, users may choose a password consiting of non-printing characters. Users may not change passwords themselves.

Some terminals have an identification device called an answerback drum, which may be interrogated under computer control. Thus it is possible to determine the terminal from which a user is calling and to refuse him service if he is attempting to access the system via a terminal he has not been authorised to use.

An additional control is provided by the time of day at which a user is attempting to gain access to the system. Each user number has associated with it a set of legal sign-on times defining those time periods during which that number may access the time-sharing system. The sign-on times range from zero to twenty-three in one-hour intervals, based on the twenty-four-hour clock.

The specific resources available to a user after he has successfully identified himself to DTSS are controlled by various parameters associated with his number.

CATALOG STORAGE

The catalog storage allotment determines the amount of storage which a user or group of users may occupy with saved programs and files. This limit is specified in multiples of 1024 (1K) words and may range from 0 to 65 million words. The value MAX may be used to specify the system limit.

SCRATCH STORAGE

The scratch storage allotment determines the maximum amount of temporary uncatalogued storage that a user may employ while connected to DTSS. All data in scratch storage are lost when the file is closed. This limit is specified in multiples of 1024 words and may range from 0 to 65 million words. Scratch quotas normally are eight times catalog quotas. The value MAX may be used to specify the system limit.

RUNNING TIME

The running time allotment determines the maximum amount of computer resource units (CRU) a user may obtain while executing any one program. This limit is specified in CRUs and may range from one second to approximately four hours. The value MAX specifies unlimited CRUs. Internally, the number of CRUs is stored as the log base 2 of the CRU time. If the number of CRUs requested at validation time is not a power of two, the next higher power is used.

PERMISSION

Each user number has a set of special permissions associated with it. These permissions are of the on/off type, and control a wide variety of miscellaneous functions such as the ability to use the background system, the ability of use the Operations Monitor, the ability to use the high-speed printer, and so on. Each permission occupies one bit position in full 36-bit word set aside for this purpose. Table 4.3 briefly describes each of the available permissions.

Table 4.3 – Permission word entry.

Bit	Name	Meaning
0-5		User partition category.
The next six bits control the user's ability to use background and the various peripherals.		
6	P$BAK	May run programs in background.
7	P$PUN	May use card punch.

Table 4.3 – *continued.*

Bit	Name	Meaning
8		Spare.
9	P$MTH	May use magnetic tapes.
10	P$PRT	May use high-speed printer.
11	P$GEC	May use GECOS III simulator.
The next eight bits control miscellaneous privileges pertaining to the running of programs.		
12	P$SYS	May run any system.
13	P$OPR	Operator status.
14	P$LGC	May use 24K memory.
15		Spare.
16	P$SLC	May use 32K memory (128K in background).
17	P$PR5	May create preference 5 files.
18		Spare.
19		Spare.
The next nine bits are job access permissions. These bits determine which privileged files and catalogs the user will be able to access when running a job under SIMON.		
20	A$VAL	Protects validation files.
21	A$MUD	Protects the Master User Directory (MUD).
22	A$BIL	Protects billing catalog and files.
23	A$PER	Protects peripherals.
24	A$BMC	Protects background catalogs.
25	A$LIB	Protects the public program library (:DLIBRARY).
26	A$MOS	Protects monitor and system catalogs.
27	A$MFD	Protects the Master File Directory (:MFD).
28	A$XSY	Protects the experimental systems catalog.
Bits 29-35 control the user's ability to use privileged system monitors. These permissions are checked by LOGIN.		
29		Spare.
30	P$FAST	Allows access to the Priority Demonstration Queue unless bits 29-33 and 35 are set.
31		Spare.
32		Spare.
33	P$NAN	May access Validation Monitor.
34	P$CTY	May access Operations Monitor from a control line.
35	P$SUP	May access Super Monitor (a debugger).

4.5.2 Access permissions

Access permissions are the most fundamental means of protecting files on DTSS. They limit the actions which may be performed and facilitate selective access.

All files, including catalogs, have access permissions associated with them that determine which operations may be performed and how the file may be retrieved. The permissions, their letter code abbreviations, and meanings are as follows:

A APPEND: Permits the user to add information to the end of the file. In the case of a catalog this means entries may be added to the catalog.

C COMPILE: Added automatically when a file is compiled; X or R must also be used, depending on the language in which the program is written.

F FETCH: The file may be retrieved by any program or system command, even if the catalog in which the file is saved is not itself accessible (as for other user numbers). The operations which may be performed on the file depend on the permissions available.

G GROUP: If R permission is also present (or XC for compiled files), the file is available to user numbers in the same user group via the OLD command. (A user group consists of any set of user numbers having the same first five characters.)

L LIST: The file contains printable characters; if READ permission is also available, it may be listed.

P PUBLIC: If R permission is also present (or XC for compiled files), the file is available to any user via the OLD command.

R READ: Permits the user to look at the contents of a file. In the case of a catalog this means examine entries in the catalog.

S SHARED: Permits the file to be open by more than user for writing (see section 4.6).

T TRAP: This specifies either that the file is a slave-trap program (when in the protected accesses) or that it is protected by a slave-trap program (when in the protected accesses). (See section 4.5.5 for more information on slave-trap programs.)

W WRITE: Permits the user to change the contents of the file.

X EXECUTE: The file may be executed directly as machine-language instructions; usually appears in conjunction with Compile permission.

Three permissions apply only to catalogs:

C CATALOG: The file is a catalog. Catalog permission can occur only among accesses available without a password (see section 4.5.3). C represents Compile permission when it appears as an access available with a password or when it appears as a non-password access of a file that is not a catalog.

O OWNER: Used only with catalogs; means passwords may be read with a READ CATALOG MME, password protection on files is ignored, and the protected accesses are assumed to be RWAX.

S SEARCH: Used only with catalogs; allows access to files in the catalog.

No permission implies any other permission. Thus, for example, write permission does not imply append permission.

In addition to the above access permissions, there are other ways in which the system protects itself from mishap. That is, there are rules to prevent usage conflict. These rules prevent access to partially updated information. They specify that it is possible for any number of jobs to have a file open with Read or eXecute permissions. However, any job having Write permission prevents access by all others seeking Read, Write, Append or eXecute permissions. A job which has any permission on a file prevents access by jobs seeking Write permission. Append prevents access by others seeking Append except when the file is a catalog.

Table 4.4 is a list of some frequently used SIMON commands and the minimum access permissions needed for successful completion of each command:

Table 4.4.

Command	Minimum accesses required on file
LIST	R and L on current file.
OLD	R (or XC for compiled files) on file to be called up. If the file is saved in a catalog on which the user does not have search permission (typically another user's catalog), then fetch, public or group permission must also be available.
REPLACE	R on current file and RWA on file to be replaced. If the file is saved in a catalog on which the user does not have search permission, then fetch permission must be available.
RUN	R (or XC for compiled files) on the current file.
SAVE	R on the current file; A on the current catalog.
SCRATCH	None if SCRATCH without <filename>; RWA on saved file if SCRATCH <filename>. If the file is saved in a catalog on which the user does not have scratch permission, fetch permission must be available in the file.
UNSAVE	RWA on the file to be unsaved. If the file is saved in a catalog on which the user does not have search permission, fetch permission must be available on the file.

Thus the following are true:

- A user may not replace, scratch or unsave a file in DLIBRARY since although the user has search permission on the catalog (to allow users to access files in DLIBRARY) the files are not saved with RWA permission.
- A user may not in general replace, scratch or unsave a file in another user's catalog although the file may be saved with RWA permission, since the user will not have search permission and unless the owner of the file has explicitly taken steps to give it, the file will not have fetch permission. The access permissions on the current file depend on how the current file was created. The NEW command creates a file with RWAL. The OLD command creates a file with the permissions of the saved file (but see section 4.5.3).

4.5.3 Passwords on files

Files may be saved with a password. This allows the user to restrict access to a saved file. Anyone attempting to access the file may do so by specifying the file name either with or without the password. Normally, the access permission available to the user supplying the password is greater than to the user not supplying the password. Indeed, it is possible to specify no access whatsoever without the password.

The format of the SAVE command is:

SAVE [<filename>] [,<password] [;<accesses>]

The simple form of the SAVE command is without any options in which case the file is saved with the current name, with a default password (of eight spaces, created by SIMON) and default access permissions of RWAL both with and without password. (The two sets of access permissions, without and with password, are called the unprotected and protected accesses.)

Typing:

```
SAVE,<password>
```

gives RWAL (the default access permission) with the password and no access without it. Two sets of access permissions may be used to grant selective access to password files. The two sets determine, respectively, which operations will be allowed on the file when no password is supplied and which may be performed with the correct password. The two sets are separated by a comma. The file is saved with the specified accesses, provided RWA accesses are part of one set. Examples:

```
SAVE,PASS;RL,RWAL
SAVE MYFILE,PASS;RPL,RWAL
SAVE,PASS;,RWAL
```

The first example allows the user himself to read and list the file without the password but the user himself must give the correct password (PASS) to modify the file. The second example allows other users to call up the file using

OLD and to list it. The last example is interpreted as no access permission without the password.

Access permissions may be appended to the default permissions using the SAVE command. Thus:

```
SAVE ;P
```

saves a file with RWALP, RWALP. The access permission with the SAVE command are summarised in Table 4.5.

Table 4.5.

When user types	Accesses granted	
	Without password	With password
SAVE	DEFAULT†	DEFAULT†
SAVE,password	none	DEFAULT
SAVE;accesses	DEFAULT+acc†	DEFAULT+acc†
SAVE,password;acc1,acc2	acc1‡	acc2‡
SAVE;octal accesses	first half	first half
SAVE,password;octal accesses	first half	first half

† For non-compiled files, DEFAULT accesses are RWAL; for compiled files, RWAXC or RWAC is assigned instead of RWAL, depending on the system. In Basic, RWAXC is used for compiled programs and RWAC for compiled files of subprograms.

‡ RWA must be part of one set of accesses, unless the user has Owner permission on the current catalog. By supplying the proper password, those in charge of sublibraries of the program library may have Owner permission on their own catalogs.

The REPLACE command can be used to specify access permissions in a similar way to the SAVE command. In particular the REPLACE command can be used to change accesses. The access permissions with the REPLACE command are summarised in Table 4.6.

Table 4.6.

When user types	Accesses granted	
	Without password	With password
REPLACE	UNCHANGED	UNCHANGED
REPLACE,password	UNCHANGED	UNCHANGED
REPLACE;access	DEFAULT+acc†	DEFAULT+acc†
REPLACE;pass;acc1,acc2	acc1‡	acc2‡
REPLACE;octal access	first half	second half
REPLACE,password accesses	first half	second half

† and ‡ see footnotes to Table 4.5.

As shown in Fig. 4.7, word 4 of a catalog entry is the access word. It contains information related to the operations which may be performed on the file or catalog. The access word contains two sets of accesses; the upper half-word represents those available without specifying a password and the lower half-word those available if a password is supplied. These two sets have different meanings when master-trap bits or slave-trap programs are used. See the descriptions in sections 4.5.4 and 4.5.5 respectively for details.

The access word may be considered an octal number; the 36 bits representing access information may be divided into two groups of six octal digits (the two sets of accesses). In the first group are the accesses that determine which operations can be performed on the file when no password is supplied. Usually, the second group of accesses determine which operations can be performed on the file when the correct password is supplied.

Each 18-bit group of access contains nine bits used by the Executive, and nine 'user' bits used by simple Monitor. Fig. 4.9 indicates the bit positions in the access word of a password-protected file.

Accesses without password (bits 0–17)						Accesses with correct password (bits 18–35)					
C T –	O F S X	A W R	L – C	P G	System code	– – –	O F S X	A W R	L – C	P G	System code
0	3	6	9	12	14	18	21	24	27	30	32 35

Bit	Octal value (bits 0–17)	Meaning for files	Meaning for catalogs
0	400000	–	C (catalog)
1	200000	T (slave trap)	–
2	100000	–	–
3	40000	S (shared)	O (owner)
4	20000	F (fetch)	F (fetch)
5	10000	X (execute)	S (search)
6	4000	A (append)	A (append)
7	2000	W (write)	W (write)
8	1000	R (read)	R (read)
9	400	L (list)	–
10	200	–	–
11	100	C (compile)	–
12	40	P (public)	–
13	20	G (group)	–
14–17	0–17	System code	–

Fig. 4.9 – Access word bit.

Several items should be noted. User bits (9 through 13) and system bits have no meaning for catalogs.

Bits 21-35 of the passworded accesses have the same meaning as their counterparts in the unpassworded accesses. Bit 18 is not used with password-protected files. (The Catalog attribute occurs only with unpassworded accesses.) Bit 18 has the special meaning that a file is trap-protected; bit 19 specifies the type of trap (0 master, 1=slave) (see sections 4.5.4 and 4.5.5).

The system code associated with a file is stored in bits 14-17 of the file's access word. Not all systems have an assigned system code; files saved by Simple Monitor while using such systems are assigned system code zero. For files having system codes which have no associated system (presently codes zero and twelve), the SYSTEM option of the CATALOG command prints **** for the system name. Files may be saved under the systems listed in Table 4.7.

Table 4.7.

System	Octal code (bits 14-17)
DATA	0
BASIC	1
ALGOL	2
FORTRAN	3
LISP	4
APL	5
FORTRAN 76	6
COBOL	7
SNOBOL	10
PL/I	11
Unassigned	12
CPS	13
DXPL	14
GMAP	15
SBASIC	16
DYNAMO	17

4.5.4 Master-trap programs

In order to protect system-related files, a capability based system of protection is used. When a user number is created a number of master-trap permission bits are assigned to it (bits 20-28 as shown in Table 4.3). These nine bits relate to file protection. The capability system is invoked only if the master-trap bit (bit 18) is set in the access word on a file. The inclusion of the master-trap bit in the access word has two effects. It changes the meaning of bits 27-35 of the file access word (see Figs. 4.9 and 4.10) and it specifies that any attempted access on the file will cause the master-trap program to be run. This program compares the access bits in the file access word (27 through 35). The job accesses are the logical and of (a) traps bits of file on which run was issued, (b) trap bits in job access mask specified on the RUN MME by supra-job, and (c) job accesses on subpra-job. If the job has all the bits specified in bits 27-35 of the file access word, the master-trap program grants the accesses on the right-hand side (bits

21 through 26) of the access word. If the necessary job access bits are not present, the accesses on the left-hand side (bits 3 through 8) are granted. Fig. 4.10 shows the one character abbreviation for the bit positions in the access word of a trap-protected file. The meaning of each is described in Table 4.8.

Standard accesses

As above (section 4.5.3)	+ T	O F S X	A W R	V M $	P B L	S ! X

0 17 18 21 24 27 30 33 35

Fig. 4.10 – Master-trap protection access word.

Table 4.8.

Bit	Octal value (bits 27-35)	Letter code	Meaning
27	400	V	Validation files. Protects validation files as one of nine bits. The reason for the protection by nine bits is the fact that, given the ability to change a validation file, a user may be able to obtain any of the nine master-trap bits. The VAL bit is used primarily to update highly privileged modules (LOGIN, TCFACE, OPMON, LOAD, DUMP, VAL) which need to run with all nine bits, but by itself grants no special privileges.
28	200	M	Master User Directory. Allows owner permission on any user catalog. The catalog MUD itself is protected more heavily and super-groups and groups within MUD are trapped in a fairly complicated scheme which maximises both security and flexibility.
29	100	$	Billing files. Protects billing files. This bit gives Read permission on billing files and programs; together with the MFD bit, it gives permission to alter billing files.
30	040	P	Peripherals. Protects PERCAT and device files.

Table 4.8 – *continued.*

Bit	Octal value (bits 27-35)	Letter code	Meaning
31	02	B	Background catalogs – BMC. Protects files which SIMON and other selected systems may retrieve from a non-control line (see below), but which are denied to non-control line users: for example :BAKCAT: SUBMIT and systems which may only be run by a monitor.
32	010	L	DLIBRARY. Protects files normally maintained by the program librarian.
33	004	S	SMSYSCAT. Protects all official systems. By itself, this bit gives Append permission on various systems catalogs; with the MFD bit, it permits updating official systems. No system module should be protected with more than the S bit, unless it specifically needs to run with more bits.
34	002	!	Master File Directory. Protects MFD and therefore gives Owner permission on catalogs when combined with other trap bits. By itself, this bit is worthless; it should not be used to protect anything expect as one of nine.
35	001	X	XSYSCAT. Protects experimental systems and anything else which systems programmers should be able to access (usually Read) from a non-control line.
27-35	777	9	Protects only MUD, validation files, and those few modules which need to run with all nine trap bits. Use of all nine trap bits should be rare.

In Fig. 4.10 the '+' (bit 18) specifies that a file is trapped. If both bits 18 abd 19 (+ and T) are set, a slave-trap program protects the file or catalog. If only + is set the file is master-trap protected.

Most users have no master-trap permissions and can do little to system related files. SIMON is a non-privileged monitor, having only P, B and X permissions. Other monitors are privileged and have more permissions. In particular

OPMON, the operation monitor, is a copy of SIMON but with all nine master-trap permissions. Since OPMON is so powerful, it must itself be protected. In order to access OPMON two conditions must be met. Firstly, the user must have the CTY bit in his permission word (see Table 4.3). Secondly, the user must be using a terminal on a line designated a control line (usually restricted to two terminals in the computer room or a secure office and thus subject to physical security). In a similar way the validation monitor, VALMON, may only be accessed by a user having both the NAN and CTY bits in his permission word. Thus the security of the system is provided by a multilevel protection scheme since a destructive user needs to know the user number and password of a very highly privileged user and must also be using a terminal in a secure location.

A description of the scheme used to protect systems, catalogs and files related to system maintenance is given below.

When only security is concerned, it is preferable to protect files rather than catalogs. Read and Search permission should be generally available. MUD, the Master User Directory, is a special case, since privacy as well as security is necessary.

Most systems catalogs have three sets of accesses:

(i) Those available to anyone (usually CSR).
(ii) Those available only to authorised personnel, possibly on a non-control line (usually CSRA).
(iii) Those available only to authorised personnel on a control line (COSRWA).

Everything is trapped, noting the possibility of giving all trap bits except V, M, B, and ! bits to non-control line users. Currently only XSY and PER are available to non-control line users.

To do this, the Master File Directory (MFD) is trapped CSR, OSRWA+! and other catalogs in the MFD are trapped CSR, SRA+<appropriate bit>. This results in a three-tier access system if all control line users had the MFD bit.

Note that if a user is able to obtain Owner permission on MUD (that is, he has all nine trap bits), all accesses are allowed on all structures below MUD.

MUD is a special case because it contains validation files. The ability to examine or alter validation files effectively gives access to all bits (assuming a control line is available). Therefore, consistency requires that these files be protected with all bits. Since the ability to create new groups and supergroups gives the ability to create validation files, MUD and supergroups must be Append-protected by all bits. Anticipating the possibility of password user catalogs with the password of the user number, Owner permission on groups must be protected by all bits. The substructure within MUD is:

:MUD	CSR	OSRA+9
:MUD:DEFAULT	R	RWA+9
:MUD:*X	C	SR+M
:MUD:*X00	CSR	SR
*X00000	COSRWA	OSRWA
Validation file	***	RWA+9

Optionally, some groups and user numbers may be saved as:

*X00000	COSRWA	FSRA (passworded user catalog)
*X00	CSR	FSR (passworded group)

The following permissions are available:

(i) With no bits and no passwords: SR on MUD, R on :MUD:DEFAULT.
(ii) With user catalog password: SRA on user catalog.
(iii) With group password: OSRWA on user catalogs in the group.
(iv) With the MUD bit: OSRWA on all user catalogs; SR on all substructures.

Catalogs in DLIBRARY are trapped in essentially two manners. Sublibraries which are maintained by persons other than the program librarian are saved:

CS[R] OSRA (passworded)

[R] indicates that Read permission is optional. Sublibraries maintained by the program librarian are saved with:

CSR [O]SR[A]+L

Note that this also results in a three-tier scheme; those catalogs saved CSR SR+L can only be modified by persons with the Owner (MFD) bit as well as the Library bit. Information catalogs in SMSYSCAT are trapped:

CSR OSRA+L

so that the librarian may alter files in them without the official systems bit which protects SMSYSCAT.

A typical accesses and protection system for files is as follows:

(i) Files which contain 'programs' (source, object code, or instruction input) are Write-protected by any trap bits that the program ever runs with. This may take the form of protecting the catalog in which the file is saved.
(ii) Files critical to the system are Write-protected by one of the following bits: MUD, VAL, or MFD. These bits should never be available from a non-control line.
(iii) Files are no more heavily protected than is necessary; no bit should be added without careful consideration.
(iv) Due to the Executive's attempt to enforce (i) above, a file which is executed must be directly protected by all the bits it ever runs with.

While it is impossible to examine all the possible kinds of files on the system, there are two main categories: system and information files.

It is possible to get a copy of any module on the system (to load under DDT, for example) given only the XSYSCAT bit, but official systems are protected from modification from non-control lines. Most modules are available without the XSYSCAT bit.

The following are typical ways of protecting systems:

(i) XC, RX+X. Official system where protection of 'trade secret' information is deemed more important than allowing non-systems programmers to load it under DDT. Note: This cannot be done for systems which 'go pure' (specify that a section of their code will not be modified during the system's execution).

(ii) RXC, RX+X. Official without 'trade secret' protection.

Note: The two preceding accesses without the C bit indicates Run permission is needed to be able to run the system.

(iii) RXC, RX+X[P][B]. Official system which requires peripherals or BMC bit. If any such system must be protected for proprietary reasons, a second copy should be saved Read-protected by only XSYSCAT permission.

(iv) [R]X[C], RWAX+X[P][B]. Experimental system (in XSYSCAT). Should have [R] and [C] if the official version does.

(v) Privileged systems and monitors: [R]X RX+bits with which the module runs. Copies might be made of those which are Read-protected with more than X.

Representative information files are billing files, library programs, explain files, etc. With a few exceptions, information files are protected by only a single trap bit, if that. Typical file accesses are:

***[L]	R+$	Confidential billing files
R[L]	R[L]	Library files
***	RWA+9	Validation files
R[L]	RWA+X	Experimental run-time packages

The overall trapping scheme is based on a few general concepts:

(i) The system should be secure. All files on which the system and its users depend should be subject to modification only by a properly validated user on a control line. Privacy of users should be similarly protected. Access to proprietary information (important object modules) and all systems resources should be protected by at least a password.

(ii) The protection system should be flexible. One should be able to create user numbers with limited but useful domains; almost no one should need all nine trap bits to get work done.

(iii) The system should be useable. Any use of the system which does not comprise system reliability, user privacy, or proprietary information should be allowed. Nothing should be protected without a good reason.

(iv) The protection system should be consistent. No user should be able to use his authorised trap bits(s) to get other unauthorised trap bits indirectly.

4.5.5 Slave-trap programs

Slave-trap programs provide a sophisticated means of protecting saved files; they allow users to associate with a file a personalised algorithm for determining permissible accesses. A slave-trap program is a directly executable file which is run by the Executive before any accesses are granted on the file being opened. The slave-trap program receives information about the job attempting to open the file for use in deciding which accesses to grant. Upon termination, the slave trap informs the Executive of the accesses to grant on the slave-trapped file. This feature was implemented to allow sophisticated file protection.

The following is a sample session in which a user saves two files, FILE1 and FILE2, which are to be slave-trap protected by the slave-trap program TRAPPROG. The trap program grants Read and Fetch permissions to any job requesting the file, but allow Fetch, Read, Write, and Append to user *C38734. It also records information about the conditions under which the slave trap was run. The trap program uses a subprogram in the library, SYSLIB***: TRAPSUB, to obtain the necessary information and grant the desired accesses.

The following sequence calls up the source for the slave-trap program and lists it:

```
OLD TRPSOURC
READY

LIST

TRPSOURC  05 OCT 77   10:57

100 Library "SYSLIB***:TRAPSUB"  'Insert utility program
110
120 Rem Get file name, password, user number
130 Rem accesses requested, access type
140
150 Call "STINFO": N$,P$,U$,R$,T$
160
170 Rem Set access to allow  Read, Fetch
180
190 Let A$="RF"
200
210 Rem Check for our favourite user
220
230 If U$<>"HDC38734" then 290
240
250 Rem Also give him Write, Append
260
```

```
270 Let A$=A$ $ "WA"
280
290 File #1:"*C38734:LOG"  'Open our output file
300 Print #1
310
320 Rem Save file name, password, user number
330 Rem for simplicity, password is not checked
340 Print #1: N$,P$,U$
350
360 Rem Save Access request, type, granted
370
380 Print #1: R$,T$,A$
390
400 Rem Record date and time
410
420 Print #1: DAT$,CLK$
430
440 Call "STTERM": A$  'Tell the Executive what to allow
450 End
```

Since a slave-trap program must be able to be executed directly, the source must be compiled and the compiled version saved as the trap program.

```
COMPILE

TRAPSOURC  05 OCT 77  11:08

COMPILATION SUCCESSFUL
READY

SAVE TRAPPROG;TRWA,RWA
READY
```

The most significant access specified in the preceding example is the 'T' (Slave-Trap) which, when placed in the unprotected accesses, as above, means that the file is a slave-trap program. No other accesses are necessary for the file to be a slave-trap program. Other accesses are only necessary if the slave-trap program is to be accessed in any other way (via the OLD command in Simple Monitor, for example). Read, Write, and Append permissions in the unprotected accesses allow the user to unsave or change TRAPPROG as necessary. This is useful while debugging the slave-trap program. When debugging is complete, the slave trap can be used to protect itself by resaving it with the accesses T,TRWA, specifying itself as the slave-trap program.

Next two slave-trap-protected files are saved. For clarity, password overprinting is not shown.

```
NEW FILE1
READY
```

```
SAVE, TRAPPROG;,TFRWA
READY

NEW FILE2
READY

SAVE,TRAPPROG;,TFRWA
READY
```

The access T on the right-hand side of the access word indicates that FILE1 and FILE2 are slave-trap protected. This gives their passwords special meaning. Instead of being the password associated with the file, they specify the name of the slave-trap program protecting them, which must be saved in the same catalog. The remaining accesses on the right-hand side are the maximum accesses which the slave-trap program may grant and this will be checked by the Executive. Unprotected accesses are those which will be granted without running the slave-trap program. They are left null (empty) here so that the slave-trap will always be run. It would, for example, seem reasonable to save a file with Read and List on the left hand, and TRWA on the right, to allow the file to be called up without running the slave-trap. However, due to the way Simple Monitor opens files, both R and X should be supplied on the left to avoid running the slave-trap program. It is also necessary to specify desired user bits on the left side, as they may not be included on the right (where they are master-trap bits).

Given below are the results when FILE1 is accessed by user C38734 and FILE2 by C77800; the output file is also listed. Note that FILE1, FILE2 and TRAPPROG are saved in user catalog C38734.

```
USER NUMBER--C38734,
NEW OR OLD--OLD FILE1,PWD1
READY

HELLO

USER NUMBER--C77800,
NEW OR OLD--OLD *C38734:FILE2,PWD2
READY

HELLO

USER NUMBER--C38734,
NEW OR OLD--OLD LOG
READY

LIST

LOG     05 OCT 77  11:37
```

```
FILE1          PWD1           HDC38734
FXR            OPEN           RFWA
10/05/77       11:20:10

FILE2          PWD2           HDC77800
PFXR           OPEN           RF
10/05/77       11:21:30
READY
```

Note that the 'P' in the list of accesses requested for each file specifies that a password was requested.

If a file is to be opened, and the requested accesses are not available from the unprotected accesses of the file, the slave-trap program associated with the file will be run, if one exists. There is one exception to this rule: programs with all nine master trap bits which have Owner permission on a catalog may be granted all permissions and bypass the slave trap. Slave-trap protection of a catalog is usually irrelevant as files saved in the catalog may be 'Fetched' without causing the slave-trap program to be run.

Due to the special manner in which a slave-trap program is run, it is passed limited information. The slave trap may, however, determine the following information about attempted accesses to a file:

(i) The name of the file being opened and the password supplied in the attempt to open it;
(ii) The accesses requested by the opening job, including password check request bit and the Fetch bit if Fetch is needed;
(iii) User number from which the open attempt is made;
(iv) Type of open (MME number).

To reduce the overhead caused by slave-trap programs, limitations are placed upon the actions which a slave-trap program may perform. A slave trap may use a maximum of one CPU second, may catalog a maximum of 2048 words, and may utilise a maximum of 10240 words of scratch storage. The limit on job size is 16K words of memory. Master-trap bits are limited to those possessed by both the opening job and the saved slave-trap program. The slave-trap program is passed one scratch file when run but no current catalog. In the case of BASIC slave-trap programs, this means simply that:

(i) PRINT statements will not appear on any teletype, but will be written to scratch storage (effectively discarded), unless the PRINT statement specifies the file number of a file which the BASIC program has opened using a FILE statement; and
(ii) The argument of FILE statements must specify tree names.

In any case where the slave-trap program does not run to completion, it is assumed that the slave trap disallowed all accesses. Once the trap has run, the unprotected accesses are not available; whereas for master-trap protected files, the unprotected accesses are granted if it is not possible to get the protected accesses due to insufficient job access permissions.

4.6 FILE SHARING

One of the advantages of a time-sharing system is the ability to share programs and data amongst users to the benefit of all. The previous section emphasised how to control file access and indicated that the default accesses on a file created by a user protected that file from all other users. However, if the file is saved with P, G or F then the file may be read by other users. Public files are made accessible by giving all users search permission on the catalog.

The reading of a file by a number of jobs at the same point in time is transparent to those jobs. Thus many jobs may have a file open for reading without interference since the Executive creates a file control block for each job using the file (see Fig. 4.3), and each read pointer will keep track of where each job is in the file. Unless a file is saved with shared permission it may be open by only one job for writing. If a job has a file open and another job attempts to open the file for writing then the second job will not be granted access. However, if a file is saved with shared permission then more than one job may access the file simultaneously including one or more jobs performing write operations. In such a situation, the integrity of the data read or written is a function of the various jobs using the file. This facility permits the implementation of multi-job transactions on a file at the users own risk.

4.7 FILE MIGRATION

In order to minimise the amount of disc storage required to run a DTSS system, a file migration system is provided which copies files from disc to tape. The migration system automatically copies files which have not been used for a system specifiable period of time, from disc to tape. A user may also request that named files of his own be migrated in order to reduce his use of catalog storage, using the MIGRATE command. All migrated files are catalogued in the users catalog and the CATALOG command indicates a file has migrated by enclosing the name in parentheses when it prints it. Migrated files can be recovered using the RECOVER command. Both the RECOVER and MIGRATE commands have options which allow the user to find out which files are waiting to be recovered or migrated, and to cancel the request to migrate or recover one or more files.

The migration system is in two parts. One part runs under SIMON in response to MIGRATE and RECOVER commands. This builds two files, one with a list of migration requests and one with a list of recover requests. The second part runs under OPMON. The restore module is run by the operator and processes the file of recover requests. The migrate moldue is run by the operator and does two things; it processes the migration request file and it runs another job which climbs through the catalog tree migrating unused files. This is shown in Fig. 4.11.

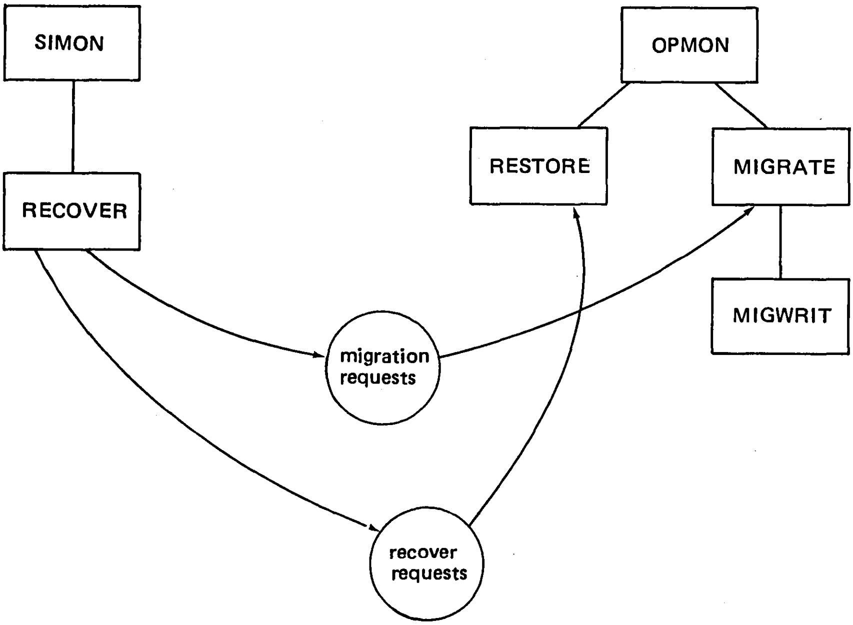


Fig. 4.11 – File migration system.

4.8 FILE SYSTEM BACKUP [7]

In order to minimise the effect of destruction of part or all of the filing system, it is useful to checkpoint the system at regular intervals. There are three types of checkpointing available. Full logical dump, where the entire catalog tree is copied from disc to tape; incremental dump, where those files and catalogs which have changed on or after some specified date are copied to tape; and selective dump, where individual branches of the catalog tree are copied to tape. The logical dump program used to do the checkpoint is called LDUMP and is run under OPMON. Since LDUMP needs all nine master-trap permissions to access all files, and since it is not desirable to give the operator all nine master-trap permissions on his user number for security reasons, OPMON treats LDUMP (and LOAD) in a special way. In order to run LDUMP the user must be on a control line and must have the operator bit set in his permission word. If these two conditions are met LDUMP is run by OPMON with all nine master-trap permissions regardless of the master-trap permissions of the user. Since running LDUMP (and LOAD) is the only time an operator needs all nine trap bits (typically an operator needs the MUD bit to load files from cards into a users directory, PER to operate tapes, and XSYSCAT to access experimental systems) this improves system security.

The logical load program to recover files created by LDUMP is LOAD. LOAD runs under OPMON and is designed to recover all or parts of the catalog tree from tapes produced by LDUMP. Thus LOAD may be used to carry out a full load of the entire catalog tree, or a selective load. This latter option is useful in the event of files lost through user error.

4.9 REFERENCES

[1] Pedersen, L. C. (October 1975), 'Basic DTSS File System Concepts', DTSS System Memorandum 214.

[2] Brinch Handen, P. (April 1970), 'The Nucleus of a Multiprogramming System', CACM, Vol. 13, No. 4.

[3] Elliot, B. (July 1977), 'DTSS Catalog Structure', Internal Memo.

[4] Fernandez, G., Mather, D., and Smith, L. (January 1977), 'The CATALOG Command', DTSS Technical Memorandum 041.

[5] Fernandez, G., Franklin, H., and Kaufman, C. (January 1976), 'File Accesses on the DTSS', DTSS Technical Memorandum 102.

[6] McGeachie, J. S. (March 1975), 'Validation Monitor User's Manual', DTSS Technical Memorandum 093.

[7] Keim, J. (November 1973), 'Logical Dump and Load', DTSS Technical Memorandum.

CHAPTER 5

Master-Mode Operating System

5.1 EXECUTIVE'S VIEW OF A JOB

As stated in section 2.2.1, the Executive provides a virtual machine for each job which consists of a slave-mode processor, a contiguous area of memory starting at location zero, access to a number of files such as the current file and the current catalog, and access to a clock. This represents the job's view of itself. The Executive's view of a job is held in a number of places – the Executive job table, Executive queues, and the job's state vector. The state vector was described in Chapter 4 and the Executive queues will be discussed in section 5.5. The main information held by the Executive about a job is contained in it's job table. The job table is typically one hundred and twenty-eight entries long, with each entry occupying sixteen words.

There is not a one-to-one correspondence between users and jobs (and hence job numbers). A user only generates a new job, and hence needs an entry in the Executive's job table, when SIMON does not contain the code within itself to carry out the service requested by the command. Thus whilst a user is typing in text to the current file, or issuing commands such as LIST, OLD, NEW, RENAME, SAVE, and REPLACE, only the single entry in the job table for SIMON, shared by all users running under SIMON, is required. Only when the user issues commands such as EDIT, CATALOG, and RUN is a new entry made in the job table; when the user returns to the ready-state following the completion of such a command the job number is returned to the Executive for re-use by another job. The twenty-two items (packed into sixteen words) in the job table for each job are as follows:

1. User number of user attached to this job.
2. Length of job.
3. Location of job in memory.
4. Requested length of a swapped-out job.
5. Length of job's state vector.
6. Location of job's state vector.

7. Requested length of job's state vector.
8. Job number of supra job.
9. Address of swapped-out job.
10. Scheduling number.
11. Time at which to give real-time timer trap.
12. Time until run-time timer runout.
13. Total accumulated run-time.
14. Job type.
15. Pure procedure information.
16. Count of number of I/O operations which must complete before job may be swapped (number of master holds).
17. Count of number of traps which must occur before continuing (number of roadblocks).
18. Link on run queue.
19. Head of queue of Executive tasks for this job.
20. End of queue of Executive tasks for this job.
21. Head of queue of locked-out copies.
22. End of queue of locked-out copies.
23. Flag to indicate the job needs service.

The first thirteen entries are fairly obvious and will not be discussed further. The job bits indicates the special privileges a job has and the maximum it may have. The eight privileges are:

- Monitor privilege indicating a job runs at monitor level and influences the scheduling of the job.
- Core resident privilege enables the job to be locked into memory and not swapped.
- Large state-vector privilege allows the job to have a larger than usual state vector.
- Special catalog privilege needed by login to create new user catalogs.
- Load/dump privilege needed by LOAD and LDUMP.
- PDQ privilege ensures the job runs at very high priority.
- Crash privilege means that if the job aborts the system should be crashed.
- Log to console privilege allows a job to send messages to the operators console; needed by background system to tell operator to mount tapes etc.

Item 15 relates to the feature of being able to specify that a contiguous section of a job's address space consists of pure procedure and hence need not be swapped out since a copy will exist on the swapping device. The master hold count is the number of I/O operations in progress to/from the jobs memory. The roadblock count is the number of traps to wait for before continuing the job. A job may be roadblocked without being master hold (for example, waiting for an open to complete) or it may be master hold without being roadblocked (for example, LDUMP writing to tape at the same time that it is running. Item 18 is used to hold linkage information when the job is linked into the run queue (see section 5.5). Item 19 and 20 are used by the Executive to maintain a queue of tasks for this job to be carried out when the job is next serviced. Items 21 and 22 are used by the Executive to maintain a queue of copies outstanding for this job. In general, the COPY MME (see section 5.2) will take place if the job is in memory or swapped out but will be suspended if the job is in the process of being swapped. This queue records copies which the Executive attempted to service during the swap. Item 23 is a flag to indicate that the job is waiting for service.

5.2 JOB-EXECUTIVE INTERFACE MECHANISM [1], [2]

When a slave job is in execution, it may perform any legal slave mode operations. Whenever a processor leaves slave mode, the Executive gains control. The way in which the processor leaves slave mode may signal the Executive that certain actions or tasks are requested by the slave job. The job cannot perform these tasks itself because it is in slave mode. Through the software features known as traps, slave faults, and special interrupts, the Executive may, more or less at its own initiative, convey information to the job. A slave mode fault vector at the beginning of each job aids in this communication.

A job actually running (executing instructions) is not affected by the Executive, except as noted below.

The base address register (BAR) limits memory references by the slave job. The base portion of the register is set according to the location of the job in memory, and may be reset by the Executive at any time that the job is interrupted (is not actually executing instructions). The length portion is set according to the length of the job, and changes only when the job issues a MEMORY REQUEST MME.

The timer register is set according to the needs of the Executive, and may change any time the job is interrupted. The slave job should not therefore depend on the setting of the timer register.

An I/O activity requested by the job may proceed concurrently with its execution. (I/O is requested with Executive calls, as described below.) A slave job should not reuse an I/O buffer until it has been notified by the Executive (by means of a trap) that the previously requested I/O has been completed on that buffer.

The Honeywell 6000 hardware provides two general methods for leaving slave mode: execute interrupts and faults. Execute interrupts cannot be generated by a processor in slave mode. Certain types of faults can be generated deliberately by a slave job; other types are due to outside causes. The action taken by the Executive will depend on the type of interrupt or fault, and possibly on the state of the job's registers and memory when the fault occurs.

Certain types of interrupts are normally transparent to a slave job. That is, the job is restarted after the interrupt with all its memory and working registers preserved, so that it need not even be aware that the interrupt occurred. Included in this category are all execute interrupts, connect faults, timer runout faults, and startup and shutdown faults. These events are not completely undetectable by the slave job, though. Any time a job is interrupted, including transparent interrupts, the Executive may take advantage of its opportunity to give a pending trap, slave fault, or special interrupt (see below). It may also modify the BAR or timer register as previously described.

If it is imperative to the slave job that not even these events occur, it may prevent all transparent interrupts by the use of the inhibit bit. This hardware feature prevents any of the transparent interrupts from occurring.

Slave jobs may request services from the Executive via the MME (Master Mode Entry) instruction. The address field of the MME instruction contains a code for the type of service to be performed, and the job's registers and memory contain any additional parameters necessary.

Executive calls are of two types: non-trapping and trapping.

5.2.1 Non-trapping Executive calls

The action requested by a non-trapping call is performed by the Executive before the slave job regains control. From the point of view, of the slave job, its action is instantaneous. The action may include the modification of the slave job's registers or memory, Examples of non-trapping MMEs are:

TIME OF DAY	The current time of day in ASCII is loaded into a pair of the slave job's registers.
JOB TIME	The running time for the slave job is loaded into a register of the slave job.
PAUSE	The slave job is suspended until n (one of the parameters of the PAUSE MME) traps have occurred.
MEMORY REQUEST	Provided the request is within the memory limit of the job, the base address register will be changed to reflect the requested size.

5.2.2 Trapping Executive calls

Trapping MMEs are used to request Executive services which may require a significant amount of time to perform, such as input/output requests. The contents of index register six (X6) specifies the address of a trap block. When the Executive responds to a trapping MME, it records the parameters associated with it, including the address of the trap block. The service requested is then initiated (if all the parameters were correct) and control is returned to the slave job at the instruction immediately following the MME. The slave job may continue processing, modifying any registers as necessary, and issue other trapping or non-trapping MMEs. When the requested service has completed, the Executive will notify the job by giving it a trap at the address specified in X6 at the time the MME was issued.

The following sequence of events occurs when a slave job is trapped upon completion of a trapping command:

(a) execution of the slave job is suspended;
(b) two words of status information are stored in the first two words of the block pointed to by the slave job's X6 when the command was issued;
(c) the current value of the slave job's instruction counter (IC) and indicator register (IR) are stored in the third word of the trap block; and
(d) the slave job is restarted at the fourth word of the trap block.

A trap block may contain any sequence of instructions starting in the fourth word. This trap routine should end with an return instruction whose address field points to the third word of the block in order to resume execution of the code interrupted by the trap.

Normally, the code from the fourth word onwards will interrogate the status information to determine the degree of success of the operation.

The general format of the status information is as follows:

Access (bits 0-8 of status word 1).

On access error (status 100), gives the access bit needed for successful completion of the command.

On OPEN, RELINQUISH, and REPLACE, gives the access with which the file is open.

On RUN, EXECUTE and CONTINUE, gives termination access of infra job.

Zero in all other cases.

Status (bits 9-17 of status word 1).

0	Successful.
20	Status was reset.
40	Quotas exceeded.
60	System out of storage.
100	Access error.
120	Busy.
140	A register parameter error.
160	Q register parameter error.
200	X0 parameter error.
220	X1 parameter error.
230	X2 parameter error.
260	X3 parameter error.
300	X4 parameter error.

320 X5 parameter error.
360 X7 parameter error.
400 Recoverable error.
420 Unrecoverable error.

File reference number (bits 18-35 of status word 1).

On OPEN, gives file reference number of opened file.

On copy type commands, gives device status for recoverable errors (status 400) or device copies.

Zero in all other cases.

STATUS WORD 2

Typically used to return the length of a file or the number of words transferred.

A PAUSE MME may be used by a slave job to wait for an outstanding trap to occur. All trapping MMEs (with adresses in the range 500100-500135) have corresponding MMEs (with addresses in the range 500200-500235) which function the same as their 100 series counterparts but also issue an automatic PAUSE for one trap. This facility alleviates the need for a wait routine in some non-multiprogramming jobs. The technique used by multiprogrammed jobs such as SIMON is discussed in section 6.3.

5.2.3 The ERASE Executive call

As a simple example of a trapping MME, the ERASE MME is described. This MME is used to delete a file/catalog from a specified catalog. The parameters required are:

X0 File reference number of initial catalog to search.
X1 Points to name of file/catalog to erase.
X3 Points to password (or is zero).
X4 (bits 0-8) must be zero (bits 9-17) trap bit mask.
X6 Trap location.
X7 Number of entries in tree name (if zero, then 1 is assumed).

The ERASE command attempts to open the specified file/catalog (with the specified password if X3 is non-zero and without a password otherwise) in the specified catalog with read, write, and append permissions. If it is successful it then destroys the file/catalog together with its catalog entry.

Catalog quota checks are suspended for the ERASE command, hence, it can be used to destroy files in catalogs whose quotas have been exceeded. Conflict checks with accesses currently held by other jobs having the given file/catalog open are suspended for all accesses except write. An ERASE command may therefore be used to destroy a file/catalog which is open, provided that it is not open with write access.

Upon completion of the command a trap occurs to the location specified in X6.

The status returns of the ERASE MME are given in status word 1 as follows (status word 2 is set to zero):

0 Successful: the file/catalog was erased.
2 Lockout: the file/catalog is currently in use with write permission.
3 The desired file/catalog was not found in the searched catalog.
4 Protection violation: the specified password was wrong.
5 Fail: job could not have obtained read, write, or append permissions.
6 Bad tree name: search permission was available on some intermediate catalog and the next entry in the tree name was not the last entry, but was the name of a file which was not a catalog.
7 Fetch error: either (a) an error of type 3, 4, 5, or 6 occurred and search permission was not available on the catalog in which the search was being performed; or (b) search permission was not available on the (N-1)st catalog and fetch permission was not available on the file.
11 ERASE disallowed: the name file/catalog is a device file.
120 Busy: another command is outstanding on the catalog.
200 X0 parameter error: X0 does not contain the file reference number of a catalog.
220 X1 parameter error: the pointer to the name is out of bounds.
260 X3 parameter error: the pointer to the password is out of bounds.
300 X4 parameter error: bits 0-8 of X4 were non-zero.
320 X5 parameter error: X5 was non-zero.
360 X7 parameter error: X7 was greater than 10 (maximum of 10 names in the list of entries pointed to by X1).
400 Recoverable error: an error occurred in reading the catalog.
420 Unrecoverable error: information in the catalog has been destroyed.

5.2.4 The EXECUTE Executive call

A more complex MME is the EXECUTE MME. A simplified description of its action is given below. The parameters for EXECUTE are:

X0 File reference number of source file (binary image).
X1 Length of scratch area to append to source file.
X3 Partitioning category (bits 12-17).
X3 Job type (bits 0-8).
X4 Maximum amount of memory job is allowed.
X5 Pointer to run list.
X6 Trap location.
X7 Pointer to 8-word block containing job's initial registers.
A Time allowed to job.
Q Job access mask.

The source file, which must be open with execute permission, is run as a new job for the amount of time specified and with the files in the run list open for it. The new job is started at location zero with the specified registers and with all indicators off except the zero indicator, which is on.

A job file is created for the new job, and its file reference number is returned in bits 18-35 of status word 1 of the trap block specified by X6 before execution of the creating job is resumed. The job file retains all the accesses possessed by the source file, and remains open until closed by the creating job.

The amount of memory with which the spawned job is run is determined as follows: the length of the source file is rounded up to a multiple of 1K (1024 words) and to this length is appended the length of the scratch area specified in X1, also rounded up to a multiple of 1K. The spawned job can issue a MEMORY REQUEST to change the size of its memory, but can request no more memory than the minimum of the amount specified in X4 and the amount of memory allowed to the job which issued the EXECUTE command.

Bits 0-8 of X3 are ANDed with the job type bits allowed to the job issuing the EXECUTE to form the permissions allowed to the spawned job. (See the description of the type bits in section 5.1 for a list of these permissions.) At the time the job is run, the large state vector and permissions are automatically enabled if they are allowed. The spawned job must enable itself for any other allowable permissions by issuing an ENABLE command.

The run list pointed to by X5 must be at least five words long and be terminated by a zero. The first two words in the run list are the identifying words for the spawned job. These words can be read by the spawned job and all jobs which are run below it by issuing a REQUEST STATUS on file reference number 0.

The third and fourth words of the run list limit the number of words that the spawned job can append to catalogued and scratch files. These limits should be less than the limits of the supra job. If they are not, the Executive will substitute the supra job's limits. The limits actually passed to the spawned job will be indicated to the supra job in the third and fourth words of the run list. If any file operation issued by the spawned job results in a situation where these limits are exceeded, then that operation will be trapped with a status return of 40 (quotas exceeded). The catalog and scratch word allotments for the spawned job are subtracted from the corresponding allotments for the spawning job. Upon termination of the spawned job, the job which ran it will have its catalog and scratch word allotments incremented by the allotments remaining for the terminating job.

The remainder of the run list contains a list of file/catalogs to be passed to the new job. A zero word indicates the end of the list. Each word in the list contains the following information:

Bits	0-8	Access to pass with file.
	9-17	Preference (for created file).
	18-35	File reference number or zero.

The nth word in this list causes the file/catalog with the file reference number specified in bits 18-35 to be passed to the new job with the access specified in bits 0-8. In the new job this file/catalog will have file reference number n. If the file reference number in bits 18-35 is 0, then a file will be created and passed to the new job. If bit 1 of the access is 0, then a scratch file will be created with the preference specified in bits 9-17 and passed. If the return bit is set in the access for a created scratch file then the file reference number of the file for the creating job will be returned in bits 18-35 of the run list entry before execution of the job is resumed.

Files which have been passed already or which are currently busy cannot be passed to the new job. The file reference number of a file which has been passed without the return bit set becomes invalid. Files which are passed with the return bits set become busy until they are returned by the spawned job (except for communication files). A file which has the return bit set in its access must be passed with the return bit set. A scratch file which is passed without the return bit set automatically has all accesses set for the new job. As opposed to files passed by the PASS command, no messages accompany files passed through the run list. The current setting of the read/write pointer for a file/catalog is preserved when it is passed.

The job access mask of the issuing job is ANDed with the job access mask specified in the Q register to form the access mask for the new job. The bits in this access mask have the following significance:

Bits 0-9 Not used.
20-28 Permission bits for master trap program.
29-35 Specify which preferences may be assigned when creating scratch file/catalogs.

The job file for a successfully spawned job remains busy until a trap occurs to the location specified in X6. The lower half of status word 1 contains the file reference number of the job file. Status word 2 contains the total running time accumulated by the spawned job and any jobs which ran below it. The trap occurs whenever the time allotted to the spawned job is exceeded, status is reset on the job file, or the job terminates or is aborted.

5.2.5 Trap-handling code

Most non-multiprogrammed slave jobs wait after issuing a trapping Executive call. The following sequence of instructions is typical:

```
EAX6    TRAP       Load X6 with address of trap block.
STZ     2,6        Clear trap block return word as flag.
MME     (type)     Issue trapping Executive call.
INHIB   SAVE,ON    Set bit of following instructions to inhibit interrupts
                   (whilst testing flag).
```

```
SZN      2,6        Check return word.
TNZ      *+4        If not zero, trap has sprung.
LDX5     1,DU       Load a pause count of 1.
MME      PAUSE      Wait for one trap, since this returns control to
                    Executive interrupts no longer inhibited.
INHIB    RESTORE    Following instructions will not have inhibit bit set.
TRA      *-R        Check flag again.
LDX0     TRAP       Pick up status word 1.
ANX0     =0777,DU   Mask status field.
CMPX0    GOOD,DU    Check for expected status.
TNZ      ERROR      Go analyse error.
  •
  •      remainder of program
  •

TRAP:  BSS 2    Two status words.
       BSS 1    Return word.
       RET *-1  Return immediately from trap.
```

This routine always waits after issuing an Executive call. Note that the trap routine returns immediately through the return word, and the main routine checks the status words. The return word is also used as a flag, since the Executive will not put the IC/IR into this word until it springs the trap.

Note that INHIB ON is not a machine instruction but an assembler directive which causes the inhibit bit to be set in assembled words following it up to an INHIBIT RESTORE directive. Thus if the TNZ *+4 causes a transfer it will be to an instruction without its inhibit bit set.

5.2.6 Fault handling

Since not all address fields of a MME instruction are defined and since it is possible to pass invalid parameters on an Executive call (such as a trap address of zero) the Executive needs a mechanism for informing the slave job that such a situation has arisen. Since the trap address may be the problem it is not possible to handle this through the status return in the trap block. Moreover, such faults are part of a large class of faults which a job can generate. To facilitate the communication of the occurrence of a fault to the slave job which generated it, each slave job has a forty (octal) word block, called the slave fault vector. This block occupies the first forty words of the slave job's memory, and is divided into two-word fault cells.

When certain faults occur, they are returned to the slave job which generated them as slave faults. To give a slave fault, the Executive stores the job's IC and IR registers at the time of the fault into the first word of the appropriate fault cell in the slave fault vector. The job's IC is reset to point to the second word of the fault cell. If the first word of the fault cell was zero before the IC/IR were stored, the procedure is complete. However, if it was non-zero, the job had a 'dirty fault vector'. In this case, the job is 'aborted'; that is, it is suspended, and the job which ran it is notified by a trap that the job had a dirty fault vector. The supra job then has the option of continuing or terminating the aborted slave job. This feature is intended to prevent an undebugged job from looping indefinitely due to a fault in its fault processing routine.

The second word of a fault cell typically contains a transfer instruction to the appropriate fault handling routine.

The following are returned as slave faults:

- Memory fault (generating an address outside the bounds specified by the current setting of the base address register).
- MME fault (invalid address field, invalid parameter, or MME issued by a job running in squeezed mode.
- Fault tag fault (a bit can be set in the isntruction word such that if the instruction is executed a fault is generated).
- Command fault (attempting to execute a master mode instruction in slave mode) (in special circumstances this generates a system crash).
- DRL fault (executing a DRL instruction).
- Lockup fault (running inhibited for longer than the time set by the Executive as the maximum for running inhibited).
- Illegal op code fault (executing an instruction for which there is no hardware definition).
- Operation not complete fault.
- Overflow fault (generating a number too large to be represented).
- Divide check fault (attempting to divide by zero).

Two software-simulated timers are available to a slave job. A run-time timer is set by the Executive call STIME, and a real-time (elapsed time) timer is set by the Executive call CLOCK. Both these calls are non-trapping. As soon as possible after either of these timers counts down to zero, it is disabled and a slave fault is given.

The first fault cell has a special use. Word zero is the initial entry point to the job. When the job is first run, its IC is set to zero. Word one, together with the fault cell at locations twenty/twenty-one are used for special interrupts which are discussed in section 5.3.

As noted in the MME fault, one possible cause of this fault is a MME issued by a job running in squeezed mode. This mode, obtained with a SQUEEZE MME, set a pseudo base address register (smaller than the setting of orginal base address register – hence the name) and also treats MMEs as faults, thus allowing the squeezing job to appear to the squeezed job as an Executive.

5.3 INTER-PROCESS COMMUNICATION [4]

Communication between a job and one or more jobs running beneath it in the job tree is achieved through pseudo-files called communication files. Communication files are an inexpensive way (since in general they require no physical input/ouput) for running jobs to communicate. The operating protocol for communication files resolves most of the synchronisation problems inherent in communication between independent processes.

The primary uses of communication files are:

- Communication between job and user terminals via TCFACE.
- Communication between LOGIN and the monitors and hence inter-monitor communication.
- Passing the user's command line by SIMON to jobs running on the edit interface.
- By the DO module, invoked by the SIMON PERFORM command, to simulate a user terminal.
- Communication between BAKMON and jobs running under it.
- By FORSIM to simulate normal files to the jobs running under it.

A communication file is a scratch file which is represented by file control blocks in the state vectors of those jobs holding the file open, and by entries in the Executive's table of open files. Each different file control block associated with a communication file is called an end. Each end has a file reference number and may be operated on as if it were an ordinary file. A single communication file may have any number of simultaneously valid ends, where normally each end is held by a different job.

A communication file may be created using the PASS MME or using the implicit pass invoked by the EXECUTE or RUN MMEs. The job creating the file obtains a file reference number for an end of the file which is called the master end. This end cannot be passed; it remains with the job which created it until the communication file is destroyed. The target of the pass may be any job running below the issuing job or the single job above it. The job which was the target of the pass will receive a file reference number for a slave end of the new communication file. The file reference number of the slave end is a function of the file reference numbers already in use by the receiving job at the time of the pass and bears no relation to the file reference number of the master end. A slave end of a communication file may be passed to an infra-job. Whenever a job holding a slave end passes the slave end, a new slave end is created for that communication file. It is a unique feature of communication files that operations may be issued and completed at any end even though the file has been passed to another job. Data exchanged on a comminication file is always between the master end and one of the slave ends. Thus it is not possible to transfer data from one slave end to another.

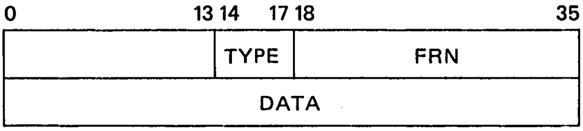
5.3.1 Special interrupts

The mechanisms employed to co-ordinate activities at each end of a communication file is called a special interrupt. Special interrupts are primarily used for communication file handling but are also used for passing and returning non-communication files between jobs and for notifying a job that the Executive attempted to give it a trap but found the trap address invalid (caused by releasing memory containing the trap block before the trap occurs). In general a special interrupt is used to inform one job of some action taken by another job. Thus a special interrupt differs from a trap or a fault in that a special interrupt does not normally result from an operation issued by the job receiving the interrupt whilst a trap and a fault does. Special interrupts are handled in a similar way to faults and use three locations (words one, twenty and twenty-one) in the slave fault trap vector.

If word one is zero, then the job is not accepting special interrupts and will never be given one. If word one is non-zero, then it is treated by the Executive as a pointer to an area of memory used as a stack.

When the Executive gives special interrupts to a job (in general, the Executive will give all the specials it has for a job at one time) it stores a pair of words of information on the top of the special interrupt stack, increments the stack pointer by two and decrements the tally by one. It then stores the jobs IC and IR registers in word twenty of the slave fault vector and transfers to word twenty-one, much like a fault. (The documentation calls this a connect fault since it corresponds in location to the connect fault cell in the hardware fault vector.) However, since a slave may not wish to be interrupted while it is in its special interrupt handling routine, a special convention is used if the Executive finds word twenty non-zero. Rather than generating a dirty fault vector abort as it does with faults, the Executive simply adds the special interrupt information to the stack but does not transfer to location twenty-one. By leaving location twenty non-zero the job may protect itself, but must examine the stack on exit from its special interrupt handling routine to process any specials which need processing. When no more specials require processing by the job it can set word twenty back to zero and future specials will cause the Executive to transfer to location twenty-one and hence back into the special interrput processing routine in the normal way. If the job's special interrupt stack is full when the Executive comes to give a job a special interrupt, the Executive will store the pairs of words in the job's state vector and give them to the job as soon as TALLY becomes non-zero.

The format of the two words of special interrupt information is shown in Fig. 5.1.



TYPE Type of special interrupt.
FRN Depending on the type of the special interrupt, this field may contain a file reference number. On drive specials, this field contains the FRN of the end of the communication file held by the receiving job. On file closed specials, the FRN of the former communication file end is given.
DATA The use of this field depends on the type of the special interrupt. For a drive special on a communication file, the lower half of the DATA field always contains the drive function data (from the lower half of the issuing job's A register). Other uses of the DATA field are listed below.

Fig. 5.1 – Special interrup pair format.

A typical routine for handling special interupts is given below:

```
       0: TRA        INIT
1 (SPTAL) : TALLDY     STAK,COUNT,2      SPECIAL TALLY
          .  .  .    .
          .  .  .    .      .
20 (SPRET):            ZERO              SPECIAL INTERRUPT
           TRA        SPINT             PROCESS SPECIAL INTERRUPT


    SPINT: SREG       SPREG             SAVE REGISTERS
           LDA        PSPTAL            GET PROTOTYPE TALLY
           STA        TSPTAL            SAVE IN WORKING LOCATION
           INHIB      SAVE,ON           MAINTAIN CONSISTENCY
   SPNEXT: LDA        SPTAL             ARE WE..
           CMPA       TSPTAL            ...DONE YET?
           TNZ        SPIN1             GO PROCESS NEXT PAIR IF NOT
           LDA        PSPTAL            WHEN DONE...
           STA        SPTAL             ...RESET REAL SPECIAL TALLY
           LDA        SPRET             GET RETURN ADDRESS
           STA        TSPRET            SAVE IT
           STZ        SPRET             ALLOW MORE CONNECT FAULTS
           LREG       SPREG             RESTORE REGISTERS
           RET        TSPRET            AND RETURN
           INHIB      RESTORE

   PSPTAL: TALLYD     STAK,COUNT,2      PROTOTYPE SPECIAL TALLY

    SPIN1: LDAQ       TXPTAL,AD         GET SPECIAL INTERRUPT PAIR
           CALL       SPROC             ***PROCESS SPECIAL***
                                        MAKE TSPTAL POINT TO NEXT PAIR
           TRA        SPNEXT            GET NEXT PAIR TO PROCESS
```

Notice that new word pairs can be added to the special stack once out of the inhibited portion.

5.3.2 Operations on communication files

Communication files are temporary entities, existing only as long as the master end and at least one slave end exist. If either the master end or all slave ends are closed, the communication file disappears. Thus the file reference number of an end of a communication file is an invalid parameter for many Executive calls dealing with the catalog structure, such as OPEN, ERASE, CATALOG, or REPLACE. Intended for communication, the ends of communication files cannot be 'run'. An attempt to do so will result in a parameter error status. Many other Executive calls are similarly restricted to files other than communication files.

The COPY Executive call and its variants, READ and WRITE, are used to perform the primary function of communication files: the direct transfer of data between jobs. Generally speaking, data which is copied 'into' one end can be copied 'out of' the other end. Data transfer is always initiated at a slave end. The valid Executive calls are COPY, READ, WRITE, DRIVE, PASS, CLOSE, REQUEST STATUS, RESET STATUS, TRUNCATE (or SCRATCH), and SET POINTER. Associated with each of the valid Executive calls is a special interrupt and the meaning of each is described in Table 5.1.

The only effect of TRUNCATE, SCRATCH or SET POINTER commands when issued from a slave end job is to release the reservation on the communication file (see below).

Slave issued set mode, slave issued drive, slave issued truncate, and slave issued set-pointer special interrupts simply inform the job holding the master end of a communication file that the corresponding operation has been issued at a slave end. The issuing job at the slave end is trapped immediately and no further action is expected of the master end job.

Slave issued read, slave issued write, and slave issued request-status special interrupts tell the job at the master end that the corresponding operation is outstanding at the slave end. That is, the issuing job is not trapped immediately. Normally, these operations are followed by a complementary operation at the master end. A WRITE at the slave end is complemented by a READ at the master end. READ and REQUEST STATUS at the slave end are complemented by WRITE commands at the master end. The length of the data transfer requested at the slave end is stored in the second word of the special pair by the Executive for the master end job.

Note that a COPY at the slave end with a pointer specified will generate two special interrupts at the master end. A slave issued set-pointer special interrupt will precede the slave issued read or write special interrupt.

A slave issued reset special interrupt tells the job at the master end that a slave operation which has not yet been completed is no longer outstanding due

Table 5.1.

Type	Mnemonic	Use
0	STM	Set mode – used to notify the job holding the master end of a communication file that a slave end job issued a set mode DRIVE. The lower half of DATA contains the mode. A set mode DRIVE is used to specify directions for the handling of data to be transferred through the communication file. In particular, for terminal communication files, it is used to set input and output modes. The DRIVE functions which may be issued on the slave end of a communication file to TCFACE are: 000000 Set line-by-line input mode. 000001 Set file building input mode. 000002 Set file building mode for paper tape. 000003 Set direct input mode. 000004 Set command input mode. 000005 Set command mode for paper tape. 000007 Answer back drum request. 000020 Set full duplex output mode. 000021 Set half duplex output mode. 000022 Set direct output mode. 000023 Reset direct output mode. 000024 Set friden output mode. 000025 Reset friden output mode. 000026 Stop output. 001000 Output any currently pending warning.
1	RCF	Read communication file – used to notify a slave end job that the master end job has data which the slave end should read. The master end job may not simply write directly into the communication file, since only a slave end may initiate data transfers. The lower half of DATA contains the length of the data to be read.
2	SRS	Slave issued reset – notifies the master end job that a slave end job issued a RESET STATUS on a COPY to which the master end job had not yet responded.
3	BRK	Break – notifies a slave end job that the master end job has issued a break DRIVE. The typical use of this is with a terminal communication file, to indicate the receipt of a break signal from the terminal.

Table 5.1 – *continued.*

Type	Mnemonic	Use
4	PF	Passed file – notifies the job that a (communication) file has been passed to it via a PASS Executive call. This may be the result of creating the communication file or passing a slave end. The lower half of DATA contains the FRN of the job file of the passing job.
5	RF	Returned file – notifies the job that a (communication) file which it passed with the return bit set has now been returned to it. The lower half of DATA contains the FRN of the job file to which the file being returned was passed.
6	FCLO	File closed – notifies the job that a communication file of which it held an end has been destroyed because a CLOSE was issued at another end.
13	R	Slave issued read – notifies the master end job that a read-type command was issued at the slave end. DATA contains the requested length. The master end job should issue a WRITE to transfer the data to the communication file.
14	W	Slave issued write – notifies the master end job that a write-type command was issued at a slave end. DATA contains the requested length. The master end job should issue a READ to transfer data for the communication file.
15	RS	Slave issued request status – notifies the master end job that a REQUEST STATUS was issued at a slave end. DATA contains the requested length of the status block. The master end job should send status information using a WRITE command.
16	T	Slave issued truncate – notifies the master end job that a TRUNCATE or SCRATCH command was issued at a slave end. DATA contains the requested truncation length (zero for SCRATCH).
17	SP	Slave issued set pointer – notifies the master end job of a SET POINTER command issued at a slave end. DATA contains the requested pointer setting.

to a RESET STATUS issued on the slave end. This special interrupt is generated only if there was an operation outstanding. The communication file is left in a busy state at this point, preventing further slave end operations until the job at the master end issues a RESET STATUS on the file.

Once a job has passed its slave end, it may not close or pass it again until the newly created slave end is returned. From this it follows that there can be at most one end of a given communication file present at any one level of the job tree.

Moreover, only the lowest slave end can be closed. When the lowest end is closed, that end disappears and the job holding the next lowest slave end is notified by a 'returned file' special interrupt.

When the last slave end of the communication file is closed, the job at the master end (the creator of the file) is given a file closed special interrupt. A file closed special interrupt is also generated at all slave ends if the master end is closed first.

5.3.3 The no-trap option

In order for data to be transferred between the ends of a communication file, the master end job must issue a COPY (or READ or WRITE). The normal action taken by the Executive is to transfer the data and then to trap the COPY at both ends involved.

Consider what happens when a user calls up a file that is so large that it will not fit into memory and then types LIST to SIMON. SIMON will simply issue a COPY specifying the current file as the source and the user's terminal (a communication file slave end) as the destination. This will generate a slave issued write special interrupt at the master end, held by TCFACE.

TCFACE needs to be able to read a portion of the file into its memory, transfer it to the 716 and then repeat when the 716 has printed the data on the user's terminal. This facility is implemented with the 'no-trap' option.

Normally, when the length specified on a slave end WRITE is greater than the length specified on the complementary master end READ only the minimum of the two lengths is actually transferred and the slave end is trapped with a 'destination exhausted' status. The master end operation is trapped with a good status. This rule is generalised for other complementary copies with mismatched lengths. For example, if the job at the master end attempts to read more than was written by the slave, it will be trapped with a 'source exhausted' status and the slave end operation will be trapped with a good status.

If, however, the no-trap option is specified for the COPY issued at the master end, then the slave end will not be trapped if the length of the master end COPY is less than the length of the slave end COPY. The amount of data specified by the job at the master end will be tranferred. The next COPY issued at the master end will pick up where the previous one left off. The slave end

COPY will trap when the number of words specified on the slave end COPY have been completely transferred.

Thus the master end job can satisfy the slave end's request in a piecemeal fashion.

5.3.4 Reserving the communication file

It is sometimes desirable for a job at one end of a communication file to complete a sequence of operations on the file without danger of interference from the other ends. For example, one slave end job might wish to write a message to the master and then read a reply, without the possibility that another slave end might 'sneak in' between its operations and read its reply from the master end. The process of reserving the communication file allows for uninterrupted sequences of operations. Either a slave end or the master end may reserve the file, but only one slave end can reserve it at any given time. Reserving by a slave end has the effect of locking out all other slave ends from the communication file. Only local operations can be issued at the other slave ends. When the master end reserves the file, all slave ends are locked out.

Reserving is specified by setting the appropriate flag bit when issuing those commands which allow communication files to be reserved. A slave end may reserve a communication file on a COPY, READ, WRITE, or DRIVE command. The master end may reserve the file only with a RESET STATUS command. Once the file has been successfully reserved, it remains reserved until an operation is issued at the reserving end which does not call for continued reservation. Any operation which does not explicitly specify reserving will release an existing reservation (by the same end), except that a REQUEST STATUS issued to the master end will not release an existing master end reservation. Note that both a slave end and the master end may have the file reserved at the same time, providing the slave end reserves first. (If the master end reserves first, the slave end will be locked out and will not be able to issue the reserving operation.)

The master end is not restricted by a slave end reservation. When the master end has reserved the file, it appears busy to all slave ends. This gives the master end job the ability to 'squelch' a runaway slave end job.

5.3.5 Special interrupt routing

When the master end job issues a DRIVE Executive command a special interrupt is generated at a slave end. Since there may be more than one slave end the question arises as to which slave end to give the special interrupt to. It was stated earlier that if word one of the slave fault vector is zero, a job will not receive special interrupts. A number of jobs, either for their entire life or for short periods, do not accept special interrupts. Jobs may choose to not accept special interrupts for the good reason that they have no need to. Thus, if a user types a carriage-return to a running job which is not waiting for input, the running time for the job is returned to the user's terminal as a formatted sequence of ASCII characters. Rather than have every job carry code to respond to the 'read communication file' special interrupt which TCFACE generates in response to this, and to generate the formatted string, SIMON undertakes this task (for jobs not accepting special interrupts). Another reason for not accepting special interrupts is that certain sections of code are time critical and one way of ensuring no interference while in a time critical section is to not accept special interrupts for that time.

The rules for the selection of the slave end to receive a special interrupt generated by a DRIVE Executive command issued at by the master end job are as follows. (The only other special interrupts given to a slave end job are 'passed file', returned file' and 'file closed'. Passed-file specials to jobs not accepting special interrupts are discarded. Jobs not accepting special interrupts cannot be informed of returned files and such special interrupts are discarded. If a communication file is destroyed and a slave end job is not accepting special interrupts the only way it will find out that the communication file has gone, is by receiving a parameter error on an Executive command involving the file.)

In the case of a read communication file special interrupt, if the communication file is reserved to a slave end, the special interrupt is given to the job holding that end if the job is accepting special interrupts. If the communication file is not reserved to a slave end the special is given to the lowest job which is accepting specials. If no job meets these criteria, the master end job has its DRIVE trapped with a status of 'other end not accepting specials'.

In the case of a break special interrupt, if no slave end has reserved the communication file, the special is given to the lowest job in the job tree which holds a slave end open with break permission (since communication files may not be executed, the execute permission bit is used to signify break permission for communication files) and is accepting special interrupts. If a slave end has reserved the file, the special is given to the lowest job meeting all these conditions and which is also not lower than the job holding the reserving end. If no job can be found meeting these criteria, no special interrupt will be given, and DRIVE command is trapped with the status of 'other end not accepting specials'.

Thus the routing of special interrupts is also affected by slave end reservations. This ensures that slave ends which are locked out of the file do not receive special interrupts to which they are unable to respond. The general rule is that if a slave end has reserved the communication file, special interrupts generated by the master end DRIVE commands will be given to the job holding that slave end, or not at all. The only exception comes in the case of a break drive. In this case, if the reserving end does not have break permission, the break special 'bounces' up the communication file until it finds a slave end which does have break permission. Thus a reserving slave end job is not only guaranteed that no other slave end is stealing its data, but also that its special interrupts are not being led astray.

Suppose that job A has run job B, which has in turn run job C. The portion of the job tree is shown in Fig. 5.2.

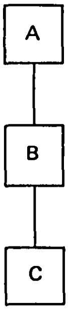


Fig. 5.2 – Job tree example.

Job A now issues a PASS command to job B, specifying a file reference number of zero (create-type PASS) and access bits for read, write, append, break (execute), and communication file. A communication file is thus created. Job A receives the file reference number for the master end (suppose it is 5) in the trap of the PASS, and job B receives the file reference number of the slave end (suppose it is 7) in a 'passed file' special interrupt. The file is now idle.

Upon receipt of the 'passed file' special interrupt, suppose job B wants to get more information about its new file. It therefore issues a REQUEST STATUS to its file reference 7 (the slave end), requesting ten words. A special interrupt (slave issued REQUEST STATUS) is generated at job A, specifying a length of ten words and file reference number 5 (job A's end). Job A then issues a WRITE to FRN 5 (the master end) for four words, placing a standard status block in the first three words and a special identifying code in the last word. These four words are presented to job B as its status block. Job A's WRITE is trapped with a status of 'successful', since all the data that job A requested were transferred. But job B has its REQUEST STATUS trapped with a status of 'source exhausted', since only four of the ten words it requested were available.

Suppose now that upon examining the data received on its REQUEST STATUS, job B decides the job C should have the slave end. It therefore issues a PASS to job C, specifying FRN 7 (its slave end) and access bits for read, write, append, return, and break.

Job C receives a passed file special specifying its FRN for the slave end (suppose it is 3).

Suppose job C immediately issues a RUN on a new job, specifying FRN 3 as the first file in the pass list, with access bits for read, write, append, and return. Job D is thus run with a slave end as its file reference number 1. Since both job B and job C set the return bit, they retain slave ends. The situation is shown in Fig. 5.3.

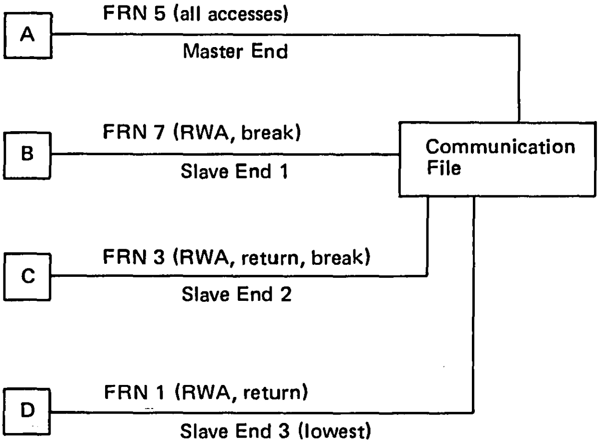


Fig. 5.3 – Communication file example.

Job D now issues a COPY command from its memory to FRN1 for twenty words. A slave issued write special is generated at job A specifying FRN5 and a length of twenty. Job A issues a READ for ten words, specifying the no-trap option. The first ten words are then transferred from job D's memory to job A's memory, and job A is trapped with the 'successful' status. Job D is not trapped, since the master end job specified the non-trap option.

Suppose job A now issues another ten-word READ, not specifying the no-trap option. The last ten words are copied from job D to job A, and both jobs are trapped with the 'successful' status. The communication file is now idle again.

Suppose job C now issues a COPY from its FRN3 to some other file (its FRN4, for example), for five words. Job A receives a 'slave issued read' special. If job A then issues a WRITE for ten words, specifying the no-trap option, the first five words will be transferred from job A's core to job C's FRN4. Job C will then be trapped with the 'successful' status, while job A is trapped with the status of 'destination exhausted'. The no-trap option had no effect, since the status presented at the master end was not 'successful'.

Suppose job D now issues a READ to FRN1. Job A is notified with a slave issued read special interrupt. However, job A decides that a break drive should be given. It issues the DRIVE command. The DRIVE is rejected with the 'inappropriate' status because the slave end has issued a copy-type command. Job A therefore issues a RESET STATUS requesting that the communication file be reserved. Job D's READ is trapped with a 'status was reset' status, and the file becomes reserved to the master end. If job D attempts to reissue the READ, it will receive the 'communication file busy' status. Job A now issues the break

DRIVE. The Executive discovers that job D does not have break permission on its end of the communication file. Assuming job C is accepting specials, it will receive a 'break special' and job A will be trapped with a 'successful' status. The file becomes idle, since job A's DRIVE had the effect of releasing its reservation. If job C issues a set mode DRIVE requesting reservation of the file, job A receives a 'set mode' special interrupt and the file is now reserved to job C.

Suppose job D now runs a new job (job E), passing it the slave end with the return bit set. It may not pass break permission, since it does not have break permission to pass. The situation is shown in Fig. 5.4.

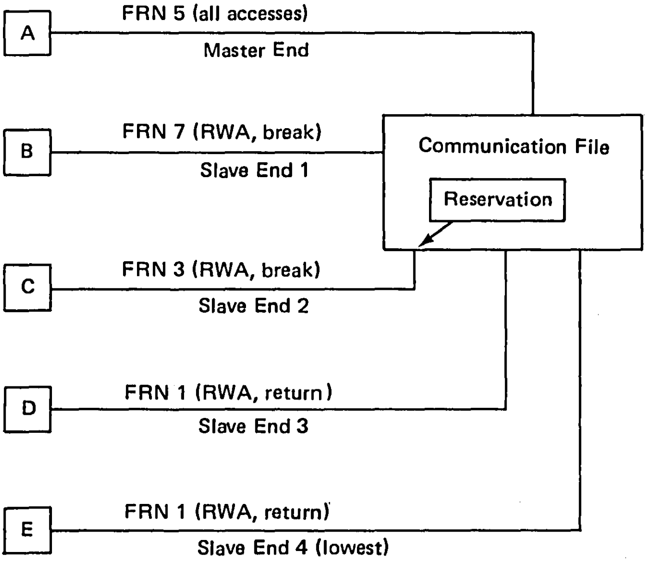


Fig. 5.4 – Communication file reservation.

Job D's implicit PASS is legal, since it is a local operation. Any non-local operation attempted by job D would be rejected, since the file is reserved to job C's end.

Similarly, if job E now attempts to issue a REQUEST STATUS on the file it was just passed, it will receive the 'communication file busy' status.

If job A now issues a 'read communication file' DRIVE, the special interrupt will be given to job C (assuming it is accepting specials). This will be true even if job E or D is accepting specials, since the file is reserved to job C's end. Suppose that job C, on receipt of the 'read communication file' special, issues a READ command, not specifying reservation. The reservation is released, and the file becomes busy until job A issues the corresponding WRITE command. When it does so, the data is transferred, and the file becomes idle.

Job A now issues another 'read communication file' DRIVE. Since no slave end has reserved the file, the Executive attempts to give the special interrupt to the lowest slave end (at job E). However, suppose job E is not accepting specials. In that case, the 'read communication file' special interrupt will 'bounce' up to job D. Assuming job D is accepting specials, it will receive the special interrupt.

Suppose job A issues a RESET STATUS to reserve the file. It then issues a CLOSE. Before the Executive can process job A's CLOSE, suppose that job E also closes its end of the communication file. Job D receives a 'returned file' special interrupt due to the closing of job E's end. This special is closely followed by a 'file closed' special as its FRN 1 is changed into a non-file by the master end CLOSE. Jobs B and C still hold slave ends of the files, and jobs B and C receive 'file closed' special interrupts. No more slave ends exist, so job A's CLOSE is trapped. Job A's FRN 5 becomes invalid.

Jobs B, C and D are still left with non-files. Jobs B and D issue a CLOSE to their FRN 7 and 1 respectively, and their non-files disappear. Job C, though, being a little slow to process its file closed special, attempts attempts to issue a set mode DRIVE to what used to be its slave end (FRN 3). The DRIVE is rejected with a parameter error status, since DRIVES are not legal on non-files. Job C finally CLOSEs its FRN 3, and the last trace of the communication file disappears from the system.

5.4 INTERNAL STRUCTURE OF THE EXECUTIVE [3]

The Executive consists of seven major modules together with a number of queues. The inter-relationship of the modules is shown in Fig. 5.5.

The non-control Executive is so called because it is the part of the Executive which runs in the non-control processors in a multiprocessor system (it also runs in the control processor). The primary functions of the non-control Executive is to respond to all slave generated faults and Executive calls. The non-control Executive deals with the slave generated faults but does not provide the services requested by the Executive calls, rather it calls the control Executive by generating an interrupt in the control processor to be responded to by the control Executive. The non-control Executive takes jobs from the run queue (a queue of jobs waiting for a virtual machine) and runs them for the length of time determined by the control Executive. When a job requires the attention of control Executive for an Executive service, the non-control Executive puts the job on the processing queue (see section 5.5).

The non-control Executive also responds to interrupts from the control processor if the non-control processor is idle (since there are no jobs to run) and the control Executive has put a job on the run queue. All other executive routines run only in the control processor.

The job service routine either provides the services requested by valid MME instructions or branches to other routines to provide them.

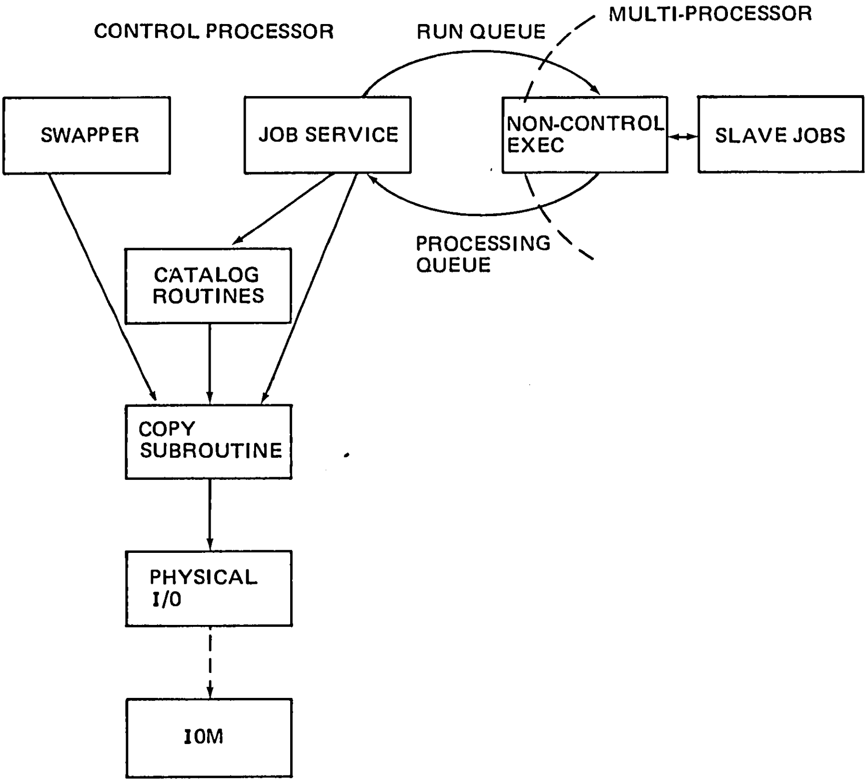


Fig. 5.5 – Structure of the DTSS Executive.

The catalog routines are called by job service to perform various catalog manipulation functions such as opening a file, saving a file, replacing a file, and unsaving a file. Since the catalog routines manipulate files, in order to maintain the integrity of the filing system only one task (MME) is carried out at a time. To ensure this each task is put on the catalog queue. This queue is a queue of tasks rather than jobs since one job may issue many MMEs, one after another, and hence have many entries on the catalog queue.

Since everything on DTSS is a file, almost all significant operations involved the movement of data from one file to another. Thus the copy routine which implements this is central to the whole system. The COPY MME has the following parameters:

X0 File reference number of source file.
X1 Points to pointer to starting location M1 in source file (used since index registers are only 18 bits long and a 36-bit pointer is required).
X2 File reference number of desination file.

X3 Points to pointer to starting location M2 in destination file.
X4 Flag bits.
X6 Trap location.
X7 Points to number of words N to copy.

N words in sequence are transferred from the source file starting at word M1 to the destination file starting at word M2. All other words of the destination file are unaffected. Read permission is required on the source file and write permission is required on the destination file if any existing portion of the file is to be modified; append permission is required if the file is to become longer.

During the actual copy, if the end of the source file is reached, an end-of-file condition occurs and data transmission stops. If any of the words transferred to the destination file are outside the range of that file, the file is extended if it is open with append permission. Otherwise, an end-of-file condition occurs and data transmission stops. Upon completion of the copy, a trap occurs to the location specified in X6.

If X0 (X2) is zero, then the source (destination) file is assumed to be the job file consisting of the issuing job's memory with read and write permission set. If X1 (X3) is zero, then the current position of the source (destination) file's read/write pointer is used. Both X0 and X1 (X2 and X3) cannot be zero at the same time. Upon completion of a copy, the read/write pointers for both files are updated to point to the word following the last ones accessed.

If the source (destination) file in a COPY is a device file, then the destination (source) file must be a core file, or a communication file whose other end is being copied to or from a core file.

Upon initiation of a COPY into a scratch (catalogued) file which is open with append permission, the job's scratch (catalog) word allotment is decremented by the amount by which the destination file may be extended. At the completion of the COPY, the job's scratch (catalog) word allotment is incremented by the number of words charged to this allotment but not appended to the destination file.

If the status return in status word one is not between 100 and 360 on completion of the COPY, then status word two contains the difference between the number of words transferred and the number of words requested (that is, it contains the negative of the number of words not transferred). If the status in word one is between 100 and 360, then status word two is zero. If either the source of the destination file is a device file, then the lower half of status word one contains the status return from that device.

If a job file is involved in the copy operation, it is locked in memory for the duration of the copy.

The swapper, the physical input/output routines and the interrupt routines are discussed in sections 5.7 and 5.8.

5.5 THE EXECUTIVE'S QUEUE STRUCTURES

The Executive maintains a number of queues on which appropriate jobs are held. Not all jobs are on a queue at any given time. There are six major job queues whose functions are discussed below. In addition to the job queues, the Executive also maintains a number of task queues which are also discussed.

The space required to maintained the queues and for other functions of the Executive requiring temporary space is provided by a block of memory composed of areas called list elements. This block of memory overlaps and area set aside for I/O buffers. When the Executive requires a list element it calls a subroutine (GET) indicating the number of words required. The GET routine returns the address of a list element of the requested length. Each list element carries with it a prefix word which contains the length of the element. Thus n-word element uses n+1 words.

As an example of the use of list elements, consider what happens when a job issues a COPY MME. In order that the job can proceed following the MME, the Executive requires space to store the parameters passes on the COPY. It then needs space to pass parameters to its COPY subroutine. The COPY subroutine requires space to hold copies of the file control block involved and finally, space is required by the Physical I/O routines. Thus the sequence would be:

```
GET n1         'to save MME parameters
GET n2         'for COPY control blocks
GET n3         'for (2) FCBs
GET n4         'for physical I/O
RELEASE        'when I/O finished
RELEASE (2)    'both FCBs
RELEASE        'on exit from COPY subroutine
               'status returned in MME list element
RELEASE        'after passing status and giving trap
```

The free list of elements is kept in memory address order to speed up the coalescing of the elements. However, most requests for lists elements are for elements whose size is less than twenty-four words. In order to speed up the majority of GETs, the quick get area is maintained as twenty-four stacks, each of which holds free list elements of exactly n words, n=1,2,...,24. Thus if the Executive requires a list element of twenty-four words or less it uses the QUICKGET routine which looks at the quick get stack for the number of elements requested and if it is not empty, returns the requested element. If the stack is empty then the routine invokes GET and a list element is found from the general pool.

If the list element block is full (that is, there are no free list elements) then since this indicates a heavily loaded system, special action must be taken. In particular, a system crash should be avoided. To handle this, a section of the list element block is set aside for just such a situation – the panic area. When the list element block becomes full (that is, when the GET routine is unable to get a requested list element) a global semaphore is set to stop any more jobs running and the panic block is appended to the list element block. When a certain amount of list element space has been returned as a result of job service, the semaphore is reset. At some time in the future, when all the list elements have been returned to the panic block, the panic block is removed from the pool.

5.5.1 The run queue

The run queue is the list of slave jobs which are in memory and waiting to use a processor. The jobs on the run queue are held in scheduling number order (see section 5.7). Whenever a processor is free to run a job it takes the job on the front of the run queue and runs it, removing it from the queue so that any other processor will not find it and run it. Associated with each job on the run queue is a quantum run time which is recorded in two words in the job's state vector. When a job is first entered into the run queue the two words are set to be the same (300 ms is a typical value). When the job runs, a copy of the second word is loaded into the timer register. The job is then run until either the timer runs out or the job issues a MME instruction or faults. In either case, the job is requeued on to the processing queue.

5.5.2 The processing queue

This queue contains all jobs waiting for processing by the Executive. It is accessible by all processors on the system and is protected by a semaphore. Jobs are taken from the processing queue by the job service routine in the control Executive and placed on the job service queue. The processing queue is processed on a first-in-first-out basis.

5.5.3 The job service queue

Jobs joining this queue from the processing queue require one of three types of service. If the job's timer has run out, the Executive charges the job for the resources used, reallocates a quantum, checking that the running time for the job has not been exceeded, increases the scheduling number according to the resources used, and requeues the job on the run queue. If the job issued a MME instruction or faulted, the Executive carries out the appropriate action, leaving the job on the job service queue during this time. On completion of the MME service (but not necessarily the MME itself since this may involve lengthy I/O) the job is requeued to the run queue if the MME was a non-pause type, or simply taken out of the job service queue if it is to pause.

Jobs are also added to the job service queue when the Executive needs to trap a job or give it a special interrupt. Such jobs which are on the run queue or are in memory but not on any queue at the time are placed on the job service queue. (The service they require is to have an entry on their Mlink queue processed, which is described below.) All other jobs which need to be trapped or given a special interrupt are so marked in the job table entry for the job. Those out of memory are placed on the non-core queue, to bring them in memory. Those in memory are so treated that they will return to the job service queue for some other reason in the near future. For example, if the job is running it will go to the processing and hence the job service queue in due course, if it is on the processing queue it will be moved to the job service queue next, if it is in the process of being swapped in it will be put on the job service queue since all jobs are so transferred immediately after being swapped in. Thus jobs on the non-core queue (see below) waiting to be swapped in will also, ultimately join the job service queue. Jobs in the process of being swapped out are tested immediately the swap out completes to see if the Executive wishes to trap them or give them a special interrupt, and if it does the job is requeued on the non-core queue.

5.5.4 Free job number queue

All jobs are referenced by the Executive by job number. When a job is initiated it is allocated an arbitrary job number from the pool of free job numbers. When a job terminates, its job number is returned to the pool. The free job numbers are kept in the free job number queue in no particular numerical order (since the job number value is of no significance).

5.5.5 In-core queue

This queue records the job numbers and location and size of all jobs which are in memory. The ordering of this queue is not significant. This queue is used by the Swapper as described in section 5.7.

5.5.6 Non-core queue

This queue holds all those jobs which are waiting to swap in. The queue is in scheduling number order and is used by the swapper to determine which jobs to swap in, as described in section 5.7.

5.5.7 Task queues

The Executive maintains a number of so called task queues. A task is an Executive process or a subprocess of a job. It is not a job in that it does not have a job number. The queues are used to record tasks which need to be done but which cannot be done at the present moment. The master task queue is used by the Executive to record tasks generated by other tasks. For example, the Executive may be processing a task when an input/output interrupt occurs. The interrupt is analysed and the task required to service the interrupt identified. The interrupt service routine (task) is queued on the master task queue for later processing. Similarly, in processing a MME from the job service queue, the Exective may discover it requires the catalog routines. This will result in a call to the catalog routines. If the buffer space required by the catalog routines is not available the task will be queued by the catalog routines, on the catalog queue.When an operation in the catalog routines completes, realeasing the buffer space (typically via an interrupt, whose service routine will first be queued on the master task queue) the return from the catalog routine is put on the master task queue and the catalog scanned to see if the catalog routines should continue processing or not.

There is a similar queue for buffer space for the copy subroutine.

To illustrate the sequence of events, Fig. 5.6 shows the action following a slave job executing an OPEN MME. In all cases in the diagram, the simplest situation is shown.

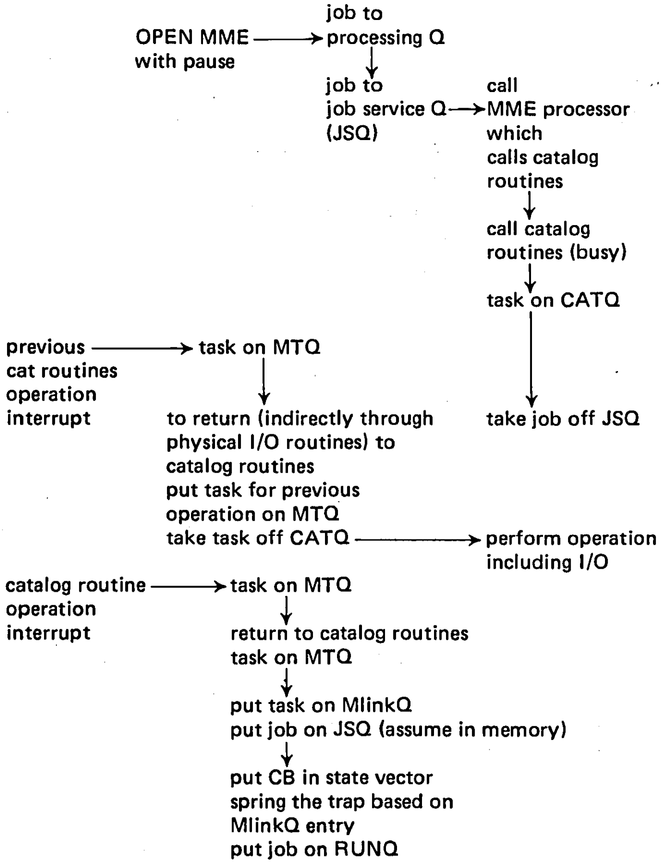

OPEN MME with pause
job to processing Q
job to job service Q (JSQ)
call MME processor which calls catalog routines
call catalog routines (busy)
task on CATQ
previous cat routines operation interrupt
task on MTQ
to return (indirectly through physical I/O routines) to catalog routines
put task for previous operation on MTQ
take task off CATQ
take job off JSQ
perform operation including I/O
catalog routine operation interrupt
task on MTQ
return to catalog routines task on MTQ
put task on MlinkQ
put job on JSQ (assume in memory)
put CB in state vector
spring the trap based on MlinkQ entry
put job on RUNQ

Fig. 5.6 – Processing an OPEN MME.

Associated with each device is a task queue and associated with each channel is a task queue. Thus a task requiring access to a device must first queue for a device (assuming it is busy) and then queue for the channel on which the device operates (assuming it is busy), with the exception that when there is but a single device on a given channel, in which case only a single queue is used.

As shown in Fig. 5.6, following the completion of the catalog routines, the Executive puts the task on the Mlink queue. The Mlink queue is a queue of tasks for the Executive to do before the job next runs and in particular is the traps and special interrupts which the Executive wishes to give the job. Thus when the Executive wishes to trap the job in the example in Fig. 5.6 following the completion of the OPEN MME, it makes an entry on the job's Mlink queue which consists of the status returns generated by the OPEN MME and the address of the routine (task) in the Executive which will process it. Since the job must be in memory in order to give it a trap, the Executive determines if it is in memory and if so, places the job in the job service queue. As discussed earlier, if the job is not in memory or is either actually running or in the processing queue, the job is so treated that it is known it will end up in the job service queue and is marked as requiring Mlink processing. When the job is serviced from the job service queue, the Mlink processing routine puts the file control block in the job's state vector, copies the status words and the job's IC/IR into the trap block, and sets the job's IC to point to the trap code. The job is then transferred to the run queue.

5.5.8 Executive cycle

Based on the queues, the control Executive organises its work by first processing all entries in the Executive task queue. Only when this queue is empty does it take jobs off the job service queue. When both these queues are empty it runs the swapper and when has completed it services real-time timer tasks.

5.6 SCHEDULING [5]

Only three of the many queues around the system operate on other than a first-in-first-out basis. Regulating how long jobs spend on the two non-FIFO queues – the run queue and the non-core queue, is usually sufficient to control the total resources consumed by a user.

5.6.1 Scheduling parameters

One of the files available to the Executive is an environment description file called ENVINFO.

The scheduling algorithms allows a total of fourteen parameters to be specified for it in ENVINFO. Most of them are in a sense 'noise words' in that small changes in them will not noticeably affect system performance. Changes can be made to give optimal behaviour when changing user loads or hardware available.

The following is a list of parameters that may be set, the meanings of each, and where possible a guide for determining the ideal values:

SWEEP: This parameter may be set to zero or one and determines which of two possible swapping algorithms is used. SWEEP=0 tells the swapper to pick a job to swap out on the basis of scheduling number (how many resources it has used recently) and whether the job is paused. SWEEP=1 tells the swapper to always pick the job just above the one most recently swapped as the next one to swap (where memory is thought of as a circular list excluding the Executive and memory resident jobs).

SWEEP=0 works best on systems where memory is plentiful, jobs are fairly uniform in size, and there is a great deal of variation in CPU required by various jobs. SWEEP=1 works best on systems with less memory where there is a great deal of variety in job sizes but CPU requirements are fairly uniform.

MAXSWAPS: This parameter tells the swapper the maximum number of jobs it should ever try to swap simultaneously. Typically this number is set to the number of physical channels and the number of swapping devices. Too large a value for this number may cause serious contention problems for channels, thus slowing down I/O bound jobs. Too small a value will slow response time for interactive jobs.

DECAY: This parameter tells the scheduler how quickly to forget the quantity of resources used by the job. Scheduling number decays over time and have a 'half-life'. The decay constant can be computed from the desired half-life as follows:

DECAY = .5   (.5/half-life in seconds) * 2ˆ35
(Expressed in octal)

A decay constant of 0.99, represents a half-life of 35 seconds. The effect of a longer half-life is to run small jobs faster and large jobs more slowly.

NCQRATIO: This number specifies how much lower the scheduling number of a job out of memory must be compared to a job in memory before the two may exchange places. Too small a value will cause excessive thrashing. Too large a value will cause erratic response times. A value of 200 (octal) specifies that the job in core must have twice as large a scheduling number to be forced out. A value of 140 specifies one and one half times as large, etc. This number is best thought of in terms of the decay half-life. A value of 200 will cause jobs to exchange places every half-life, causing response time to long running jobs to be erratic to that degree.

CRUNCHER: This number specifies the minimum memory size of a job before it will be considered atypically large. It is used to reduce memory fragmentation, and thus is particularly valuable when SWEEP=0. Jobs larger than the specified size are given somewhat more erratic service.

This number should be set high enough so that there are not normally more than two or three crunchers competing for service at any given moment. A value too high will cause excessive fragmentation and degrade overall system performance. A value too low will cause the Executive to waste CPU time doing unnecessary calculations.

FREECORE: This number specifies the amount of memory the swapper will try to keep empty so that out of memory jobs can be swapped in immediately upon request. Only jobs which have issued long pauses are swapped out to make room.

EPSILON: This number specifies a value of scheduling number below which values are considered negligible. Too small a value will make response given to small interactive job slightly erratic. Too large a value will make service given to small interactive jobs very consistent but on average slower.

CPU, I/O, CORE, FAUTIME, SWAPCHG: These parameters determine the formula for calculating computer resource units (CRUs). CPU, I/O, and CORE give the relative weights of these considerations. The three octal numbers should sum to 200000 (octal). They are scaled so that a 16K job using 60 I/O units per second of CPU will be charged the same number of CRUs regardless of the relative weights. FAUTIME is the number of milliseconds of CPU charged for each master mode request the job makes. SWAPCHG is a factor which is multiplied by the number of K words swapped and added to the scheduling number.

For scheduling purposes, CPU, I/O and CORE are optimally set according to the average number of users waiting for the corresponding resource. Looking at a snapshot of the job table, the number of users in each category can be estimated as follows:

(i) In memory waiting for CPU: number of jobs 'RUNNING' minus number of processors.
(ii) In memory waiting for I/O: number of jobs 'MASTER HELD' minus number of I/O channels.
(iii) Out of memory waiting for CPU (I/O): number of jobs on NCQ times percentage of jobs either RUNNING or MASTER HELD which are RUNNING (MASTER HELD) times CPU (channel) busy time percentage.
(iv) Waiting for memory: number of jobs on NCQ not accounted for in (iii).

FAUTIME is optimally set at the actual time required by the Executive to process the requests.

SQUAN, LQUAN: These values tell the scheduler how much CPU to allow a job in a single quantum. Too large a value will cause erratic response time when jobs tie up the CPU for long periods. Too small a value will cause increased overhead due to slicing.

Immediately following a MME, a job is given SQUAN milliseconds with slightly increased priority. Thereafter, it gets LQUAN milliseconds at a time until it issues another MME. Typical values used are 24 ms and 320 ms respectively.

5.6.2 Scheduling number

Every running job on DTSS has associated with it a scheduling number indicating the quantity of resources consumed 'recently'. Whenever a job consumes a resource – CPU, I/O, or CORE – a charge is added to the scheduling number. The conversion factors used to convert these three factors to 'CRU's are input in the environment file and should reflect the relative scarcity of the three. Every half second, all scheduling numbers are multiplied by a decay constant between zero and one, so the effect of resources consumed long ago slowly fades away.

The run queue and non-core queue are kept sorted by scheduling number. Whenever a CPU becomes idle, the job on the run queue with lowest scheduling number is given control. The situation with memory is more complex, since the Executive must act to make it available. This action is discussed in section 5.7.

Since some jobs represent the wishes of many users rather than just one (monitors, etc.), they must be given priority access to resources. Associated with this are two job type bits. These two bits form a priority number which overrides the scheduling number. Thus, monitors compete only among themselves for resources; any resources left over are divided among the non-monitor jobs.

5.6.3 Partitioning

Over and above the scheduler, a high level system of resource control, called partitioning, is provided.

Partitioning controls what portion of the system each of several groups of users gets by controlling the response time given to each group. The scheduler gives each job a startup quantum of resources. Jobs run very quickly while consuming this initial quantum, and are largely unaffected by partitioning. Eighty per cent of all jobs complete in their initial quantum and twenty per cent of all system resources are consumed by them. The remaining eighty per cent of system resources are divided among long running jobs. Without partitioning, all jobs are given resources at the same rate. If partitioning is active, response time ratios are set so as to keep group usage within bounds. By controlling response time given to foreground users, partitioning controls what portion of available resources are comsumed by each group. Partitioning has no effect on the allocation of storage (a static resource) nor on who uses background. It does regulate total consumption of the background system.

For efficiency and flexibility, the partitioning software is broken into two pieces. The Executive has a six bit partitioning code for each running job. For system overhead jobs this code is zero; for user jobs, the code comes from the user's validation record (see Table 4.3). The Executive maintains a table of total resources comsumed by each of the sixty-four partitioning categories. It schedules jobs according to ratios supplies to it in a sixty-four word table. Every two minutes, the PRIORITY module wakes up. It compares the amount of resources each group has used in the last two minutes to its quota. It then adjusts the response time ratios by a complex algorithm that ensures that each group gets its fair share. The PRIORITY module is a short program written in a high level language, which runs with BAKMON. Extensive changes to the algorithm can be made and tested in a way that is safe to the system.

Every job belongs to a partitioning category. Partitioning categories are numbered from zero to sixty-three. When a foreground user runs a job, the category number comes from the user's validation record. When a job runs in background, it runs in either category one, two, or three depending on whether it is a Bulk Media Conversion activity (one), a run accessing tapes (two), or a run which does not access tapes (three). When a job runs beneath a Monitor which does not reliably supply partitioning information to the Executive, it runs in category sixty-three. Jobs which run at the top of the job tree run in category zero. Only jobs which category zero may specify a category other than their own when running a job.

Associated with each partition category, is a total service quota and a response time quota. The total service quota is the portion of the system to which the group is entitled. The sum of all the service quotas is one hundred per cent. The response time quota indicates the level of service which should be given to each member of the group if the group is using less than its total service quota.

If a group is using less than its total service quota, the excess quota is distributed to other groups.

Users may get a snapshot of partitioning effects and system responses by typing EXPLAIN RESPONSE.

5.7 SWAPPING

To do almost anything, a job must reside in memory. Normally, all running jobs will not simultaneously fit in memory, so some are written out to mass storage. When a job not in memory wants to do something, it is put on the NCQ (Non-Core Queue). It is the responsibility of the swapper to pick jobs in memory to be written out and jobs on the NCQ to be read into memory.

The swapper normally writes out jobs which are not expected to be serviced in the near future, either because of a high scheduling number or because the job does not want service. It normally reads in the jobs wanting service with lowest scheduling numbers first. The swapping algorithm is given below. The actual code in the Executive is called whenever a swap might be called for, so the algorithm is best thought of as a continuously operating process. A detailed explanation of each step of the algorithm follows:

```
       begin
1.   if donotswap = true
       then exit
2.   else if NCQempty = false
3.       then headNCQ(job)   'front of NCQ in job
4.           do if corefull = false
5.             then initcopy
                 exit
6.             else if coreresenabled(job) = true
7.                 then swapjob(blockingjob)
                     exit
8.                 else if skip(job) = true
9.                     then nextNCQ(job)
                       else loopexit
             loop
             swapout
10.      else if enoughfreecore = true
             then exit
             else swapout
       end

       define swapout
11.  selectjob(job)   'pick job most eligible to swap out
12.  if motivation = true
13.  then swapjob(job)
           exit
       else exit
       defend
```

1. No additional swaps will be considered if any of the following conditions hold: there are already at least MAXSWAPS swaps in progress; the system is hung (the operator typed 'STOP'); there is a shortage of list elements and at least one swap is in progress; or there are fewer than three memory device addresses left for use as pointers in the copy.
2. If there are jobs waiting to enter memory, they will be considered before swap outs are considered.
3. The NCQ is sorted by scheduling number within priority. The first job is the one with highest priority/lowest scheduling number.

4. When looking for a space to put a job, the Executive must find a 'hole' big enough to fit the job and its state vector. Normally, any place in memory not occupied by the Executive or another job is considered. If the job has enabled core-residence permission, it is only allowed to swap into low memory abutting the EXEC or other core-resident jobs. This avoids the fragmenting of memory by non-swapping jobs scattered through it. If the SWEEP flag is set, jobs are only allowed to swap in immediately above the job which most recently swapped in (or in low memory just above core-resident jobs if there is not room between the last job and the top of memory). This has the effect of cycling through memory and avoiding fragmentation.
5. If the job fits in memory, the memory is allocated to the job, the job is removed from the NCQ, and a read is initiated to get the job into memory. The swapper then exits and the process of looking for another job to swap begins again. When the read completes, the job is put on the core queue (CQ) and is serviced.
6. If the job cannot fit in memory, the action taken depends on whether the job has enabled core-residence permission.
7. If the job has enabled core-residence permission, it must swap into a particular space in memory. The core queue is scanned for jobs occupying that space. If any are found, they are swapped out. If none are found, the swapper waits for any swaps in progress to complete. If there are no swaps in progress, it means memory is not big enough to hold all of the core-resident jobs. A 'swap panic' message is printed on the console and the highest core-resident job is swapped out. This enables the system to 'limp along' while the operator tries to find the cause of and correct the excess of core-resident jobs.
8. If a job has not enabled core-residence permission and if it is very large (bigger than CRUNCHER), the swapper may try to swap in a job with higher scheduling number first. This will only happen when there would not be enough memory available to swap the job even if all jobs in memory with higher scheduling numbers were swapped out. Otherwise, the swapper will try to find a job to swap out to make room.
9. If the above conditions are met, try to swap in the next job on the NCQ.
10. Even if no jobs are currently waiting to swap in, the swapper may want to swap out a job to make free space. Having free space available makes it possible to swap in jobs immediately when they request service. FREECORE in the ENV file specifies how much memory to try to keep available.
11. Picking the job most eligible to swap depends on the value of SWEEP as set in the ENV file. If SWEEP=1, the job chosen will be either the first job in memory above the job most recently swapped in or if there is

not one, the lowest non-core-resident job. If SWEEP=0, the job chosen is the 'Long Paused' job which has been in memory the longest or if there is not one, the job with the lowest priority/highest scheduling number whose core ticker has run out. A job is 'Long Paused' if it has done a Long Pause MME or if it has been suspended or roadblocked without master holds for over one second of real time. The core ticker is a guaranteed period of core-residency. When a job is swapped in, its core ticker is set for one half second for each 16K or fraction of memory size. The job may not swap until the ticker runs out. The ticker is disabled if the job issues a Long Pause MME.

12. Once the job most eligible to swap has been chosen the swapper decides whether to swap it out. If it is considering a swap out only to make free space (see 10), it will swap the job out only if no swaps are in progress and the job is 'Long Paused'. If there is a job waiting and the job most eligible is 'Long Paused', it will be swapped. If the job's core ticker has not run out or if there are MAXFORCE swaps already in progress, no swap will be considered unless the job is 'Long Paused'. A job that is not 'Long Paused' will swap out only if a job on the NCQ has significantly lower scheduling number. 'Significantly' is defined by NCQRATIO. The default value is 1.5, meaning that the job in core must have a scheduling number at least 1.5 times as large as that of a job on the NCQ.
13. When a job is chosen for swapping, the executive removes it from the queue of jobs to be run, waits for any I/O on the job's memory to expire, and/or waits for it to finish its quantum if it is running. Then it initiates a copy to mass storage. When the copy completes, the memory occupied by the job is deallocated.

5.8 INPUT/OUTPUT [6]

As noted in Chapter 2, and shown in Fig. 2.1, all input/output devices are connected to an input/output multiplexor (IOM). All input/output operations are accomplished using a single instruction – connect I/O channel (CIOC). The address field of the instruction is used to select a word in memory which contains the address of the channels on the system control unit on which the appropriate IOM is connected. This generates a connect pulse on the IOM. When the IOM wishes to signal its completion of an action, it interrupts the control processor. There are thirty-two interrupt cells, each two words long which occupy the first sixty-four words of memory. Most of these are unused by either the hardware or the software. Each IOM is associated with four cells and the software uses two other cells for inter-processor communication (slave processors signal the master processor using an interrupt generated with a Set Memory Controller Interrupt Cell (SMIC) instruction) and for crashing the system when the software detects an inconsistency in its operation. Of the four cells associated with each

IOM one cell is used to signal errors associated with the CIOC instruction, one to signal normal I/O completion, one for unsolicited interrupts such as a peripheral being powered-up, and one is unused.

Information if transmitted to and from the IOM via the areas called IOM mailboxes and the interrupt multiplex word (IMW). The IOM mailbox is a one hundred twenty-eight word block for each IOM. The IMW table is a single thirty-two word block shared by all IOMs and corresponds to the thirty-two interrupt cells.

5.8.1 IOM mailboxes

Each IOM has a mailbox area of one hundred twenty-eight words. Since each IOM may have thirty-two channels, this gives four words per IOM channel. Channels 10–37 are used to connect devices to, whilst channels 0–7 are special. In particular, channel 2 is the connect channel through which all CIOC instructions are routed and channel 6 is the special status channel (see section 5.8.2). The format of the connent channel mailbox is shown in Fig. 5.7.

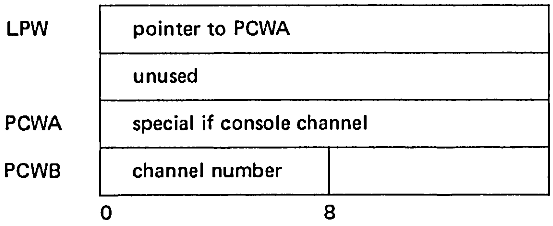


Fig. 5.7 – Connect channel mailbox format.

When the IOM acts on a CIOC, it picks up the LPW word from the connect channel and, using the channel number in PCWB, transfers control to the appropriate channel routine. If the channel is the console, the one and only command word (IDCW) is the one in the connect channel word PCWA. If the channel is a common peripheral interface (the 716 is on such a channel), the IOM treats the PCWA in the connect channel as the first of possibly many command words. DTSS places a no-operation IDCW in the connect channel so that the handling of common peripheral interface channels and peripheral subsystem interface (PSI) channels, which ignore the PCWA in the connect channel, are treated the same. Thus, in general, the IOM uses the mailbox of the channel named in PCWB in the connect channel mailbox to perform the operation. The format of the general mailbox is shown in Fig. 5.8.

LPW	address of DCW/IDCW list
	unused
SCW	address for status return
	unused

Fig. 5.8 – Data channel mailbox format.

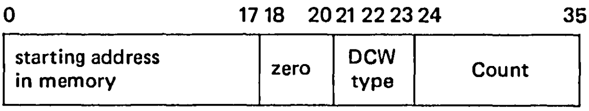
Each data transfer is controlled by a Data Control Word list that specifies the areas of memory to or from which data is to be transferred. The DCW list allows data to be gathered from, or distributed to, non-continuous locations in memory. A DCW list consists of words of two formats. The list alternates an IDCW word which is the command word (its format is shown in Fig. 5.9) and some number of DCW words which are the pointer words (shown in Fig. 5.10). A console channel IDCW (in the PCWA word of the connect channel) has a similar format to that shown in Fig. 5.9.

This, for example, Fig. 5.11 shows the pointers set up to read from disc to memory. Fig. 5.12 shows in detail the same situation assuming the read is for 6.1K words and seek is required before the read.

0 5	6 11	12 17	18 20	21 22	23	24 29	30 35
CMD	DEV	Add X	7	X C	M	Chan Cmd	Count

CMD Command – different codes for every type of device.
Dev Physical device address on the channel.
Add X Address extension bits, concatenated with address in DCW word for extended memory addressing.
Bits 18-20 Must all be 1 for a consistency check.
Bit 21 If set, use address extension.
Bit 22 If set, another IDCW will follow after any DCWs.
Bit 23 If bits 22 and 23 are set, a marker (intermediate) interrupt will be generated following this command; the channel will remain busy however.
Chan Cmd 0 = Single operation.
2 = Non-data transfer.
6 = Multiple operation.
Count Count of operations if Chan Cmd = 6.

Fig. 5.9 – IDCW format.



DCW type 00 = This is the last DCW in the chain.
01 = More DCWs to follow.
11 = More DCWs follow but ignore this address – generate zeros if output, throw away data if input.
10 = Transfer DCW – get next DCW from named location.

Count Count of number of words to transfer, maximum is 4096 with single DCW.

Fig. 5.10 – DCW format.

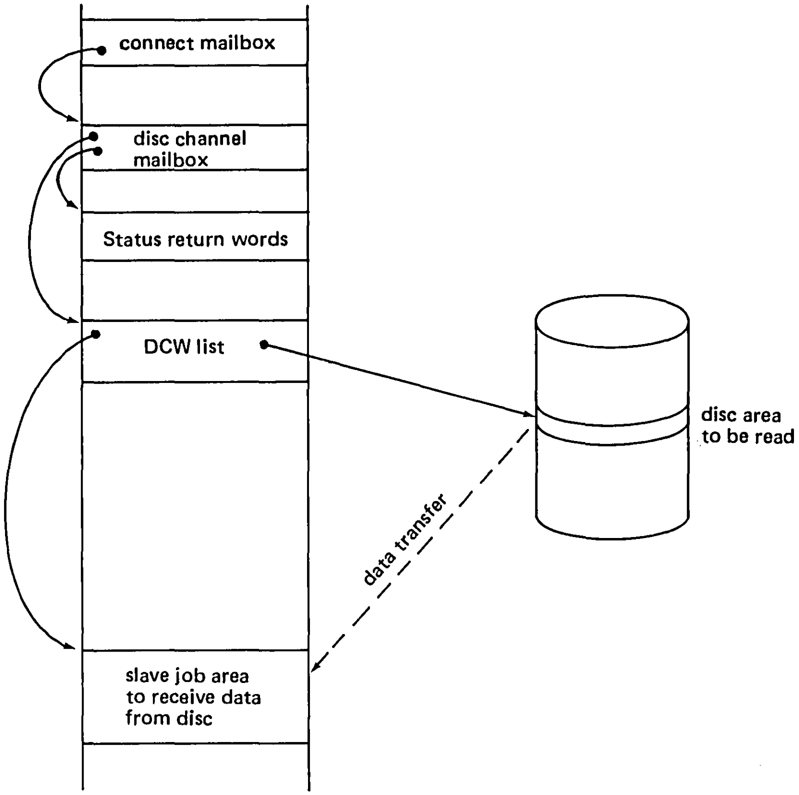


Fig. 5.11 – Pointers for a disc read.

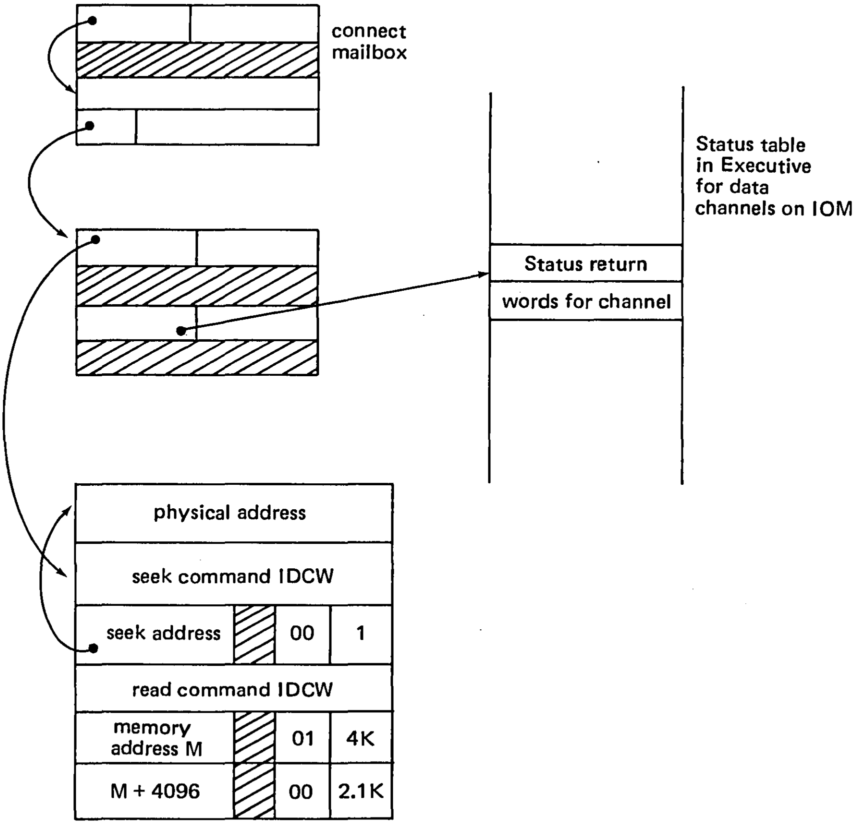


Fig. 5.12 – Seek, read on disc for 6.1K words.

5.8.2 Interrupts

When an operation completes, the IOM writes two words of status information. The Executive maintains a forty-eight-word status table for each IOM, with two words allocated to each of the twenty-four data channels (channels 0–7 are special channels which do not generate status returns handled in this way). The SCW word of the data channel mailbox points to the position in this table. The IOM also writes information into the IMW table (whose position in memory is determined by switch settings on the IOM itself which the IOM can read). The format of the IMW table is shown in Fig. 5.13.

The IOMs on the system are numbered from zero, and IOM n leaves messages in words 4 + n, 14 + n, 24 + n and 34 + n of the IMW table. The IOM turns on a bit in the appropriate word according to which channel needs attention. Thus bit twelve in word fourteen means initialisation or termination interrupt on

0 3	unused
4 7	System fault
10 13	unused
14 17	Init/term interrupt
20 23	unused
24 27	Marker interrupt
30 33	unused
34 37	Special interrupt

Fig. 5.13 – IMW table.

channel twelve of IOM zero. The word position in the IMW table times two is the address of the correspondence interrupt cell. The IOM uses this to generate an interrupt on the control processor which it does by carrying out the same action as a Set Memory Controller Interrupt Cells (SMIC) instruction with the address of the interrupt location calculated from the IMW table offset.

The control processor must then interrogate the appropriate IMW word to discover the channel number and from this is able to pick up the status information from the status table and hence service the interrupt.

The IOM sets bits in a word in the system fault section of the IMW table when such things as parity error or mailbox words pointing out of memory occur. Marker interrupts are set when bits twenty-two and three in the IDCW are set as described in Fig. 5.9.

Special interrupts are generated when a device/channel needs attention but is not the subject of a command. Thus mounting a tape or powering-up the card reader are events generated as a result of operator action rather than program action. The action on a common peripheral interface differs from the action on a PSI channel.

When a non-PSI channel requires attention, it turns on its IMW special interrupt bit and generates the appropriate interrupt, but no further information is given. Thus for example if there were eight tapes on a non-PSI channel and one goes ready, an interrupt is generated on the tape channel and the Executive must request status on each drive to find out what happened.

On PSI channels, all special interrupts are routed through channel six of the IOM (the special status channel). A word of data is provided by the MPC (peripheral processor) which includes the channel number, the device number, and the code for the cause of the special. This data is stored in a stack, with the stack pointer in the special channel mailbox, the format of which is shown in Fig. 5.14.

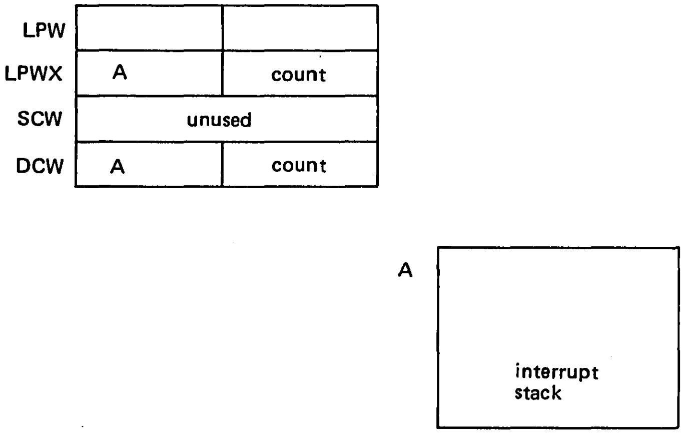


Fig. 5.14 – Special status mailbox.

The DCW word contains the stack pointer. When the count gets to zero, the stack pointer is reset from the second word of the mailbox, thus giving a circular stack.

When the Executive gets the special interrupt, it uses the information on the stack to process the interrupt.

5.9 REFERENCES

[1] Pedersen, L. C. (October 1975), 'DTSS Job-Executive Interface', DTSS System Memorandum 211.

[2] Garland, S., and Dwyer, A. (October 1974), 'DTSS Programming Reference Manual', DTSS Technical Memorandum 059.

[3] Fernandez, L. F. (October 1975), 'DTSS Executive-Internal Structure', DTSS System Memorandum 217.

[4] Pedersen, L. C. (1976), 'Communication Files', DTSS Internal Document.

[5] Kaufman, C. (June 1976), 'The DTSS Scheduling Algorithm', DTSS Internal Document.

[6] Kaufman, C. (1977), 'Input/Output Notes', DTSS Internal Document.

CHAPTER 6

Slave-Mode Operating System

6.1 TERMINAL COMMUNICATION CONTROLLER INTERFACE (TCFACE)

This module is the interface between the terminal communication controller (TCC – the 716 front-end processor), and hence between the user terminals, and the slave jobs. In common with all other devices the user terminal looks like a file to a job wishing to send or receive data to or from it. This is achieved with a communication file, a slave end of which is held by the job and the master end is held by TCFACE.

6.1.1 Overall structure of TCFACE

TCFACE is organised into a cyclic section and a non-cyclic, interrupt driven section. The cyclic section is concerned with sending and receiving data to and from the TCC. This is shown in Fig. 6.1.

Since TCFACE holds the master end of communication files it receives all the special interrupts for terminal output generated by slave jobs (see section 5.3) and the traps from the WRITE MMEs it issues to transfer data input from the terminal. These asynchonous events are analysed and placed on queues for processing in TCFACEs spare time.

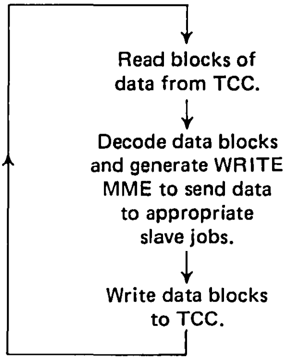

Read blocks of data from TCC.
Decode data blocks and generate WRITE MME to send data to appropriate slave jobs.
Write data blocks to TCC.

Fig. 6.1 – TCFACE cyclic routines.

6.1.2 Communication file control

The status of a communication file is controlled and monitored via a series of bits in a table called TFR which is indexed by terminal number. The function and purpose of the more important bits is described in Table 6.1.

The overall policy of TCFACE is to run uninhibited whenever possible. Traps and special interrupts are queued for later processing. In addition, tasks which must be deferred because the communication file is busy are queued as are tasks which must be deferred due to buffer allocation problems. Most elements on such queues are held in a six word block whose general format is shown in Fig. 6.2.

−4	Dependent on block usage	
−3	Dependent on block usage	
−2	Dependent on block usage	
−1		Q$LINK
0	Q$ROUT	Q$TRMNO
+1	Dependent on block usage	

Q$LINK	Pointer to next Q$ROUT. End of queue contains zero or in the case of traps a transfer to routine to process traps.
Q$ROUT	Address of routine to process queue element.
Q$TRMNO	Terminal number.

Fig. 6.2 – Generalised queue element.

6.1.3 Message formats and protocol

TCFACE reads data to be sent to the terminals over communication files and sets up message buffers which consist of a one-word header and an n-word data block. The format of the header word is shown in Fig. 6.3. Multiple messages are blocked together for transmission in a block with a maximum size of two hundred and fifty-five words.

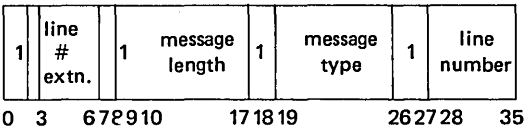


Fig. 6.3 – Message header format.

Bits 0, 9, 18 and 27 are set to one to allow a consistency check to be carried out. The line number field is split into two to retain consistency with the previous TCC (an 18-bit Datanet 30) and is used to identify the terminal to which the data following the header words is to be sent. The message length is the number

Table 6.1.

Name	Description
B$TCN	Terminal connect bit – on if the terminal is connected.
B$TDR	Terminal drive sent bit – on if a DRIVE MME has been sent to the slave end requesting a read, and the slave end has not yet done so.
B$TOP	Terminal output pending bit – on if the terminal has been sent output and the TCC has not yet sent a BLB message to TCFACE (see section 6.1.3).
B$TWS	Terminal warning pending bit – on if a warning message from the operator has been received for this terminal but has not yet been transmitted to the terminal.
B$WFR	Wait for read bit – on if the slave end has reserved the communication file. If this bit is on TCFACE will not DRIVE 'read communication file', special interrupts to the slave end when unsolicited input is received. Instead, it will wait for the slave end to issue a READ.
B$RSS	Reset status bit – on if a RESET STATUS MME was issued on the communication file by TCFACE while the file was busy and the trap on the outstanding operation has not yet been sprung.
B$TIF	Time-out field – this three bit field is incremented every eighteen minutes provided B$SRO is set. If the count reaches six the terminal file will be disconnected. The field is set to zero whenever a slave issues a READ or data is transferred to the slave end.
B$SRO	Slave read outstanding bit – on if a slave end has issued a READ and TCFACE has not yet trapped it.
B$SWO	Slave write outstanding bit – on if the slave end has issued a WRITE and TCFACE has not yet transferred all the information.
B$SCL	Slave closed bit – on if a slave end has issued a CLOSE and TCFACE has not yet received an end of output message from the TCC.
B$MSF	Master state field – this three bit field is used to represent the state of the master end of the communication file: 0 – Master end idle. 1 – Master end busy. 2 – Master end reset. 3 – Master end closed.

of words following the header. The message type is used to control the protocol between the TCC and TCFACE.

When TCFACE is transmitting data to a terminal, the message type is always 000 (continued transmission). The TCC does not acknowledge this but when the TCC calculates it has only one seconds worth of data left to send to the terminal, it sends a message consisting only of a header word with message type 021 (beginning last buffer – BLB). TCFACE responds by reading more data from the communication file proving the slave issued WRITE is still outstanding, otherwise it will do nothing. When the TCC has completed the output to a terminal it sends a message consisting only of a header word with message type 020 (end of output). TCFACE uses this to allow terminal input to commence since it enforces a protocol whereby input and output from and to the terminal may not proceed simultaneously.

A number of message types are sent by TCFACE to the TCC as a result of users issuing SIMON commands which control the action of the terminal. Thus, for example, if a user issues the command FULLDUPLEX, SIMON DRIVES full-duplex mode on the user's terminal communication file. TCFACE in responding to the DRIVE special interrupt will generate a message header word of type 060 (set full-duplex mode). The TCC uses this to control the transmission process to the terminal.

The TCC reads data from the terminals and sets up a message buffer for each terminal which is inputting characters.

When the TCC is transmitting data to TCFACE, there are two common message types which signal to TCFACE what action to take. The types may be classified as 'here is some data because my buffer has overflowed' and 'here is some data because the user has signalled this is the end of a logical transmission'. The first type is message type 000 (continued transmission from this terminal) and the second type is either 001 (break character received), 002 (end of BUILD mode – two carriage-returns received) or 003 (end of line-by-line mode, end of command mode). Thus, for example, when the user types BUILD to SIMON, SIMON DRIVESs build mode on the user's terminal communication file, prints SPEAK!, and issues a COPY from the user's terminal communication file to the end of the user's current file. This DRIVE is passed on to the TCC by TCFACE sending a message type 041 (set build mode). When the TCC detects two carriage-returns in sequence from a terminal in build mode, it sends a message type 002. TCFACE responds by trapping SIMON's COPY to the user's current file. SIMON responds to the trap on the COPY by returning the user's terminal communication file to its normal state, driving command mode and printing READY. TCFACE in response to the DRIVE sends a message type 040 (set command mode) to the TCC.

A list of the more important message types are given in Tables 6.2 and 6.3.

All message blocks in both directions, are terminated with a message type 100.

Table 6.2 – TCC to TCFACE message types.

000	Continued transmission from this terminal.
001	Break character received.
002	End of BUILD-mode.
003	Line-by-line or command mode ended.
004	Answer back drum read follows.
011	Carrier lost – user no longer there.
012	Carrier detect – new user identified.
020	End of output.
021	Beginning last buffer.
100	End of this transmission.
101	Hardware load.
102	Program load request.
104	Dump TCFACE memory.

Table 6.3 – TCFACE to TCC message types.

000	Continued transmission.
040	Set line-by-line-normal program input mode
041	Set build mode.
042	Set tape mode for build mode.
043	Set direct mode.
044	Set command mode – normal SIMON mode.
045	Set tape mode for command mode.
046	Stop-options accepted (CTRL-X, S and BREAK).
050	Delete input buffer.
051	Next command is patch or dump.
060	Set full duplex mode.
061	Set half duplex mode.
062	Set direct output mode.
063	Reset direct output mode.
064	Set friden mode.
065	Reset friden mode.
066	Stop output – dump all buffers.
100	End of this transmission.
101	Hardware load.
102	Program load follows.
103	Reset all lines connected to TCC.
140	Dump.
141	Patch.
142	Validate a debugging user.

6.1.4 Message flow control

Fig. 6.4 shows the routing of messages from the user's terminal to a slave job.

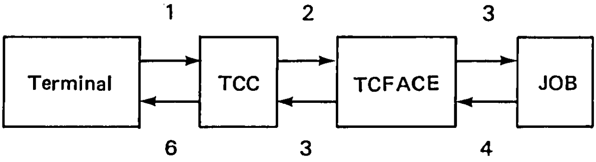


Fig. 6.4 – Message Routing.

The flow control exercised over the six paths shown in Fig. 6.4 is as follows:

1. With asynchronous terminals there is no control (other than that exercised by the user himself) and the TCC must accept each character as it arrives and buffer it.
2. TCFACE has no control over the message flow from the TCC. It must accept all messages and buffer them. To ensure nothing is lost, TCFACE maintains an overflow area on disc for three messages per terminal to a maximum of one hundred and twenty-eight words.
3. If a COPY is outstanding, the job has no control over the arrival time or rate of terminal input. If no COPY is outstanding, and the terminal input is unsolicited, the job will receive a special interrupt which it processes in its own time and also issues any COPY to the communication file to read the data in its own time. Thus TCFACE must buffer such data until the job decides to read it.
4. TCFACE controls the rate of reading, based on the availability of its buffer space, by issuing READS on the communication file when it wants to and also for the amount of data it is prepared to accept.
5. The TCC controls the flow of messages by sending beginning last block or end of message types. TCFACE does not transmit until it receives these messages,
6. The terminal must accept data asynchronously as it is sent by the TCC.

6.2 LOGIN

This module runs the various monitors and also validates each user at sign-on time. In running the monitors it provides a command interface to the monitors through which it provides a number of services the most important of which is the maintenance of billing and statistics files.

6.2.1 User validation [1]

The validation records for all the user numbers in a given user group are stored in a single validation file which is catalogued in the group catalog. The group catalog is in turn catalogued in a super-group which contains the group catalogs for all user numbers in the same super group. Fig. 6.5 shows a brief schematic of the DTSS file structure down to the user catalog level.

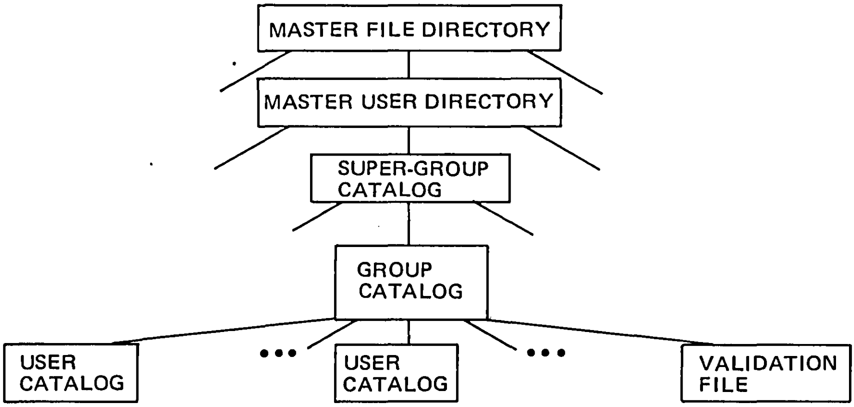


Fig. 6.5 – User Catalogs and Validation Files in the DTSS File Structure.

There are a variety of different validation record formats within a validation file, most of the variations being in the interest of conserving space. Only the last three digits of the user number in are specified in a validation record. Each record format is designated by a three-bit entry code in the first word of the entry. Codes zero and one indicate entries for a single user number with no special permissions.

A validation record may also be used to define a block of consecutive user numbers with identical permissions. Entry codes two and three indicate records containing blocks of continuous user numbers with no special permissions.

Entry codes four and five are used whenever special permissions or answerback drum entries are required, whether for a single user or a block of users. Entry code four does not specify a password; entry code five is similar to entry code four but with the addition of a password; it is shown in Fig. 6.6.

Entry code six is always used for the first validation record in a validation file; it lists the common answerback drum entries for the user group (if any).

Finally, entry code seven is used as a terminator for the validation file.

LOGIN carries out the checks based on information supplied by the user and information held in the validation record such as an answerback drum check of a check against legal log-on times. Assuming the user passes the validation check, LOGIN passes the slave end of the terminal communication file and the user catalog to a monitor together with a message block which contains the status on the user catalog together with the user number, the terminal number, the permissions word (see Table 4.3) and the limits word (see section 4.5.1). This message is shown in Fig. 6.7.

<table>
<tr><th>0 3</th><th>4 11</th><th>12 19</th><th>20 26</th><th>27 29</th><th>30 35</th></tr>
<tr><td colspan="4">FIRST USER NUMBER IN BLOCK</td><td>5</td><td>UC</td></tr>
<tr><td colspan="4">LAST USER NUMBER IN BLOCK</td><td colspan="2">ZERO</td></tr>
<tr><td colspan="6">PERMISSIONS</td></tr>
<tr><td colspan="2">ZERO</td><td colspan="4">LEGAL LOG-ON TIMES</td></tr>
<tr><td>MAX TIME (LOG2)</td><td colspan="2">MAX CATALOG STORAGE</td><td colspan="3">MAX STORAGE STORAGE</td></tr>
<tr><td colspan="6">SPARE</td></tr>
<tr><td colspan="6">NUMBER OF WORDS IN THIS VALIDATION RECORD</td></tr>
<tr><td colspan="6">PASSWORD – CHARACTERS 1-4</td></tr>
<tr><td colspan="6">PASSWORD – CHARACTERS 5-8</td></tr>
<tr><td colspan="6">FIRST ANSWERBACK DRUM ENTRY</td></tr>
<tr><td colspan="6">SECOND ANSWERBACK DRUM ENTRY</td></tr>
<tr><td colspan="6">.
.
.
.
.
.
These words are used only if there are any private answerback entries.</td></tr>
<tr><td colspan="6">LAST ANSWERBACK DRUM ENTRY</td></tr>
</table>

Fig. 6.6 – Long format entry with passwords.

<table>
<tr><td colspan="2">Access catalog open with</td><td>Preference info</td><td>Type</td></tr>
<tr><td colspan="4">Catalog MAX</td></tr>
<tr><td colspan="4">Read/write pointer</td></tr>
<tr><td colspan="4">Maximum accesses available</td></tr>
<tr><td colspan="4">User number</td></tr>
<tr><td colspan="4">User number</td></tr>
<tr><td colspan="4">Terminal number</td></tr>
<tr><td colspan="4">Permission word (see Table 4.3)</td></tr>
<tr><td>Max CRUs</td><td>Max cat storage</td><td colspan="2">Max scratch storage</td></tr>
</table>

Fig. 6.7 – Message block from LOGIN to monitor.

6.2.2 Command interface

LOGIN provides four basic services via a command interface. These services are gathering statistics and billing data from the monitors, making catalogs available to monitors, transmitting messages between monitors, and controlling access to the system. The commands associated with billing are:

BYE (and two other related commands) which is used to inform LOGIN that a user has been disconnected from the system and to pass the billing record to LOGIN;

HEL used to inform LOGIN that a user wishes to change user catalogs or monitors and to pass the billing record for the session so far to LOGIN;

BIL used to pass billing information to LOGIN following a background job.

The command associated with statistics is:

STA used to pass statistics records to LOGIN in response to a request from LOGIN for statistics.

LOGIN collects billing information for CPU usage and I/O units sent to it by the monitors using commands such as BYE, HEL, MON, and BILand appends to this connect time information which it holds for the user. These billing records are written by LOGIN to the billing file for later processing by the billing programs.

Every five minutes LOGIN sends a 'read communication file' drive down the communication file to each monitor. The monitor reads the file and finds a message saying 'send me your statistics'. The monitor, using the STA command to LOGIN, send the statistics down the communication file. LOGIN also collects statistics from TCFACE using a communication file and from the Executive by reading Executive's memory directly. The statistics are written by LOGIN into a file called the daily billing file for later processing by program called BILLPG.

The name used for the daily billing file is the data which it was begun in MM/DD/YY format. A given daily billing file will contain all usage information for one 'DTSS Day'. A 'DTSS Day' begins the first time DTSS is booted on a given date and continues until the system is booted on a different date.

The OPURGE module (run by the operator) is responsible for recording the amount of storage associated with individual user catalogs, and this information is stored in a file called STORAGE.

BILLPG is run on a regular basis to create a file called BILLSUM. At the end of a monthly billing period, BILLSUM is copied to a monthly summary file. Fig. 6.8 illustrates this sequence.

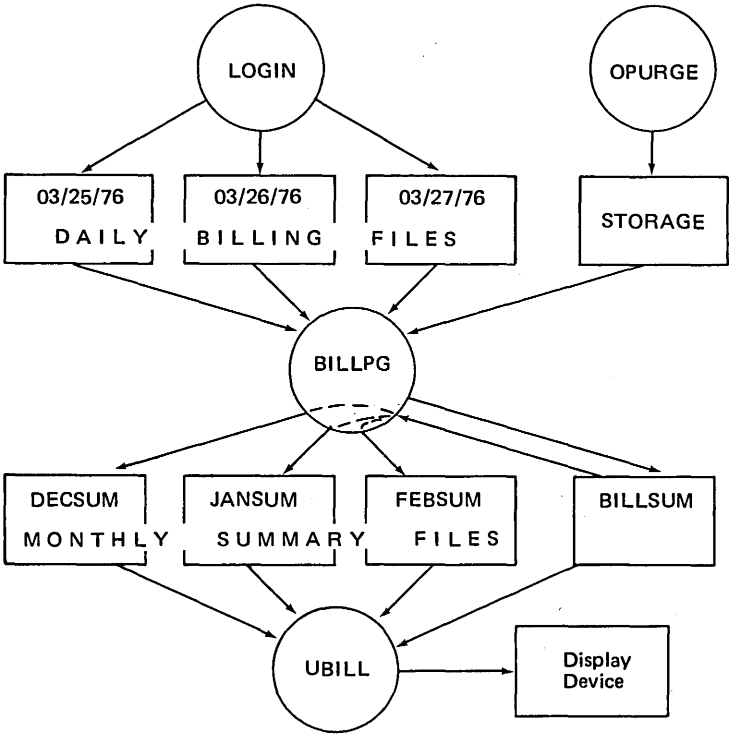


Fig. 6.8 – Statistics and billing.

The command associated with catalog access is:

CAT used, particularly by BAKMON, to access files in catalogs other than the current catalog.

The command associated with message passing between monitors is:

MES used, for example, by SIMON to send wakeup messages to BAKMON to inform it there is something for it to do.

There are a variety of commands, valid only from OPMON, which are to do with controlling system access during the period when the system is being closed down, and to do with assigning which lines are to be designated as control lines.

Since LOGIN is used infrequently, it swaps, unlike the Executive, TCFACE, and SIMON.

6.3 MONITORS

There are nine monitors running beneath LOGIN most of which, like LOGIN, swap (SIMON does not). Most of these modules are designed to provide an environment under which user jobs run. The monitor under which most users run is SIMON, and this will be described in detail. The name and brief function of each monitor is given below:

BAKMON This monitor processes job descriptions in the submission catalog as resources become available. Jobs are placed in the submission catalog by the background system, run by SIMON when a user types the command BACK (see section 6.3.3).

CHEMON This monitor is solely for running chess playing programs. It gives the user infinite run time but restricts access to chess programs only. It also limits the number of users who can use the monitor simultaneously.

MUXMON Use to control the datalink system (see section 7.6) for RJE stations and computer-computer connections. It is also used for down-line loading of remote minicomputers.

OPMON This monitor is the same code as SIMON but runs with more privileges. It is used by the system operator.

RAM This is the restricted access monitor which is used to restrict users to the use of specified resources such as a single applications program.

SIMON This is the general purpose, user interface to the system. It processes commands from the user and provides the requested services or runs jobs to provide the requested service. Most users of the system interface to SIMON, which multiprograms the users connected to it (see section 6.3.1).

SUPMON This is a special purpose monitor which is a copy of DDT (the assembly language debugger) and is used for debugging other monitors. As such it is a highly privileged monitor.

TCL This special monitor does not run users but is the telecommunications loader, invoked when the front-end communication processor requires reloading or dumping (see section 7.9).

VALMON This monitor is used to run privileged users who are setting up, modifying or deleting user validation records. The validation records specify the conditions under which a user may obtain access to DTSS and the resources available after is granted (see section 6.3.4).

6.3.1 SIMON [2]

SIMON (Simple Monitor) is the primary user interface for the Dartmouth Time-Sharing System. When a user logs on to the system, control of his terminal is normally passed to SIMON, which prints the message NEW OR OLD –. The user may then start a conversation with the system by giving commands to SIMON.

A user under SIMON has two basic files, a current file and an alter file. A user's current file may at various times contain a program, a document, a block of data, etc. The user may alter his current file via the NEW and OLD commands, the various text-editing commands, the BUILD command, etc. In addition, the user may alter his current file by typing in line-numbered text from his terminal; SIMON appends such line-numbered text to the user's alter file.

A sort consists of merging the user's alter file into his current file in such a way that the resulting current file is in ascending line-number order. If several lines in the user's alter file begin with the same line number, only the last such line is kept. A sort will be carried out only if the user's current file is in ascending line-number order; if this is not the case, a message is printed and the current file and alter file are left unmodified.

In general, if the execution of a user command will involve the user's current file is some way, SIMON will run the Majorsort module to do a sort before executing a command. If the execution of a command does not cause a sort to take place, the user's current and alter files are left unmodified; subsequent alterations are simply appended to the alter file.

The manner in which many SIMON commands are executed depends upon the settings of various parameters within the user's validation record. These parameters are passed to SIMON by LOGIN when a user signs-on to the system and consist of two 36-bit words. The first of these is a permissions word, which controls the user's ability to access certain system facilities, such as privileged catalogs, peripherals, etc. (see Table 4.7 and Fig. 6.7). If a bit is on, the user may access the corresponding facility; if the bit is off, access is denied. The second validation word passed to SIMON is a quotas word; it controls the amount of CPU time and the volume of file storage that may be used in the execution of a given command (see Fig. 6.7).

If a user has run permission, SIMON will allow him to issue a RUN, COMPILE, or LINK command on any system. If a user does not have run permission, SIMON will allow him to issue a RUN, COMPILE, or LINK command only on the official systems. The experimental counterpart of an official system is also considered an official system; the experimental counterpart of a system is formed by prefixing its name with the letter 'X' (for example, the experimental counterpart of ALGOL is XALGOL).

The current official systems are:

7MAP	DTSS 716 Marco Assembly Program
8MAP	DTSS PDP-8 Marco Assembly Program
9MAP	DTSS PDP-9 Marco Assembly Program
ALGOL	DTSS ALGOL
APL	DTSS APL
BASIC	DTSS BASIC
CHESS	Chess-playing Program

COBOL	DTSS COBOL
COURSE	IBM compatible COURSEWRITER III author program
CPS	'Complete Programming System' developed at Bates College
CROSREF	Program cross-references
DDT	Honeywell 600/6000 machine language testing program
DMAP	DTSS Datanet-30 Macro Assembly Program
DTRAC	DTSS Text Reckoning and Compiling Program
DXPL	DTSS XPL Translator Writing System
DYNAMO	Simulation language
FORTRAN	DTSS FORTRAN
GMAP	Honeywell 600/6000 Macro Assembly Program
LISP	DTSS LISP
MIX	DTSS MIX Assembler
PER	Gives a list of user permissions
PILOT	DTSS PILOT course writer
PL	DTSS PL/I
PLOT	Graphics system for use with BASIC or SBASIC
SBASIC	Structured BASIC
SIX	FORTRAN 76
SNOBOL	DTSS SNOBOL4

As a user under SIMON manipulates his current file, various bits of information about the file are accumulated. For example, the file may become nonlistable, it may contain source code pertaining to a specific compiler, or the user may wish to specify that only certain SIMON commands may be executed on the file, or that only certain users should be allowed to access the file. When a user saves his current file, SIMON preserves these information bits by setting them into the access word with which the file is catalogued. Thereafter, each time SIMON reopens the file for a user, it retrieves these information bits from the file access word and uses them to control the way in which it executes many user commands pertaining to the file. These information bits are referred to as the 'Simple Monitor User bits'.

The Simple Monitor User bits are accumulated for the user's current file during the execution of RUN, COMPILE, EDIT, TEXTEDIT, STRINGEDIT, SAVE, and SYSTEM commands. There are nine bit positions involved, and SIMON maintains two complete sets of information bits for each saved file. One set of information bits is used when the file is opened without a password, and the other when the file is opened with a password. The two sets of information bits are placed in the user bit positions (bits 9-17 and 27-35) of the file's access word when the user saves his current file via a SAVE command. The information bits are subsequently retrieved from either the no-password side (bits 9-17) or the password side (bits 27-35) of the file's access word when the user opens the file via an OLD command.

The Simple Monitor User bits are divided into two fields:

Current file permission bits (bits 9–13).
These bits control the way in which SIMON executes OLD, LIST, RUN, REPLACE and SAVE commands pertaining to the user's current file, and the way in which the various text editing systems perform editing functions on the file. They are defined in Table 6.4.

Table 6.4.

Name	Meaning
B$LIST or list	The file may be listed.
B$COMP or compile	The file is compiled object code though not necessarily directly executable.
B$PUB or public	The file is available to the public.
B$GROUP or group	The file is available to the user's group (users whose numbers differ only in the last three characters).

System code (bits 14-17).
These bits indicate what the user's current system was when the file was saved. Only compilers and interpreters are assigned unique system codes; assemblers and other official systems are assigned a system code of zero (see section 4.5.3). SIMON inspects a file's system code when the file is opened via an OLD command.

A complete list of SIMON commands is given below:

Elementary file commands – The following SIMON commands perform elementary file operations:

APPEND
BUILD
ENTER
IGNORE
LIST
MIGRATE
NEW
OLD
RECOVER
RENAME
REPLACE
SAVE
SCRATCH
SORT
UNSAVE

Program execution commands – The following commands initiate the execution of user programs:

BACKGROUND
COMPILE
DEBUG
EXECUTE
GCOS
LINK
RUN
TEST

Editing commands – The following commands perform text editing functions:

EDIT
QEDIT
STRINGEDIT
TEXTEDIT
TYPEDIT

Terminal mode commands – The following commands define the format of the data stream to or from the user's terminal:

DIRECT
FRIDEN
FULLDUPLEX
HALFDUPLEX
KEYBOARD
NFRIDEN
NPARITY
PARITY
TAPE

Miscellaneous commands – The following commands do not fit neatly into any other category:

ACCOUNT
BILL
BRIEF
BYE
CALCULATE
CATALOG
EXPLAIN
GOODBYE
HELLO
HELP
JOIN
LENGTH
MAIL
MAXIMUM
MONITOR
NBRIEF
PERFORM
PREFERENCE
PUNCH
SYSTEM
TTY
USERS

Diagnostic commands – The following diagnostic commands are available only to users who have special system privileges:

ABORT
CLOSE
CORE
DIE
DUMP
PATCH
STATUS

SIMON communicates with its environment in the following ways:

SIMON communicates with the DTSS Executive via Executive calls. By issuing an Executive call with various parameters, SIMON requests the DTSS Executive to perform a given operation; the Executive performs the requested operation and returns status information.

SIMON communicates with TCFACE via communication files. There is one communication file between TCFACE and SIMON for each user that SIMON is servicing. TCFACE handles the transfer of information between the Terminal Communications Controllers (TCCs) and various slave mode programs in the system.

SIMON communicates with LOGIN by means of a single communication file and also via information accompanying user catalogs which LOGIN passes to SIMON.

SIMON communicates with jobs running under it in various ways, including communication files, parameters set when a job is run, and status returned when a job terminates. Jobs which run under SIMON include compilers and editors.

SIMON communicates with BAKMON via an intermonitor message facility provided by LOGIN. The Background Monitor runs jobs which are not connected to time-sharing terminals and handles the allocation of peripherals to these jobs.

SIMON will service as many users as are passed to it by LOGIN. Information about each user is maintained in a separate user control block known as a JBLOCK; everything SIMON knows about a given user is contained in his JBLOCK. Each JBLOCK is fifty-four words long (octal) and contains the information given in Table 6.5.

From SIMON's point of view a user is always in one of four states. He is either:

(a) executing SIMON code (for example, having his command line scanned),
(b) on SIMON's processing queue waiting to resume execution,
(c) in limbo waiting for the trap of an Executive call to link him to the processing queue (for example, waiting for a REPLACE MME to trap), or
(d) on a subsidiary resource queue waiting to use a given subsidiary resource (for example, waiting to use the sort buffer).

6.3.1.1 *The processing queue*

SIMON can execute only one instruction at a time, and thus can process only one user at a time. A user becomes eligible for processing when an Executive call traps, or when his JBLOCK reaches the front of a subsidiary resource queue. In either case, SIMON does not process the user immediately, but places his JBLOCK at the back of the processing queue. Before a JBLOCK is placed on this queue, the address of the routine which will process it next and any parameters necessary for this processing are recorded in it. Then when SIMON becomes free to process a new task, either as a result of issuing a trapping Executive call, or as a result of placing a JBLOCK on a queue, the JBLOCK at the front of the processing queue is removed and the task recorded in it is executed.

6.3.1.2 *Subsidiary resource queues*

There are several resources other than processor time for which SIMON must queue. SIMON maintains the following subsidiary resource queues:

Short buffer queue – This queue serialises use of a 2098-word buffer used to search the user's current or alter file for a given line number in response to a 'LIST <line number>' command.

Table 6.5

Word (octal)	Contents
0	Trap status word one – The location in which the Executive places status information at the completion of an Executive call.
1	Trap status word two.
2	Trap interrupt IC/IR – The Executive preserves Simple Monitor's instruction counter and indicators here when it interrupts SIMON on completion of an Executive call.
3	Trap processing instruction – The Executive transfers to this instruction after filling in trap words 0, 1, and 2 at the completion of an Executive call. This instruction calls SIMON's trap processing routine, which links the user control block to SIMON's processing queue (see Resource Queues below).
4 (bits 0-8)	Index of current command – Offset into SIMON's command table of the user's last command.
4 (bits 9-13)	Current file status – The current file's permissions (for example, Read, List, Compile).
4 (lower)	Restart address – Address at which to resume processing when the JBLOCK reaches the front of the processing queue.
5 (upper)	JBLOCK link word – Pointer to the next JBLOCK on the linked list containing all JBLOCKS (JCHAIN).
5 (lower)	Terminal frn – File reference number of the user's terminal communication file.
6 (upper)	Multiterminal system link word – Pointer to the next multiterminal system master on a linked list of all multiple-terminal system masters.
6 (lower)	Abortable frn – File reference number of a file which has an operation pending on it which may be aborted (for example, the frn of a job running under SIMON, the user's current file frn when a LIST command is in progress).
7 (upper)	Subroutine linkage number one – Location for return address when calling a delaying subroutine which uses subroutine linkage number two.

Table 6.5 – *continued.*

Word (octal)	Contents
7 (lower)	Current catalog frn – File reference number of user's current catalog,
10 (upper)	Subroutine linkage number two – Location for return address when calling a delaying subroutine which uses subroutine linkage number three.
10 (lower)	Current file frn – File reference number of user's current file.
11 (upper)	Subroutine linkage number three – Location for return address for subroutines which do not call any delaying subroutines.
11 (lower)	Alter file frn – File reference number of user's alter file.
12 (upper)	Input buffer address – Address of general-purpose I/O buffer currently allocated to this user.
12 (lower)	Extra file frn – File reference number of a temporary 'extra' file (for example, an output file passed to an editor, a user catalog during a fetching open).
13 (upper)	Command sequence buffer address – Address of a buffer containing the current command sequence (multiple command line).
13 (lower)	Communication file frn – File reference number of the master end of a communication file to a job running under SIMON.
14 (upper)	Output buffer address – Address of a buffer containing output for a user (for example, an error message, the 'READY' prompt).
14 (lower)	User catalog frn while user is entered – File reference number of the user's own user catalog while he has entered another catalog.
15 (upper)	Pointer to current system name – Address of the table entry for the user's current system. This table includes the current system's name, file reference number, system code, and length.
16-17	Current file name – The eight-character name of the user's current file.

Table 6.5 – *continued.*

Word (octal)	Contents
22-23	Entered catalog name – Name of user's current catalog while entered.
24	User permission word – Bit-coded word passed to SIMON by LOGIN which controls the user's ability to access various system facilities.
25	User quotas word – Quotas word passed to Simple Monitor by LOGIN limiting number of Computer Resource Units and amount of catalogued and scratch storage that a user may employ while running under SIMON.
26 (upper)	User's current memory limit – The current maximum amount of main memory which a user may occupy while running under SIMON. This value is initially the system default but may be changed by the MAX command.
26 (lower)	User's current resource limit – The current maximum number of Computer Resource Units which a user may employ while running under SIMON. This value is initially the user's validated limit but may be reduced by the MAX command.
27 (upper)	Time since bootload at log-on – Time since bootload when the user logged on under SIMON.
27 (lower)	Time since bootload at alter file copy or transfer to command routine – Elapsed time since bootload when SIMON began processing the user's last command, or when SIMON initiated the last copy to the user's alter file, whichever was later.
30-37	Temporary storage – Temporary storage used to pass information between subroutines and to save information across Executive calls.
40	Session's Computer Resource Units – The total number of Computer Resource Units which the user has accumulated since he logged on under SIMON.
41	Command Computer Resource Units – The number of Computer Resource Units which the user has accumulated during processing of the current SIMON command.

Table 6.5 – *continued.*

Word (octal)	Contents
42	Command line Computer Resource Units – The total number of Computer Resource Units accumulated by the user during the current SIMON command sequence.
43	Command line Input/Output Units – The total number of input/output units accumulated by the user during the current SIMON command sequence.
44	Command line Central Processor Units – The total CPU time accumulated by the user during the current SIMON command sequence.
45	Command line Kilo-Core Seconds – The total number of kilo-core seconds accumulated by the user during the current SIMON command sequence.
46 (bits 0-8)	Current command number – The number of the command in the user's SIMON command sequence which is currently being processed.
46 (bits 9-35)	Current file length – The length of the user's current file.
47 (bits 0-8)	Unrequested input length – Length of user's unrequested input (current command length).
47 (bits 9-35)	Alter file length – Length of the user's alter file.
50	User status flags – Bit-coded flags indicating presence or absence of various files and buffers, a break received from the user's terminal communication file, etc.
51 (upper)	Run routine status flags – Bit-coded status flags for SIMON's general run routine indicating edit run, compiled current file run, compilation run, etc.
51 (lower)	Executable system frn address – Address of the file reference number of an executable system for SIMON's general run routine.
52 (upper)	Address of slave JBLOCK – Address of slave ('fake') user during PERFORM command.
52 (lower)	Address of master JBLOCK – Address of master ('real') user during PERFORM command.
53 (upper)	Line number – System access line number assigned to the user when he logged on.
53 (lower)	Unused.

Catalog queue – This queue serialises use of the 'non-reentrant' RUN subroutine.

LOGIN communication file queue – This queue serialises use of SIMON's communication file to LOGIN, by which billing messages, requests for user catalogs, etc. are transmitted. This queue is also used to serialise certain log-on and log-off operations, to make sure that SIMON does not become confused.

If a user wishes to use a subsidiary resource and that resource is not busy, the resource is marked busy and is seized immediately. If a user wishes to use a subsidiary resource and the resource is busy, the user's JBLOCK is placed at the back of the given subsidiary resource queue. When a user finishes using a subsidiary resource, the resource is realised in one of two ways. If there are no more JBLOCKS on the queue it is simply marked not busy. If there are more JBLOCKS on the queue, the queue remains busy and the JBLOCK at the front of the queue is moved to the back of the processing queue; when this JBLOCK reaches the front of the processing queue, it seizes the resource.

The format of the head of a queue is given in Table 6.6. The format of a queue block is given in Table 6.7. Fig. 6.9 shows the queue format.

Table 6.6 – Format of the head of a queue.

Word (octal)	Contents
0 (upper)	First pointer – pointer to the link word of the first block on the queue.
1 (upper)	Last pointer – pointer to the restart word of the last block on the queue.
2	Count – The number of blocks current in the queue.
3	Maximum count – The maximum number of blocks which have been in the queue since the last STA command was received from LOGIN (at which time this word was zeroed).
4	Unused.
5 (upper)	Busy pointer – Pointer to the block which currently holds the queue.
6-7	Instructions to add a block to the queue.
10-11	Instructions to sieze the queue.
12-13	Instructions to release the queue.

Table 6.7 – Format of a queue block.

Word (octal)	Contents
0	Status word one – The location in which the Executive places status information upon completion of an Executive call.
1	Status word two.
2	Return word – The location in which the Executive preserves SIMON's instruction register and indicator register when it interrupts SIMON upon completion of an Executive call.
3 (upper)	Link word – Pointer to the next block on the queue. If this is the last block on the processing queue this field is zero.
4 (lower)	Restart address – The address at which to resume processing when this block reaches the front of the processing queue.
.	
5 (n–1)	The rest of the block, where n is its length.

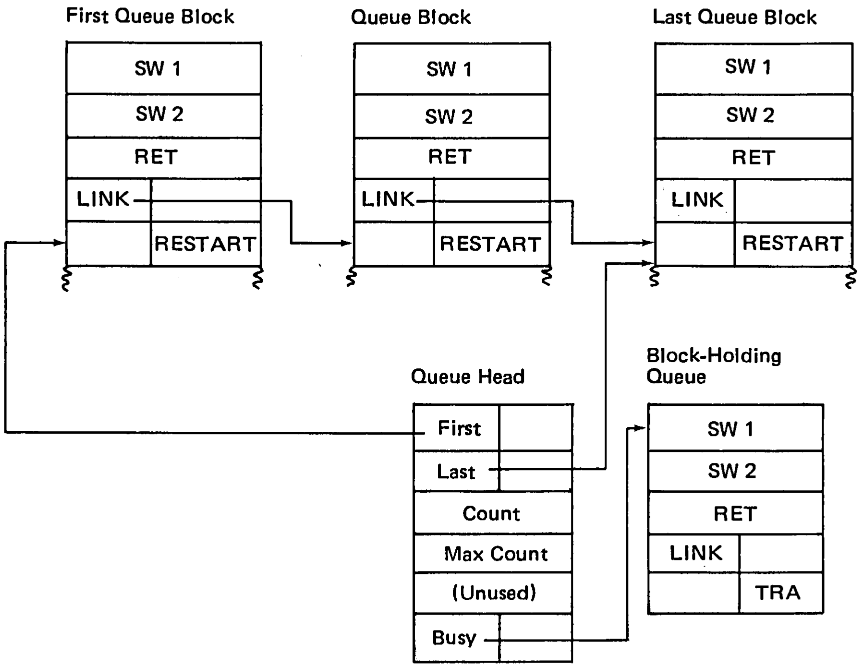


Fig. 6.9 – Queue Format.

The path that a user follows through SIMON is usually quite straightforward. As soon as a user logs on, or as soon as the execution of a command has been completed, the user is transferred to the command routine. From there the typical user path is composed of the following steps:

(a) If a STOP message is required, print it and throw away the rest of the user's command sequence.
(b) If there are more commands to be executed from the user's command sequence, go to step (h); otherwise:
(c) Print any I/O and CPU time charges accumulated during execution of the command sequence.
(d) Drive command mode on the user's terminal communication file.
(e) Print READY.
(f) Issue a copy from the user's terminal communication file to the user's alter file.
(g) Read the user's command line to a buffer.
(h) Decode the user's next command and transfer to the proper command processing subroutine.
(i) Execute the user's command.
(j) Return to the command routine, step (a).

The usual mode of communication between a user and SIMON is command mode. SIMON sets command mode on the user's terminal by driving command mode on his terminal communication file immediately before printing NEW or OLD–, WHAT?, or READY. SIMON then issues a copy from the user's terminal communication file to the end of his alter file.

In command mode, the TCC Executive sends all input lines that begin with a line number to TCFACE, in blocks of 256 characters. TCFACE writes this data to the user's terminal communication file with SIMON, but does not trap the copy issued by SIMON. Thus the data are transferred from the user's terminal to his alter file with no additional action being taken by SIMON Monitor.

When the TCC Executive receives a line of input that does not begin with a line number, it informs TCFACE that command mode has terminated, and TCFACE traps SIMON Monitor's copy to the user's alter file. When SIMON receives the trap indicating completion of the copy, it assumes that the next line of input from the user's terminal communication file will be user command.

The TCC Executive sends the non-line-numbered line to TCFACE. As there is no longer a copy outstanding on the user's communication file with SIMON, TCFACE issues a drive out this file. SIMON receives this as a special interrupt and responds by issuing a read on the appropriate communication file.

SIMON's read causes TCFACE to receive a special interrupt, to which it responds by writing the user's non-line-numbered line to the communication file. SIMON receives this line and interprets it as a user command. If it is a valid command, SIMON executes it, drives command mode on the user's terminal communication file, and prints READY. If the command is not valid, SIMON drives command mode and prints WHAT?

Thus, for example, if the user types RUN, the following actions take place in SIMON:

TCFACE sends a special type 1

.
.
.
.

SIMON issues a READ on the communication
file to get the command line

.
.
.
.

Executive traps the READ
SIMON queues the trap block on processing queue

.
.
.
.

SIMON checks the command line and then
issues a RUN MME

.
.
.
.

Exective traps the RUN
SIMON queues the trap block on processing queue

.
.
.
.

SIMON issues a REQUEST STATUS to get run-time etc.

.
.
.
.

Executive traps REQUEST STATUS
Simon queues trap

.
.
.
.

and so on from step (c) in the path described above

When SIMON discovers an internal inconsistency, it requests a system 'crash' by executing a master mode instruction. If the Executive allows SIMON to interrupt time sharing, the crash-restart section of the Executive copies all of memory (including SIMON's memory) into :DUMPCAT:CDUMP. If SIMON is not allowed to interrupt time sharing, it copies its memory into :DUMPCAT:SIMON, and terminates with a bad status.

SIMON copies its memory into a dump file to enable a systems programmer to diagnose and correct the internal inconsistency which caused the dump. To aid the systems programmer in this process, a dump formatter named —SIMON is saved in :DUMPCAT.

The dump formatter is run by the DUMP module whenever SIMON dumps information into :DUMPCAT:SIMON. It is also run when SIMON causes a full CDUMP by halting the system. The dump formatter can also be executed in Foreground with a SIMON Core DUMP image as the current file. (In this case output is directed to frn 4 and becomes the current file on termination). The dump formatter formats the following information:

1. Type and location of fault.
2. Indicator register at time of fault.
3. Code causing fault.
4. Registers at time of fault.
5. Release date.
6. Assembly date.
7. Date and time of crash.
8. Checkpoint queue.
9. Master task (processing) queue.
10. LOGIN communication file queue.
11. Log-on request status queue.
12. Sort buffer queue.
13. Catalog queue.
14. Multiple-terminal system structure.
15. Special interrupt stack.
16. Free storage.
17. Initialisation request status data.
18. JBLOCK of user who caused crash.
19. Status of all users logged on to module at time of crash.

As was noted in section 4.2, when SIMON runs a job it passes a number of files to the new job. The files and other information passed depend on the type of job. Listed below are some of the various interfaces provided by SIMON to the jobs it runs.

STANDARDISATION RUN INTERFACE

Frn	*Description*
1	Terminal (Return, Read, and Append).
2	Current catalog (Return, Catalog, Owner, Search, Read, Append).
3	Current file (Return, Read, Execute).

Registers

X0-X3	Zero
X4-X7	Current catalog name (if entered); otherwise, user number.
AQ	Current file name.
E	400000 – Running an experimental system.
	100000 – User is in Brief Mode.
	040000 – User has entered a catalog.

EDIT INTERFACE

Frn	*Description*
1	Terminal (Return, Read, Append).
2	Current catalog (Return, Catalog, Owner, Search, Read, Append).
3	Current file (Return, Read, Execute).
4	Output file (Return, Read, Write, Append).
5	Communications file (Read).

Registers

X0	Current file user bits and current system code.
X1-X3	Zero.
X4-X7	Current catalog name (if entered); otherwise, user number.
AQ	Current file name.
E	400000 – Running an experimental system.
	100000 – User is in Brief Mode.
	040000 – User has entered a catalog.

Termination status

400000	Edit not successful.
200000	Swap output file with current file.
100000	Sort output file.
040000	Make output file not listable.
020000	Rename current file '.RESULT.'.

COMPILER INTERFACE

Frn	*Description*
1	Terminal (Return, Read, Append).
2	Current catalog (Return, Catalog, Owner, Search, Read, Append).
3	Current file (Return, Read, Execute).
4	Output file (Return, Read, Write, Append).

Registers

X0	000003
	000043 ('SIZE' option).

X3-X3 Zero.
X4-X7 Current catalog name (if entered); otherwise, user number.
AQ Current file name.
E 400000 – Running an experimental compiler.
100000 – User is in Brief Mode.
040000 – User has entered a catalog.

Termination status

200000 Compilation successful, swap output file and current file, rename output file '.OBJECT.'.
010000 Object code is executable.

COMPILED CURRENT FILE INTERFACE

Frn	*Description*
1	Terminal (Return, Read, Append).
2	Current catalog (Return, Catalog, Owner, Search, Read, Append).

Registers

X0-X3 Zero.
X4-X7 Current catalog name (if entered); otherwise, user number.
AQ Current file name.
E 400000 – Current system is an experimental system.
100000 – User is in Brief Mode.
040000 – Under has entered a catalog.

MAJORSORT INTERFACE

Frn	*Description*
1	Terminal (Return, Read, Append).
2	Output file (Return, Read, Write, Append).
3	Alter file (Return, Read).
4	Current file (Return, Read, Execute).

Registers

X0 000001 – Sort only alter file.
X1-X3 Zero.
X4-X7 Current catalog name (if entered); otherwise, user number.
AQ Current file name.
E 400000 – Running experimental Majorsort.
100000 – User is in Brief Mode.
040000 – User has entered a catalog.

Termination status

040000 System storage exhausted.

020000 Scratch quotas exceeded.

010000 Alter file empty (or alter file contained nothing but carriage return – line feed pairs and/or nulls).

004000 Majorsort coding error.

002000 Current file out of order.

001000 File system error.

DO INTERFACE (For the PERFORM module)

Frn	*Description*
1	Terminal (Return, Read, Append).
2	User catalog (Return, Catalog, Search, Read, Append).
3	Command file (Return, Read, Execute).
4	Communications file (Read).

Registers

X0-X3	Zero.
X4-X7	Current catalog name (if entered); otherwise, user number.
AQ	Current file name.
E	400000 – Running experimental Do module. 100000 – User is in Brief Mode. 040000 – User has entered a catalog.

MULTIPLE TERMINAL SYSTEM INTERFACE

Frn	*Description*
1	Terminal (Return, Read, Append).
2	Current catalog (Return, Catalog, Owner, Search, Read, Append).
3	Current file (Return, Read, Execute).

Registers

X0-X2	Zero.
X3	Number of joiners allowed.
X4-X7	Multiterminal subsystem name.
AQ	Current file name.
E	400000 – Running experimental multiple-terminal system. 100000 – User is in Brief Mode. 040000 – User has entered a catalog.

BACKSYS INTERFACE

Frn	*Description*
1	Terminal (Return, Read, Append).
2	User catalog (Return, Catalog, Search, Read, Append).
3	Current file (Return, Read, Execute).

Registers

X0-X1	User permissions word.
X2-X3	User quotas word.
X4-X7	User number (may not be entered).
AQ	Current file name.
E	400000 – Running experimental Backsys.
	100000 – User is in Brief Mode.

Termination status

100000	Awake BAKMON.
020000	Awake SBKMON

6.3.2 OPMON

The Operations Monitor, OPMON, runs from the same source code as SIMON. In order to use OPMON a user must have the P$CTY bit in his permissions word (see Table 4.3) and must be using a terminal on a line designated a control line. SIMON is a non-privileged monitor having only P, B, and X permissions, OPMON is highly privileged, having all nine bits (see section 4.5.4). In addition, OPMON will accept the following commands not valid under SIMON:

CTY	Designates a port to be a control line.
DCTY	Give access to TCC memory for debugging.
DEDICATE	Restrict use of system to those running under OPMON.
DISABLE	Stop users or certain lines from logging on.
ENABLE	Permit users on certain lines to log on.
KILL	Stop a named monitor.
LCORE	Dump LOGIN memory.
LDUMP	Print contents of specified location in LOGIN memory.
LPATCH	Patch LOGIN memory.
NDCTY	Cancel a DCTY.
PROTECT	Initiates DTSS shutdown.
PURGE	Run the purge module.
UNDEDICATE	Opens DTSS for general use.

6.3.3 BAKMON [3], [4], [5]

Background provides a facility for running programs which does not tie up the user's terminal for the duration of the run. Under Background, the user submits a file containing a request for a particular program or sequence of programs to be run. The user may then sign off or continue to use his terminal for other purposes while his background job is being processed.

Background has two components. The background system (BACKSYS), the interactive portion of background is run by SIMON in response to a BACK command. It converts background job requests, written by users in the background language, into background job descriptions, which are placed in a special submission catalog. The background monitor (BAKMON) is the non-interactive component of background. It runs continuously and processes each job description in the submission catalog as resources become available.

A user submits a request for background service by signing on to DTSS under SIMON and by composing a 'background program' which specifies the activities to be performed. He then saves this program and issues a BACK command. This command causes SIMON to run BACKSYS. As it compiles, it checks for errors and user permissions and opens files specified in the program checking for passwords and tape names, if required. If any errors are detected by BACKSYS as it compiles the user's program, BACKSYS types appropriate error messages and returns control to SIMON. The user may then correct his mistakes, REPLACE his program and type BACK once more. When his program is free of errors, BACKSYS types

```
***JOB ACCEPTED
```

and returns control to SIMON. Note that this message implies nothing about the correctness of programs to be run in background, but merely indicates that the file containing the description written in the background language has no errors in it. At this point the user us free to do as he pleases; if he wishes, he may log off the system.

Whenever BACKSYS successfully compiles a background program into a background job description, the job description is catalogued in :BAKCAT :SUBMIT. Later BAKMON uncatalogs it, recatalogs it in :BAKCAT:RUNNING, and proceeds to act on it. As system resources become available (that is, a peripheral is freed or surplus central processor time is discovered), BAKMON initiates the activities at the head of the queue for the free resources. These activities are performed simultaneously with the normal time-sharing activity; if operator assistance is required by any activity, for example, to mount a tape, the BAKMON communicates with the operator via a control terminal or the console typewriter.

As activities in a background job are completed or aborted, BAKMON appends appropriate messages to the end of the user's saved background program. If more activities remain in the background job, the monitor returns the remainder of the background job description to the queue of jobs awaiting execution. If the activity just completed is the last one in the background job description, the monitor also appends a message indicating that the entire job has been completed. If through writing to files a user's catalog storage maximum is exceeded, the job is aborted when BAKMON next attempts to append a message to the saved background program. If the catalog is under two times its maximum, BAKMON attempts to overwrite the last line of the job description by the error message 'QUOTAS EXCEEDED'. No further activities are executed.

When a user wishes to find out if his background job has been completed, he signs on to the system, calls up his background program and list it (using, say, EDIT LIST) to get the latest progress reports that BAKMON has appended. If a user has been careless enough to unsave his background program before its execution has been completed, the job is aborted when BAKMON next attempts to append a message to the background program.

When a user's background job has terminated, he may obtain the output from that job either by listing appropriate files or by picking up peripheral output at the computation center. The user may also unsave his background program at this point, though he may not wish to do so if he intends to submit the same job again in the future.

BAKMON is responsible for allocating all peripherals during time-sharing. A background user with appropriate validation permissions can access the printer, magnetic tapes, remote peripherals and card punch via the appropriate commands and specifiers.

The operator is able to inform BAKMON of the availability of peripherals through a control terminal; BAKMON assumes that a set of peripherals is available at the beginning of the day.

BAKMON is capable of running a large number of jobs at the same time. These jobs run in the time-sharing environment and compete for computer resources on an equal basis with foreground jobs. If BAKMON began running every background job as soon as it was submitted, then foreground response time would be very uneven and often very slow. Therefore, in order to maintain reasonable response time for foreground users, BAKMON must limit the number of running background jobs. The process that it uses to do this is called background scheduling.

A typical background job consists of several activities (job-steps) which are to be run serially by BAKMON. When this typical background job is submitted, it joins a queue of background jobs that are waiting to be run. BAKMON continually selects a job from this queue, runs the next activity of the job, and then returns the job to the queue of waiting jobs. While carrying out this process, BAKMON has two key decisions to make: how many jobs to run concurrently, and which job to run next.

The rules used to make these decisions are called the background scheduling algorithm. The scheduling algorithm controls how and when a job's activities are initiated; once an activity is running it is no longer affected by the scheduling algorithm.

There are three types of background activitiees, and BAKMON has a different scheduling algorithm for each type. The simplest type consists of the miscellaneous activites (ABORT, STATUS, WAIT), which are initiated unconditionally. Next are the peripheral activities (COPY, PRINT, PUNCH, READ, SCRATCH, UNSAVE, WRITE), which are initiated as soon as all the required peripherals are available. Finally, there are the execution activities (DEBUG, RUN).

Each execution activity in background has a maximum memory size and a maximum number of CRU's which may be consumed. These resource maximums may be specified in the optional fields of the RUN command or by a MAX statement. If no maximums are specified, then the default maximums of 16K words of memory and 32 CRU's are assumed.

A typical set of resource limits of BAKMON are shown in Table 6.8.

Table 6.8 – Typical set of resource limits of BAKMON.

Time of day	CRU limit	Memory limit
until 0700	1024	128K
0700-0800	512	128K
0800-1200	128	128K
1200-1300	512	128K
1300-1700	128	128K
1700-1900	1024	128K
1900-0100	unlimited†	128K
0100-0200	1024	128K
0200-0300	512	128K
On Saturday, Sunday and holidays:		
0700-0100	unlimited†	128K
0100-0200	1024	128K
0200-0300	512	128K

† unlimited = 100000

BAKMON will not initiate any execution activity whose resource maximums exceed the resource limits in effect at the time. The limits are set so that execution activities which consume large amounts of resources will not be initiated just before system shutdown or during periods when forground use is normally heavy.

BAKMON uses 'job slots' to limit the number and size of the execution activities running at any given time. The number of job slots is a function of the number of foreground users. A typical set of values is shown in Table 6.9.

Table 6.9.

Foreground users	Job slots	Per size				
		1	2	3	4	5
140-	2	0	1	1	0	0
120-139	3	1	1	0	1	0
100-119	4	1	1	1	1	0
80-99	6	1	2	1	1	1
60-79	9	2	2	2	2	1
40-59	13	2	3	3	3	2
20-39	16	2	3	3	3	5
0-19	28	2	2	2	2	20

BAKMON can run one execution activity of a certain size in each job slot. The 'size' of the activity is the product of maximum memory size and the maximum CRU of the activity as shown in Table 6.10.

Table 6.10.

Size	CRU* CORE/10000
1	0-128
2	129-512
3	513-4096
4	4097-32768
5	32769-

For example, an activity which uses the default maximums of 16K memory and 32 CRU is size 2, as determined by the equation (16K*32)/1000 = 512. Such an activity could run in a job slot of size 2 or larger.

When BAKMON has an empty job slot, it examines the next activity of each waiting job. The job selected to be run will be the first job which meets all of the following criteria:

- the next activity is an execution activity;
- the size of the activity is less than or equal to the size of the available job slot;
- all the required peripherals are available;

- the activity's resource maximums are less than or equal to BAKMON's current resource limits.

The purpose of the job description is to specify to BAKMON and the various modules running under BAKMON exactly what actions the user wishes to have performed, and in what order.

The job description consists of a job description header and one or more activity descriptions. Each activity description is divided into an activity description header and several file descriptions as shown in Table 6.11.

Table 6.11.

Job description header	Activity description header
Activity description Activity description Activity description Num program	File description File description File description File description

The job description header is a fixed length (fourteen-word) block containing information relating to the job description as a whole as shown below:

0: 1:	user number	
2: 3:	background job file name	
4: 5:	background job file password	
6: 7:	job reference name	
10:	pointer to next activity	job description length
11:	progress report line number	
12: 13:	name of the system which compiled this job description	

where:

Word 10 upper: The location within the job description of the next activity to be processed. Originally set to the first activity following the job description header, it is reset to succeeding activity.

Word 11: The line number BAKMON will use when writing the next progress report message. It is originally set to 10000, but is altered by BAKMON as reports are appended.

Each activity description within the job description is prefixed by a variable length activity description header containing information pertinent to this activity alone.

The header is composed of:

- The name of the system to be run for this activity, for example BASIC or PRINTER.
- Length of the activity header.
- Length of the activity description.
- The line number at which he major command for this avtivity may be found in the user's background program.
- Memory size limit in words set by the user via a background MAX command.
- Run-time limit in seconds for this activity set by the user via a background MAX command.
- A binary number specifying the type of activity:

  0 – Old Foreground simulation.
  1 – BMC (background media conversion).
  2 – General run.
  3 – GCOS simulation.
  4 – SYSOUT run.
  5 – Abort activity.
  6 – Wait activity.
  7 – Status activity.
  8 – Foreground simulation.
  9 – Error.

- A set of activity flags specifying certain information about the activity. The only bits currently in use for this purpose are:

  000001 – Do not output a header to the output file in a foreground simulation.
  000002 – Executes next activity only (that is, cleanup).
  000004 – A limit has been set by the user on run time. A check is made for the value of the limit.
  000010 – A limit has been set by the user on core size. A check is made for the value of limit.

The max claim vector is an eight-word block used to prevent system deadlocks. The entries are indexed by resource type as follows:

Word 0: Printer.
Word 1: Remote peripheral.
Word 2: Card punch.
Word 3: 9-track tape handler.
Word 4: 7-track tape handler.
Word 5: Special printer.
Word 6: Scratch tape.
Word 7: Saved tape (not really a resource, used as a consistency check against saved tape file descriptions).

Since scratch tapes are currently the only resource that can be saved from one activity to the next, they are the only resource that can cause a system deadlock. BAKMON will allocate resources so as to avoid system deadlocks if the resource request vector R and max claim vector M for each activity A(i) is set up as follows:

Let R(r) (upper) = the quantity of resources of type r that will be used in activity A(i). The vector R(r), $0 \leqslant r \leqslant 7$ is the resource request vector for activity A(i) (see below).

Look at all activities A(j) such that a scratch tape is saved in activity A(p) and not saved until activity A(q), $p \leqslant i \leqslant q$, and $p \leqslant j \leqslant q$. Let the scratch tape max claim M(6) (lower) for activity A(i) equal the maximum for the scratch requests R(6) (upper) of activities A(j). For resource types other than scratch tapes, the max claim for an activity simply equals the resource request for that activity.

The resource request vector is an eight-word block and specifies exactly which and how many of each resource are to be used in the activity. It is indexed by resource type in the same way as the max claim vector. The entry for each resource type has the format:

Upper: The number of resources of this type that are needed in the activity.
Lower: Bits specifying which of the resources of this type may be used to fill the request given in the upper half-word. If any of the available resources of the given type may be used, this field is set to zero. If only resource numbers x and y may be used, then this field is set to 2^x + 2^y.

The run register block is an eight-word block containing the initial registers to be set when the system named in the activity description header is run.

The indent pointers block is a variable length block which exists only if the activity is a BMC activity. It provides pointers to the file descriptions for the files/peripherals (usually an input and an output file/peripheral) involved in the data transfer. The block contains:

- Number of files in the data transfer (usually two).
- Pointer to the source file description, relative to the beginning of the ident pointers block.
- Pointer to the destination file description, relative to the beginning of the ident pointers block.
- Pointer to the next file description, relative to the beginning of the ident pointers block.

The file description is a block of variable length consisting of a fixed-length (twelve-word) file description header and variable length (possible zero length) ident block.

Each file description identifies a file or peripheral that will be passed to the system that is run in the activity. The collection of file descritpions in an activity description defines the pass list for the system to be run.

Foreground simulation and BMC activities each have a particular form for the activity description, since certain special files must be passed which are not specified in the background program.

1. The general form of a foreground simulation activity is:

   1 Activity header – FORGSIM is the system name.
   2 File description for a communications file which BAKMON uses to fake the operation of a terminal.
   3 File description for the job description file.
   4 File description for the user's catalog.
   5 File description for the source file (OLD file).
   6 File description for the input file.
   7 File description for the output file.
   8
   : Extra file descriptions – may or may not be here.
   N

   The file descriptions for the OLD, input and output files may occur in any order. If the user specifies an input file, he must specify an output file. If he does not specify an OLD file, BACKSYS inserts a file description for a dummy file with the name .NONAME..

2. The general form of a BMC activity description is:

   1 Activity header – contains a pointer block.
   2 File description for a communications file which the module uses to issue mounting instructions, send billing data, etc.
   3 File description for the job description file which the module reads to find format information, IDENT data, etc.

4 File description for the source file/peripheral.
5 File descritpion for the destination file/peripheral.
6-7 File description for conversion table. Their presence is indicated by the D$CTAB bit in the input and output files file description header. Either or both may be omitted.
8
: Extra file descriptions – may or may not be here.
N

The file descriptions for the source desination files may occur in any order, but both must be present.

3. The general form of an abort activity is:
   1 Activity header – system name has name of job to be aborted.

4. The form of a general run is:
   1 Activity header – no pointer block.
   2
   : File descriptions.
   N

5. The form of a status activity is:
   1 Activity header – system name has name of job whose status is to be appended to the user execution report.

6. The form of a wait activity is:
   1 Activity header – contains the date and time of the run in the first two
   2 words.

There is a background job description formatting program available for debugging purposes under the experimental version of the background system XBACK. A sample run of this is shown below:

```
BRIEF
*NEW BACKPROG
*10 PRINT MYFILE
20 USING LOWER
30 FORMAT NSLEW
40 END
SAVE
*RUN XBACK
***JOB ACCEPTED
*OLD :BACKCAT:XXSUBMIT:BACKPROG
*RUN XJOBDESC
job description file
?
```

```
user number:           HDK52500
job name:              BACKPROG
password:
reference name:        BACKPROG
next activity at word: 14
job desc. length:      152
report line number:    10000
compiled by:           XBACK

activity description (begins at word 14)

system:                XSPRINTR
header length:         42
activity desc. length: 136
activity line number:  10
core limit:            0
runtime limit:         0
activity type:         BMC
activity flags:        000000

resource requests              max claim
0 printers             ()      0
0 remote peripherals   ()      0
0 card punches         ()      0
0 9-track handlers     ()      0
0 7-track handlers     ()      0
1 special printers     ()      1
0 scratch tapes        ()      0
0 saved tapes          ()      0

indent pointers block (begins at word 32)
2 files involved in BMC
source          file description begins at word 102
destination     file description begins at word 124

file description (begins at word 56)

description length:    12
filename:
password:
accesses:              000000
passlist position:     # 0 (FRN 1)
file type bits
o    disc file
o    communications file to BAKMON

file description (begins at word 70)

description length:    12
filename:
password:
```

```
accesses:               000000
passlist position:      # 1 (FRN 2)
file type bits
o    disc file
o    job description

file description (begins at word 102)

description length:     22
filename:               MYFILE
password:
accesses:               001000
passlist position:      # 2 (FRN 3)
file type bits
o    tally treename
o    disc file

filename tally:         2,6,0
"MYFILE"

ident block (begins at word 116)

format bits:            000000000000
tape mode bits:         000000
user´s limits:          0
block size:             0
record size:            0
ident block length:     0
report text length:     0

file description (begins at word 124)
description length:     26
filename:
password:
accesses:               000000
passlist position:      # 3 (FRN 4)
file type bits
o    peripheral

file is a special printer    ()

ident block (begins at word 136)

format bits:            000020000000
tape mode bits:         000000
user´s limits:          0
block size:             0
record size:            0
ident block length:     0
report text length:     0
user number:            HDK52500
file name:              MYFILE
```

6.3.4 VALMON [1]

The Validation Monitor provides a mechanism for specifying the conditions under which a user may obtain access to DTSS and the resources available after access is granted. Access control is provided in order to prevent one user from masquerading as another. Resource control is provided because time-sharing systems operate by alloting a small portion of the total system to each user for a short period of time; any user who absorbs too much of the system in time, space, or I/O capacity may diminish the resources available to other users.

As noted in section 4.5.1, access control is provided by user-codes, answer-back drums and legal log-on times; resource control is provided on catalog storage, scratch storage, running time and special permissions. A design objective of the Validation Monitor is to simplify the task of providing access controls and resource limitations for a large number of users. A special effort is also made to minimise the amount of storage required for each user's validation parameters.

The validation information required for each user is called his validation record. In order to reduce both the amount of typing and the storage space required to hold validation records, a classification scheme is used which takes advantage of the existence of large subsets of the user population with similar requirments for storage space, running time, log-on times, permissions, and other resources.

Each user is assigned a six-bit code which becomes an integral part of his validation record. This code is completely independent of the group to which the user belongs and places him into one of sixty-four categories having similar permissions.

Since there are large subsets of users with similar requirements, it would be wasteful to duplicate a lot of information in each validation record. Accordingly, validation records have variable lengths, and any information not present in the validation record is obtained at sign-on time from a special data file called DEFAULT, catalogued in the Master User Directory. This file contains sixty-four possible user codes. Each entry contains information pertaining to system resource limitations and special permissions. Fig. 6.10 shows the format of a default file entry

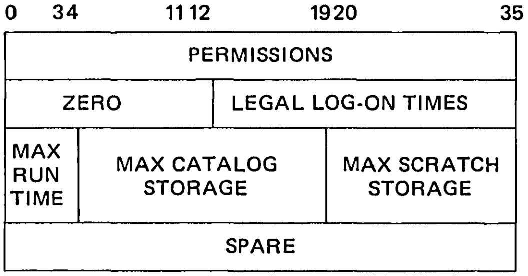


Fig. 6.10 – Default file entry.

The Validation Monitor contains all the mechanisms necessary for the generation of groups, user numbers, and default entries; for the specification of passwords, legal log-on times, and answerback drum sequences (either common to a group or pertaining to a particular user); and for the specification of system resource limitations (permissions, run-time limits, etc.).

The Validation Monitor may be accessed only from a control terminal. While commands are usually entered directly from one of the control terminals, the ability to accept input from a saved file has been added in order to facilitate the validation of large numbers of users at a time. This file can be created from a non-control terminal and then used as input to the Validation Monitor.

There are two categories of validation commands: major commands and minor commands. Major commands operate on groups, users, and default file entries; they permit adding, changing, deleting, and printing as appropriate. Minor commands modify a major command and specify additional information, such as the particular run-time limits, permissions, and catalog quotas which are to be assigned to a given user.

A group command causes the specified group to remain in effect for all subsequent commands until a new group command is given.

A user command causes subsequent minor commands to apply to the specified user number until either the end of the line is reached (in the case of terminal input), or a new major command is give:

The validation files produced by VALMON are described in section 6.2.1.

6.4 REFERENCES

[1] McGeachie, J. S. (March 1975), 'Validation Monitor User's Manual', DTSS Technical Memorandum 093.

[2] Roberts, S., and Noyce, W. (March 1978), 'SIMON', DTSS Technical Memorandum 060.

[3] Fernandez, A., and Ligett, S. (September 1977), 'DTSS Background Reference Manual', DTSS Technical Memorandum 008.

[4] Kiewit Comments (November-December 1977), pp. 5-6, Volume II, Number 2.

[5] Kozen, D. (June 1972), 'Background Job Descriptions', DTSS Technical Memorandum 064.

CHAPTER 7

Communications

As noted in section 2.1.5, the Terminal Communications Controller (TCC) is currently based upon a Honeywell 716 computer.

The 716 TCC Executive is a modular group of procedures, split broadly into three levels. The lowest level is concerned with controlling the transmission of data to and from the communications lines. This procedure is called UMLC, and since it is concerned with the control of the hardware line controllers (the UMLC and LSMLC), a description of these devices precedes a description of the software. The second level of software is composed of a number of procedures concerned with making the devices on the ends of the communication lines appear to have uniform characteristics so that the software at the third level may be device independent. The SWITCH procedure provides a great deal of flexibility for a variety of communication situations. A simplified diagram of the software structure is shown in Fig. 7.1.

7.1 THE 716 PROCESSOR AND LINE CONTROLLERS [1], [2]

The Honeywell 716 is a 16-bit machine. It has a range of programmable registers including two accumulators (the A and B registers), a stack pointer (the S register), an index register (the X register), and a program counter (the P register). A hardware stack and vectored interrupts are provided as is a real-time clock. The architecture of the 716 is shown in Fig. 7.2. Fig. 7.3 shows in more detail the data paths between peripherals and the machine registers and memory. Registers RI and RO are used by the Direct Memory Access (DMA) bus for transfers to and from memory. Peripherals connected only to the Programmed Input/Output (PIO) bus transfer to and from registers A and B via the arithmetic unit.

The memory control unit provides a priority schema for resolving conflicts in memory accesses. The priority structure is:

DMA break.
Real-time clock increment break.

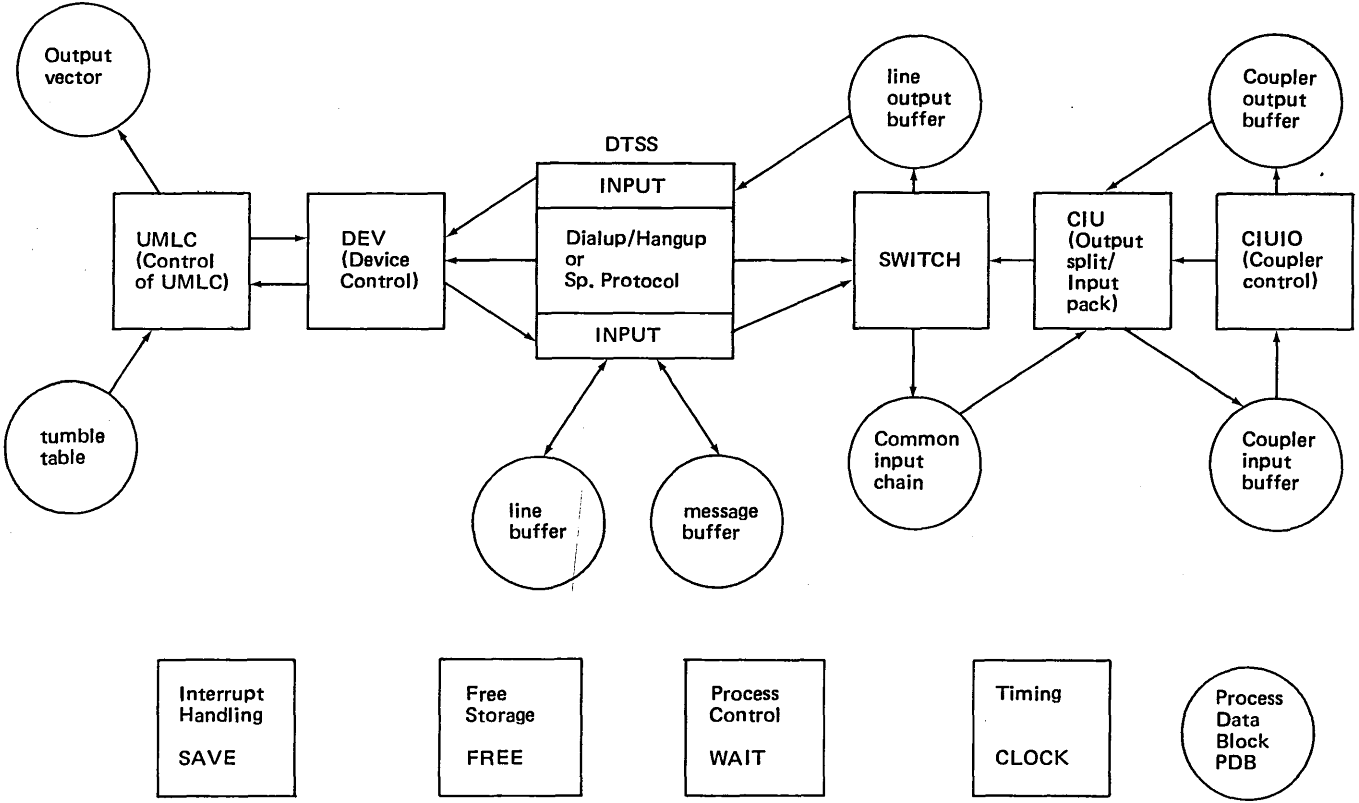


Fig. 7.1 – Simplified software structure.

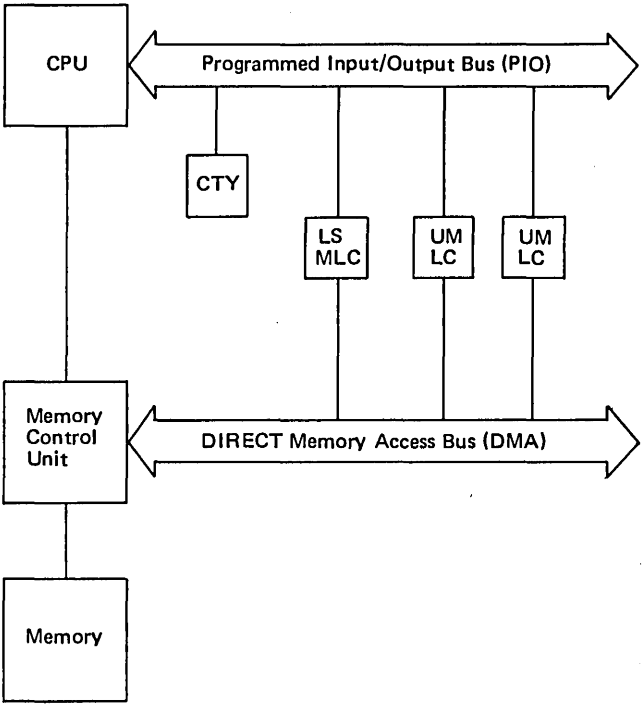

CPU
Programmed Input/Output Bus (PIO)
CTY
LS MLC
UM LC
UM LC
Memory Control Unit
DIRECT Memory Access Bus (DMA)
Memory

Fig. 7.2 – 716 architecture.

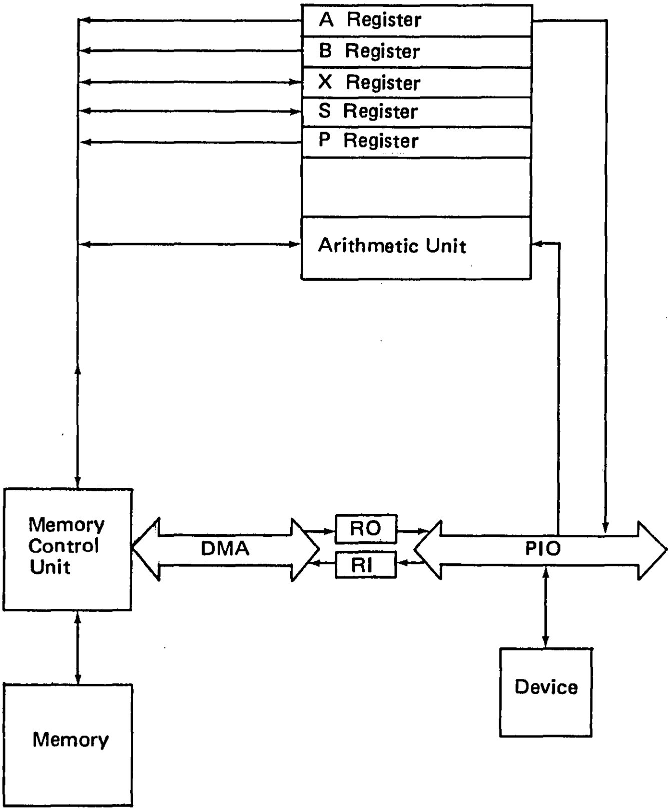


Fig. 7.3 – Device connections.

Power failure interrupt.
Watchdog timer interrupt.
Stack overflow interrupt.
Peripheral interrupt.
Instruction execution in the CPU.

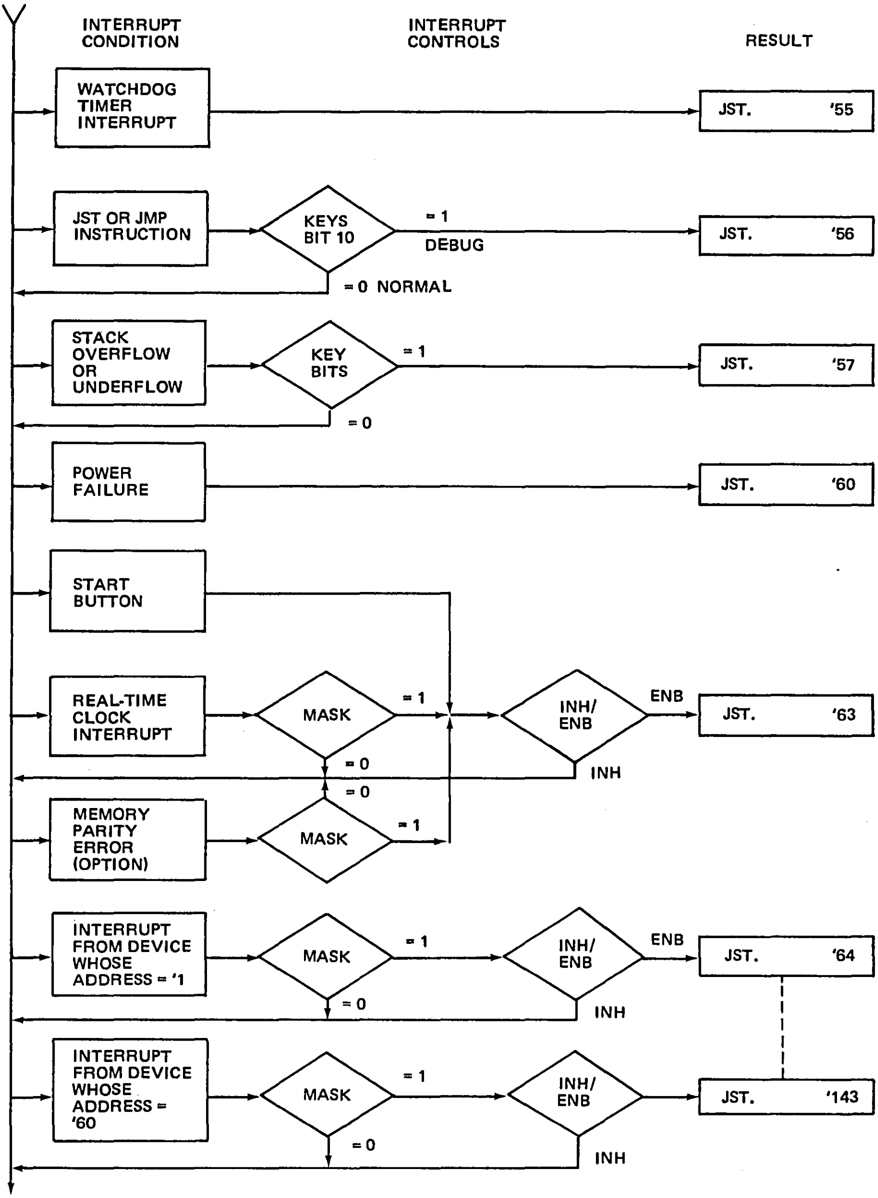
Those items called breaks do not change the contents of the P registers and are therefore invisible to the operating program, except that they may lengthen its execution time. Those items called interrupts change the contents of the P register by generating an indirect jump-store through one of a number of dedicated memory locations. These dedicated memory locations (the interrupt vector) must be set up with pointers to the relevant interrupt handling routines.

There are two general controls over most interrupts. The primary control is exercised by the inhibit interrupt and enable interrupt instructions. Interrupt requests are also controlled by mask bits. The setting of a mask allows an interrupt to reach the CPU, but an interrupt will only be generated if the CPU is in the ENABLE mode. Interrupts only occur between instructions, with the exception that power failure and watchdog timer interrupts may abort into instruction execution if too many multilevel indirect cycles are encountered. When an interrupt occurs the P register is saved automatically in the word pointed to by the location in the interrupt vector corresponding to the interrupt in question, and control is transferred to the location following that where the P register is stored. The interrupt routine must save other registers so as to protect the interrupted program's environment. On exit from the routine the registers must be restored. Fig. 7.4 shows the interrupt control mechanism.

The watchdog timer contains a line clock, a crystal clock and a watchdog timer. When the line clock is turned on it generates an interrupt request every cycle of the power line. The crystal clock can be set under program control to any period from 10 microseconds to over 40 milliseconds. It generates an indirect request at the set rate until turned off or reset with another rate. The watchdog timer, when turned on, must be reset within one second or it will cause an interrupt which has all the features of a power failure interrupt except that it occurs through a different location in the interrupt vector. The watchdog timer is used to ensure that the software does not hang.

The four main sources of interrupts on the 716 (when used as a TCC to DTSS) are the line controllers (when one runs out of tumble table space – see below), the coupler between the 716 and the mainframe, the crystal clock, and the control terminal.

The two line control modules available for the 716 computer, the Universal Multiline Controller (UMLC) and the Low Speed Multiline Controller (LSMLC) differ in that the LSMLC is for fixed speed asynchronous lines whilst the UMLC is for both synchronous and asynchronous line of varying speeds.



NOTE MASKS ALLOW INTERRUPT REQUESTS IN THE 1 STATE AND INHIBIT THEM IN THE 0 STATE. ALL INTERRUPTS PUT THE CPU IN THE EXTENDED ADDRESSING MODE AND SET INH/ENB TO INH PREVENTING ANOTHER CONTOLLED INTERRUPT UNTIL THE FIRST IS HANDLED.

Fig. 7.4 – Interrupt control.

Fig. 7.5 is a simplified block diagram of the UMLC. The UMLC is attached to the 716 computer via the PIO bus and the DMA bus. The UMLC controls the input and output of complete characters between the 716 memory and a maximum of 64 lines at rates determined by the UMLC internal clock and the speed of the lines. These lines may be either synchonous or asynchonous. In the UMLC memory, each line is assigned two memory locations for configuration and transmit and two memory locations for receive.

Communications between the UMLC and its line modules are performed serially. The UMLC scans through its memory, checking each line one at a time. During the line scan, the UMLC checks each line for a transmit ready signal at a rate of at least once every bit time. When a signal is detected, a data bit is transferred out of memory and to the line module if the transmit enable for the line is on. After the transmit operation is completed, the UMLC performs a similar operation during which it checks for a received data ready signal from a line module. If the receive enable bit for the line is on, the received data bit is written into memory when the data ready signal is true.

Because of the wide range of synchronous line speeds, a differential line scan is used in the UMLC. The high-speed syncronous lines are checked four times during each complete memory scan, and the medium-speed lines are checked twice. All other lines are checked only once during a line scan.

The UMLC performs a second type of scan, a character scan, which is independent of the line scan. During the character scan, the UMLC checks each line one at a time to see whether it is ready for the next transmit character or whether an assembled receive character is ready to be transferred to the 716 memory. When a servicing requirement is detected, the UMLC generates the necessary request as explained below and the character scan continues until the next ready condition is detected. At that time, the character scan is halted until the first request is serviced. Then the character scan is continued.

Associated with each line is a dedicated location in an area of 716 memory called the output vector. The base address of the output vector must be loaded into the Transmit Address Register (TAR) in the UMLC. The dedicated location in the output vector is found by adding the UMLC line number to the contents of the TAR. Thus when a new transmit character is required, the line number is stored in the character buffer register (CBR) and then added to the contents of the TAR. The resultant address, along with a DMA output request is put on the DMA bus. The character required is transferred out of the output vector in the 716 memory to the UMLC, where it is placed in the CBR along with the line number. The character is then written into the appropriate memory location within the UMLC for the line.

During the transfer of an assembled receive character, the character and the associated line number are transferred into the CBR when the CBR is available. Then the UMLC puts the address of the location in the 716 memory where the

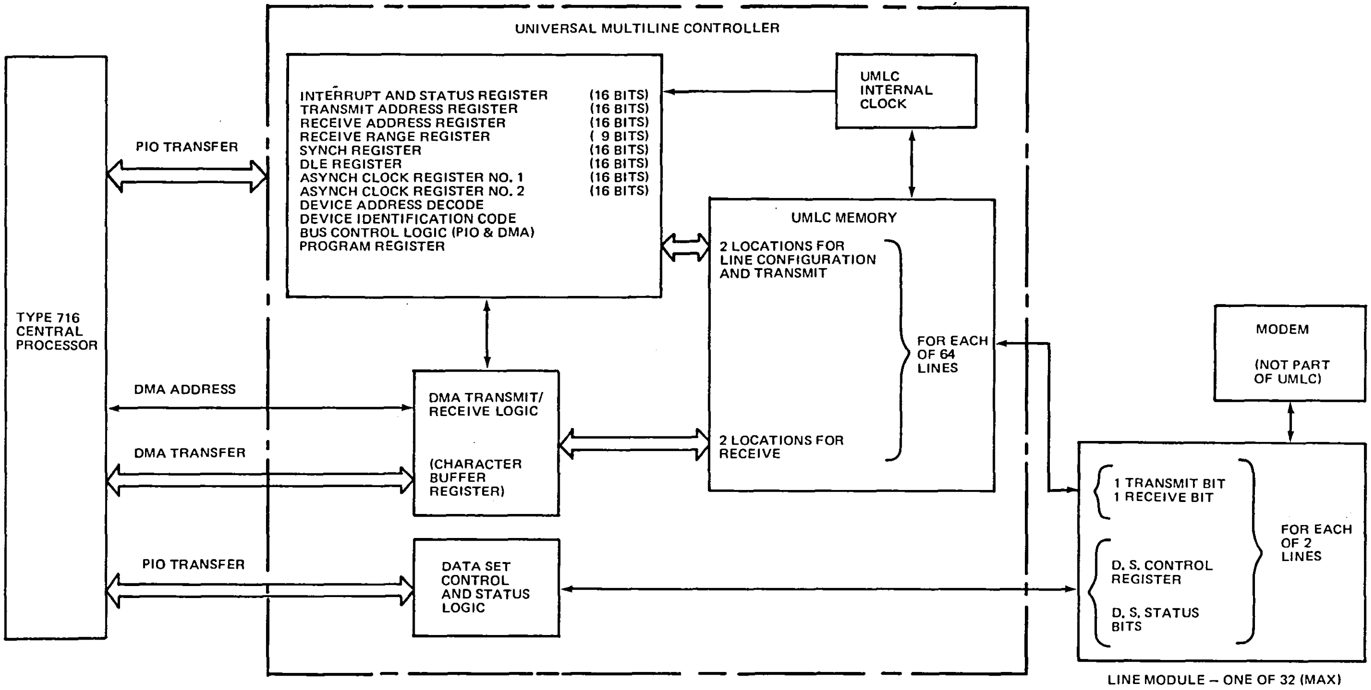

UNIVERSAL MULTILINE CONTROLLER
TYPE 716 CENTRAL PROCESSOR
PIO TRANSFER
INTERRUPT AND STATUS REGISTER (16 BITS)
TRANSMIT ADDRESS REGISTER (16 BITS)
RECEIVE ADDRESS REGISTER (16 BITS)
RECEIVE RANGE REGISTER (9 BITS)
SYNCH REGISTER (16 BITS)
DLE REGISTER (16 BITS)
ASYNCH CLOCK REGISTER NO. 1 (16 BITS)
ASYNCH CLOCK REGISTER NO. 2 (16 BITS)
DEVICE ADDRESS DECODE
DEVICE IDENTIFICATION CODE
BUS CONTROL LOGIC (PIO & DMA)
PROGRAM REGISTER
UMLC INTERNAL CLOCK
UMLC MEMORY
2 LOCATIONS FOR LINE CONFIGURATION AND TRANSMIT
FOR EACH OF 64 LINES
2 LOCATIONS FOR RECEIVE
DMA ADDRESS
DMA TRANSFER
DMA TRANSMIT/ RECEIVE LOGIC
(CHARACTER BUFFER REGISTER)
PIO TRANSFER
DATA SET CONTROL AND STATUS LOGIC
MODEM
(NOT PART OF UMLC)
1 TRANSMIT BIT
1 RECEIVE BIT
D. S. CONTROL REGISTER
D. S. STATUS BITS
FOR EACH OF 2 LINES
LINE MODULE – ONE OF 32 (MAX)

Fig. 7.5 – Block diagram of UMLC.

character is to be stored on the DMA bus, thus transferring the character and its associated line number to the 716 memory in an area called the tumble table.

The UMLC control procedure administers an area of memory called the tumble table into which the UMLC device deposits characters and from which the UMLC control procedure extracts characters. The 716 Executive conditions the UMLC for operation via PIO instructions. The Receive Address Register (RAR) must be loaded with the starting address of a tumble table. Receive characters along with their associated line numbers are stored in this tumble table as they mature, and RAR is incremented after each receive transfer. The Receive Range Register (RRGR) must contain the last available address of the tumble table. The UMLC compares RAR with RRGR to determine when the tumble table is full. The UMLC control procedure makes only part (typically half) of the tumble table memory available to the UMLC device. When the comparison between RAR and RRGR indicates that the portion of the tumble table available to the UMLC is full, a receive end-of-range interrupt is generated and the UMLC control procedure makes the other half of the tumble table available.

The UMLC control procedure extracts charcters and their associated line number from the tumble table concurrently with the UMLC filling it by maintaining a pointer itself which it compares with the value in the RAR to ensure that only active areas of the tumble table are accessed.

Line configuration information is output via PIO instructions by the 716 Executive. The configuration information is written to the appropriate UMLC memory locations.

7.2 UMLC CONTROL SOFTWARE

When a line becomes active, such as when a user dials into the system, a process data block (PDB) (see Fig 7.6) is allocated in which information regarding the characteristics of the line are recorded along with other information. On noting the new line, the procedure DTSS is activated and this procedure determines the characteristics of line. This is done by checking the configuration file, and in some cases by also asking the device control processes for information about the line. IBM 2741 compatible terminals are identified by the transmission of a special character as the connection is made and are routed to the 2741 device control process. All other asynchronous terminals are routed through a variant of ASYNC (depending only on the line speed). Synchronous lines are routed to BLU (the Telenet line level protocol handler – see below and Fig. 7.8) which attempts to run the line with the BLU protocol. If the device at the end of the line does not respond appropriately, the BLU procedure assumes that it will be handled by BISYNC and passes it to it. (A description of the software for handling synchronous lines is given in section 7.3.) Having determined the line type, DTSS tells TCFACE what sort of terminal it is and which monitor it

- stack
- stack pointer
- pointer to next PDB
- TCFACE number for this line
- pointer to UMLC output vector for this line
- pointer to device control process
- pointer to current location in input process
- pointer to current location in output process
- thread of PDBs outputting at this speed
- speed index for the line
- modem and line status and control flags
- mode settings for this line (from TCFACE)
- pointer to line buffer
- pointer to message buffer
- head of output chain
- general purpose work space
- variable length extension for synchronous lines

Fig. 7.6 – Outline of a process data block.

should be passed to (for example, SIMON for a user terminal and BAKMON for a remote printer). The DTSS procedure sets up the pointer to the relevant device control routine (ASYNC, BLU, etc.) in the PDB and calls the non-interrupt code for that device control routine. When that routine is run, it calls the UMLC control routine which puts the pointers to the input and output routines in the PDB and goes to sleep.

The input segment of the UMLC control procedure is called by the CLOCK procedure one hundred times per second. The input routine extracts character/line number pairs from the tumble table and calls the corresponding line's 'get next character' routine which passes the character to the device control procedure for the line.

The output segment of UMLC is driven by the CLOCK process. All lines of a given speed doing output are held on a chain linked through their PDBs. Each entry on the chain uses the line number to index into the output vector. If the entry in the vector for that line is zero, signifying that the line is ready for a character, UMLC deposits a character in the appropriate location in the output vector and requests another character from the device control process for this line.

7.3 DEVICE CONTROL SOFTWARE [3], [4], [5]

One objective of the various device control procedures is to hide the device dependent features of the various terminals.

There are currently five procedures at his level, although one has two variants. The functions of the various procedures are as follows:

CTY This procedure controls the control terminal which is interfaced directly to the Programmed Input/Output bus rather than through an UMLC or LSMLC. The control terminal is used for monitoring and debugging functions.

2741 This procedure is for dealing with IBM 2741 compatible terminals and their associated protocol.

ASYNC This is the general device dependent control procedure for asynchronous terminals. There is a special version for handling 110 baud devices. Terminals running at 150, 300 or 1200 baud use the general ASYNC procedure. This procedure attempts to support most ASCII terminals with minimum delays. A SIMON command may be used to condition this process to generate longer delays on the carriage control for those terminals which require such delays, another SIMON command may be used to speed up the handling of terminals where the carriage control functions are performed at the same rate as other characters.

BLU There are two levels of protocol used for communication between the 716 and Telenet: BLU level and TIU level. BLU stands for Basic Link Unit and TIU stands for Telenet Information Unit.

Blocks of data sent and received at the BLU level are called BLUs. BLUs contain TIUs which are the messages sent and received at the TIU level. Thus a BLU may be thought of as a package for a TIU. The most important difference between BLUs and TIUs is that there is a Virtual Connection Address associated with each TIU. This address associates the TIU with one of the many possible virtual connections which are multiplexed through a single physical connection between a host and Telenet. BLUs have no associated virtual connection address. The purpose of the BLU level of the protocol is to ensure that TIUs are received without errors.

The BLU process implements the Telenet BLU protocol. This protocol is also used to run the various remote printers and remote TCCs.

(Both levels of the Telenet host interface protocol use the same basic error recovery techniques as are used in the IBM Synchronous Data Link Control protocol. It is for this reason that the BLU level of the protocol have come to be called 'SDLC' on DTSS. When the term 'SDLC' is used in DTSS documentation it almost always refers to the BLU level of the Telenet Host Interface.)

BISYNC The BISYNC process implements the IBM 2780 Binary Synchronous protocol. This protocol is used both for connecting to 2780 Remote Job Entry stations or compatible devices and for connecting to an IBM system which supports 2780 RJE stations.

7.4 CHARACTER INPUT/OUTPUT

When the UMLC procedure gets a character from the tumble table for a non-Telenet, non-remote TCC line, it passes it on to the appropriate device control procedure, which in turn passes it on to the INPUT procedure. INPUT puts the character into a buffer for that line. The head of the line buffer is maintained in the PDB for that line as shown in Fig. 7.7.

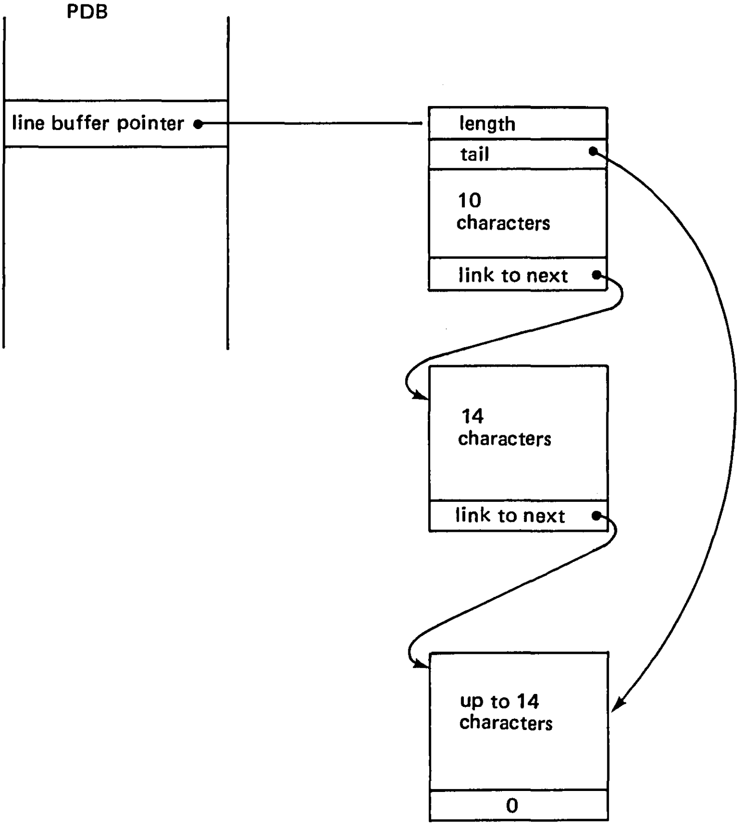


Fig. 7.7 – Line buffer organisation.

The INPUT procedure checks to see if the character is in any way special. In particular, if the character is control-X, control-R, delete, or carriage-return. Delete, is taken to mean 'delete the previous character on this line' and control-X is taken to mean 'delete the current line'. Thus INPUT uses these characters to edit the line buffer. Control-R is taken to mean 'echo the current line'. When INPUT receives a carriage-return, it copies the line buffer into the message buffer for the line. When the message buffer is full, or when the mode setting on the line dictates (see sention 6.1.3), INPUT passes the data into the Common

Input Chain for eventual transmission to TCFACE. The storage required by INPUT for its buffers and that required by all other processes is controlled by the process FREE, which maintains a bit map for each 8-word block of the entire 716 memory.

When UMLC requests another character for output from the device control procedure, the device control procedure gets the character from a message string given it by the OUTPUT routine. When the message string is exhausted, OUTPUT get the next from the output buffer for the line. The pointer to the next character in the output buffer is held in the PDB for the line.

The CIU process maintains the CIU (Computer Interface Unit or coupler) input and output buffers. As noted above, input from all lines is routed to the Common Input Chain. The CIU process takes information from the Common Input Chain, puts it in the format required by TCFACE, and then packs it in the CIU input buffer. This action is carried out whenever the CIUIO process requests input to send to TCFACE. The CIUIO process controls the CIU hardware, much as the UMLC procedure controls the UMLC.

Similarly, when the CIUIO process gets data from TCFACE, it places it in the CIU output buffer and tells the CIU process to dispose of it. CIU unpacks the data and distributes it to the various line output buffers, according to the line numbers associated with each message.

7.5 SYNCHRONOUS PROTOCOLS [6]

One of the procedures in the 716 provides the Telenet Information Unit protocol. This is the module TELENET in Fig. 7.8, which implements the TIU prototcol by performing much the same functions as the INPUT and OUTPUT processes described earlier, and by multiplexing and demultiplexing the data streams within the TIUs.

The BLU prototcol is used not only for Telenet itself but also for remote printers (controlled by Nova computers) and Remote TCCs. The remote printers are routed through the BLU process. The remote TCCs are routed through the DIRECT process, so named since it implements the same protocol as the 716-TCFACE protocol and can thus be transmitted through the 716 software directly without change.

The Binary Synchronous protocol is used to run the Datalink system. This is the BISYNC module in Fig. 7.9. The Datalink system allows users to access IBM 360 or 370 systems in a manner similar to that used for 2780 RJE stations.

7.6 THE SWITCH PROCEDURE

The SWITCH procedure is central to the flow of both control and data within the 716 Executive. By making appropriate calls to the SWITCH procedure, data may be routed along a variety of paths. Thus, for example, it is possible to use

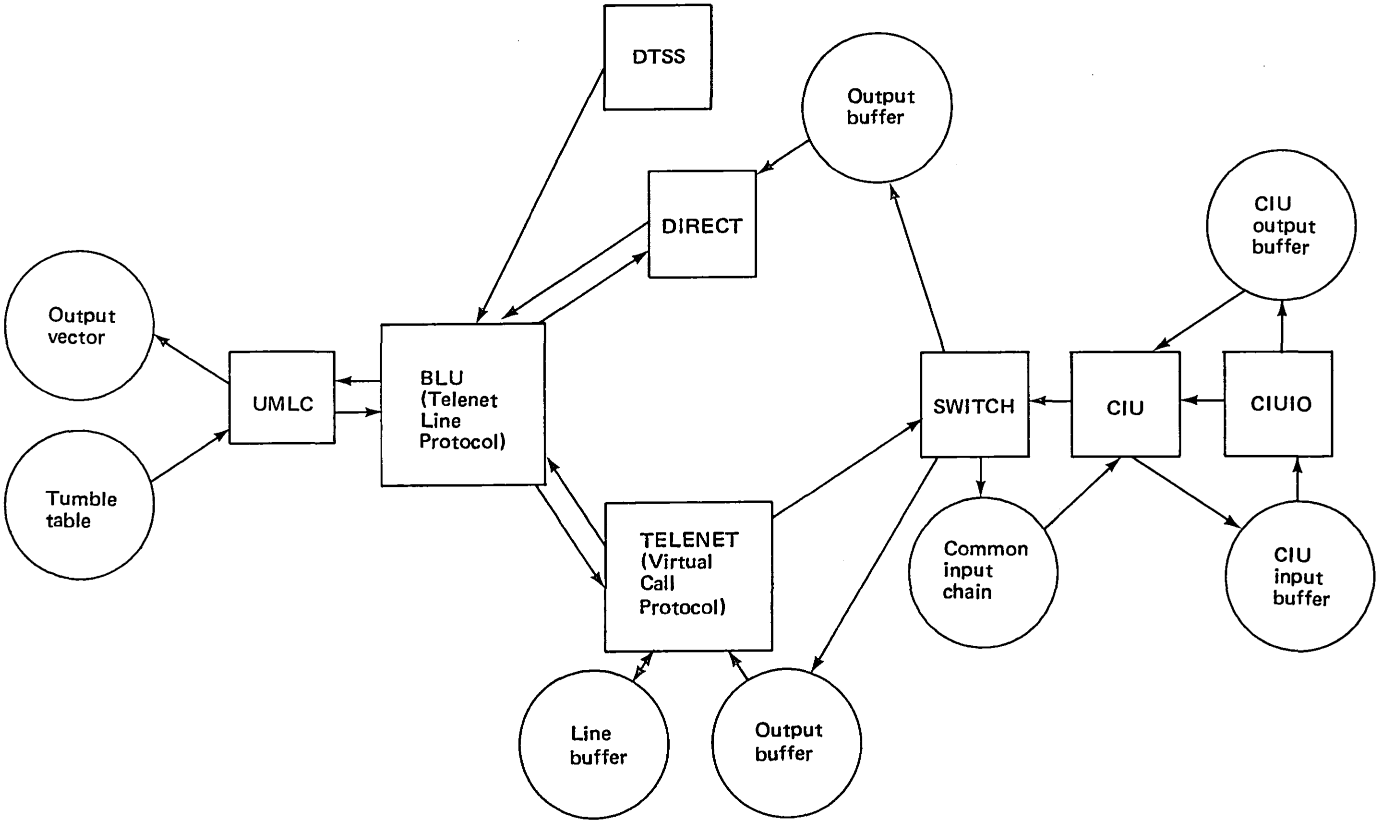


Fig. 7.8 – Telenet protocol routines.

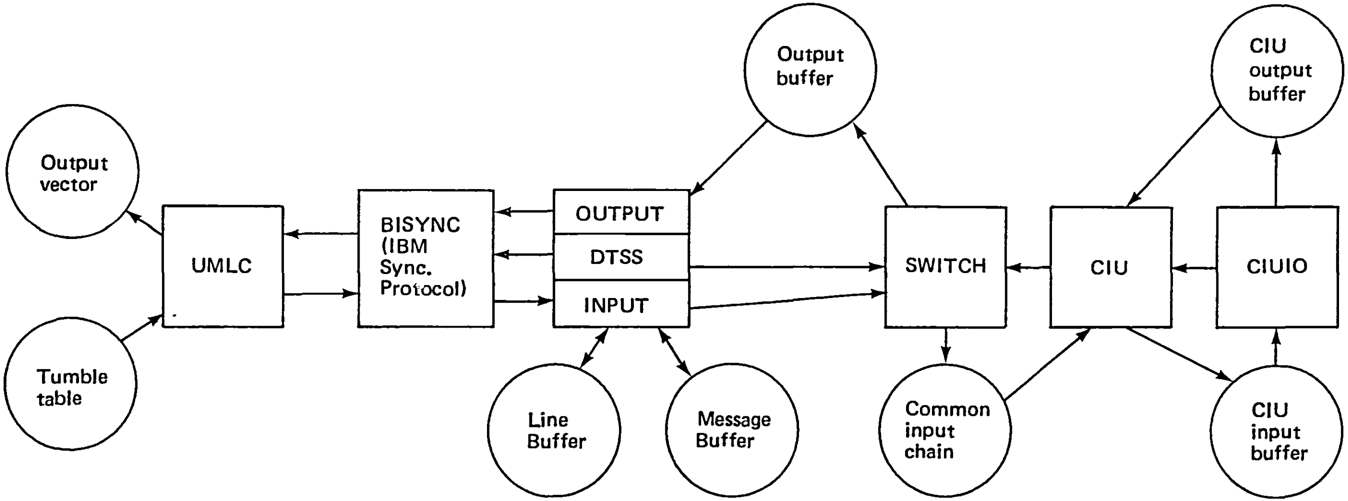


Fig. 7.9 – Binary synchronous protocol routines.

the control terminal on a local TCC to monitor a remote TCC by setting up the following path (see Fig. 7.1):

> Control terminal – CTY – INPUT – SWITCH – output buffer – DIRECT – BLU – UMLC – output vector – communications line to remote TCC – tumble table – UMLC – BLU – DIRECT – SWITCH – CTL – SWITCH – output buffer – DIRECT – BLU – UMLC – output vector – communications line back to the local TCC – tumble table – UMLC – BLU – DIRECT – SWITCH – output buffer – OUTPUT – CTY – Control terminal.

At present it is not possible to go out to another Telenet host from a terminal on DTSS. However, this capability is being added to the TELENET procedure and then a connection such as the following will be possible:

> terminal – tumble table – UMLC – ASYNC – INPUT – line buffer – message buffer – SWITCH – output buffer – TELENET – BLU – UMLC – output vector – Telenet to remote host – (operations in host) – return data through Telenet – tumble table – UMLC – BLU – TELENET – message buffer – SWITCH – output buffer – OUTPUT – ASYNC – UMLC – output vector – terminal.

7.7 PROCESSES IN THE 716

In the sections on the software in the 6000 mainframe, the terms ‘slave job’, ‘job’, and ‘module’ were used to describe various procedures and processes within the system. In this chapter the term ‘procedure’ has been used to discuss the functions of the various items of software which constitute the overall communications software in the 716. It is appropriate to consider at this stage the processes within the 716 (process being defined as the execution of a program with a given set of data).

7.7.1 CIU

There is a single process which maintains the computer interface unit. All data to and from the computer interface unit is handled by this single process.

7.7.2 Input Segment of UMLC

There is an instance of this process for each physical line controller (UMLC or LSMLC).

7.7.3 Output Segment of UMLC

There is an instance of this process for each line speed.

7.7.4 DTSS

There is one instance of this process for each user (other than those users entering via Telenet).

7.7.5 Telenet

There is one instance of this process for each active Telenet line into the system.

7.8 LOADING THE 716 SOFTWARE [7], [8]

The 716 software performs a variety of consistency checks as part of its operation. If one of these fails the 716 sends a reload request message (102) to TCFACE. On receipt of the message, TCFACE runs the TCL monitor via Login. If the 716 fails to respond to TCFACE at any point (for example, when the system is started at the beginning of the day and the 716 is not running for some reason), TCFACE initiates a reload sequence.

The TCL monitor issues a FORCED READ (one of the commands to the CIU) to get the first 512 words of the 716 memory. Following this the 716 enters its bootstrap program and then waits in an idle loop for something to happen. TCL then issues a FORCED WRITE to load the load/dump program (CLOAD) into the 716. This causes the 716 to re-enter the bootstrap and exit to CLOAD.

In the meantime the TCL monitor reads the 716 machine conditions and the setting of sense switch 2, and then procedes to read the remainder of the 716 memory. Depending upon the setting of sense switch 2, TCL now loads either the current TCC operating software or the previous version of the software. The loading is check summed by both CLOAD in the 716 and by the TCL monitor.

The 716 is set running and it checks the setting of sense switch 3, transferring either to the 716 Executive or to the debugger.

The TCL monitor files the TCC memory dump file in DUMPCAT. Every fifteen minutes, BAKMON checks to see if there are any files in DUMPCAT, and if so runs the dump formatter to produce a formatted dump on the printer. TCFACE waits for a special interrupt from the TCC to signal that communication can continue.

Fig. 7.10 illustrates the above sequence of events.

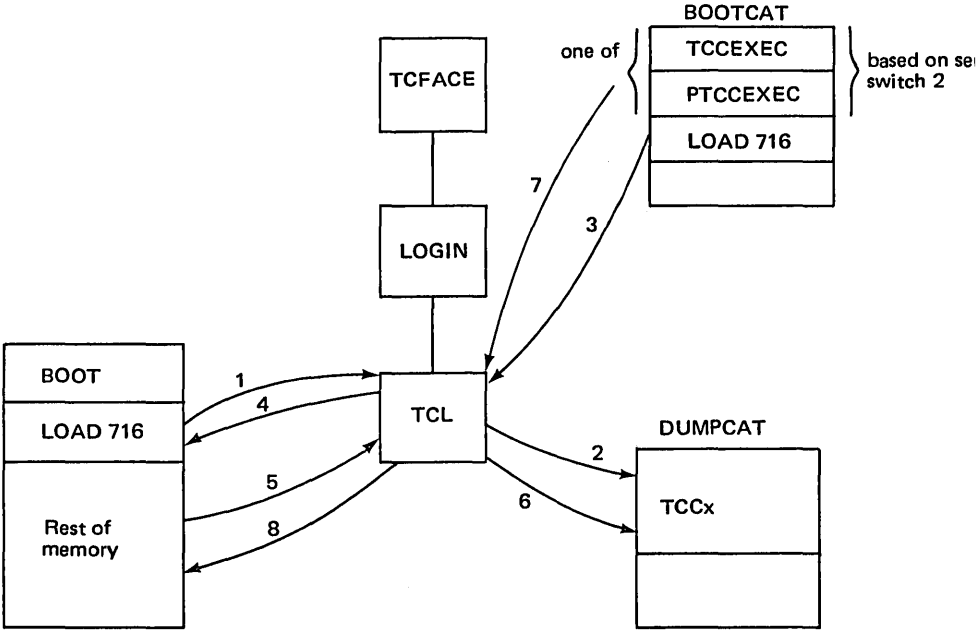


Fig. 7.10 – 716 reload sequence.

7.9 REFERENCES

[1] 'Programmers Reference Manual, System 700' (January 1972), Honeywell Information Systems, document 70130072575A.

[2] 'Honeywell Type 6322 Universal Multiline Controller Programming Manual' (April 1972), Honeywell Information Systems, document 70130072689A.

[3] 'Host Interface Specification' (1975), Telenet Communications Corp.

[4] 'Binary Synchronous Communications' (February 1970), IBM Corp., document GA273004.

[5] Pedersen, L. C. (November 1977), 'Communications', DTSS Inc. Systems Programmer's Training Course notes.

[6] Green, R. L., and Pedersen, L. C. (January 1977), 'DTSS Datalink System', DTSS System Memorandum 291.

[7] Dunten, S. D., and Hargraves, R. F. (June 1976), '716 Hardware', DTSS System Memorandum 278.

[8] '716 to 635/6000 Coupler Special Option Manual, Volume 1' (February 1975), Honeywell Information Systems, document 70130072765A.

CHAPTER 8

Operations

The object of this chapter is to elaborate on the interface to the operating system seen only by operators and systems programmers. Since both these groups of people are ultimately responsible for the efficient, secure running of a system and to some extent for the reliability of the system, this aspect of operating system design is very important and yet neglected by almost every text on operating systems.

8.1 SYSTEM STARTING

A system start-up is termed a 'boot' or 'bootload' since the operator at some point presses the button labelled 'BOOTLOAD' on the console. There are a variety of boots which can be performed depending on the situation. There are a number of reasons for booting the system.

8.1.1 Reasons for booting the system

The system must be booted at initial installation or on installing a new release of the system. This situation is described in section 8.3. After a field engineer has had the machine to perform some form of maintenance or after some other operating system (such as GECOS) has been in use, a boot must be performed. The system must be rebooted after a configuration change (such as a device going down or coming back on-line). The system is not designed to run continuously for more than a few days: a time counter overflows in six days. The normal mode of operation is to shut the system down daily and follow this sometime later with a boot. Finally, if the fragmentation of storage on discs becomes excessive, it is necessary to run a master-mode purge and following this a boot is required.

8.1.2 Disc boot

The normal procedure used to start up DTSS following a normal shutdown (see section 8.2) is a one-card boot. It is not possible to boot directly from a disc, only a card reader or tape drive may be used as a boot device. During tape start-up, the Executive creates and punches a single card which may be booted. This card, when booted (that is, when the BOOTLOAD button on the operator's console has been pressed), will read a disc boot program from a portion of the reserved area of the disc. The disc boot program reads the Executive into memory from the reserved area and transfers to the beginning of a large piece of it called the disc start-up (DSTART). The functions performed by DSTART are as follows:

1. Checks checksums on material just read in from reserved area by disc boot program.
2. Checks control CPU fault vector address.
3. Determines memory size and initializes information used to allocate space (for example, for swapper).
4. Checks to be sure all processors have a consistent view of memory.
5. Initialises interrupt vector and various tables for physical I/O.
6. Reads in core catalogs, most importantly the MFD, making sure that checksums match.
7. Reads page tables and checks checksums in case system was shutdown previously without doing a protect. Uses bad checksum as flag that page table may be invalid. May result in CHECKSUM ERROR messages. These messages may also result if packs were mounted on the wrong drives. The operator is asked to verify correct pack locations. If previous shutdown was unclean, a PURGE will be necessary to make new page tables.
8. Resets page tables for devices with preferences lower than 3 to indicate that all storage is free. Pref 0 is used for spare packs, Pref 1 is used for swapping, and Pref 2 is used for high-speed scratch.
9. Initialises various variables and starts various continuous tasks. For example, the timed task which displays statistics.
10. Deletes file with preferences less than 3 from core catalogs (they may be catalogued only in core catalogs). This is because the associated storage was freed.
11. Catalog entries for various special catalogued files are created. Most importantly, entries are made in :PERCAT for all non-allocatable devices, for example, printer, tapes, 716's.
12. At this point Disc Start-Up transfers control to a lower section of memory and reads Slave Loader from the disc into the area of memory just occupied by the previous portion of Disc Start-Up.
13. The job table is then initialised to make it look as if Slave Loader is already running. A flag is set, however, which prevents jobs from actually running.
14. 'EXEC UP' is displayed on the console.

15. At this point, the Executive simply waits for something to happen, since it tells the operator to enter the data and time. When the operator enters the date and time, the global flag is reset and Slave Loader is allowed to 'continue' running.
16. Disc Start-Up is also entered after a 'crash'. When entrance is from Crash/Restart, a flag is set such that Disc Start-Up will skip certain functions, such as reading in core catalogs from the disc. Also, SYSTEM CRASH is displayed instead of EXEC UP and the operator need not enter the date and time.

8.1.3 Tape boots

There are three types of tape boots – a cold boot, a warm boot with patches, and a warm boot.

The cold boot procedure, and the associated full logical load, is used when it becomes necessary to reload the entire DTSS file structure, including the system software modules. A cold boot is necessary following a system crash which destroys critical data on the disc, when a new system is being created, and may also be run from time to time to obtain more efficient allocation of system storage. The cold boot procedure is described in section 8.3.

A warm boot is a subset of a warm boot with patches and is primarily used as a substitute for a disc boot if for some reason the card-reader is not available.

A warm boot with patches is used whenever the configuration is changed or if the Executive needs to be patched. The patches and configuration changes can be entered from the card reader or the console. Two tapes are used in the operation – a warm boot tape and a scratch tape (which becomes a load tape). The IOMTBOOT program (initiated by the BOOTLOAD button) copies the warm boot tape to the load tape applying the patches and taking account of the ENV deck information. The load tape is the one used to do a warm boot without the use of the card reader. After the creation of the load tape, the Executive and Slave Loader are read from it into memory and control is transferred to the Executive Tape Start-Up (TSTART), which performs the following functions:

Configuration Checks:

1. Checks IOM.
   Ensures that system was booted from IOM Zero (IOM Number Switches).
   Checks Mailbox area Location (IOM Base Switches).
2. Checks IMW Area Location (Interrupt Base Switches).
   Checks Fault Vector Address on Control CPU.
3. Polls all Memory Ports.
   Makes sure that unused ports are masked off.
   Makes sure there is only one IOM.
   Counts Non-Control Processors, making sure there are not too many.

4. Computes Memory Size.
5. Adjusts for Extended Memory and other optional features.

Reads and processes Environment (ENV) deck (see below).
Data Control Word (DCW) Lists are set up for Crash/Restart.
Reserved Area on (secondary) system device is written to contain:

1. The Executive itself.
2. Disc Boot Routine to read in Executive.
3. Slave Loader.
4. DCW Lists for Crash/Restart.
5. :ENVINFO (a copy of the environment file).

A Disc Boot Card is punched.
A disc boot is simulated.

8.1.4 Environment file

As noted above, memory size and system controller configurations are determined at boot time. The configuration of the IOM is however specified by an ENV file. Since the ENV file historically was held as a card file, the literature talks of ENV decks and ENV cards rather than ENV files and ENV records. A typical ENV deck is shown below:

```
ENV INSTALLATION XYZ
ENV CONFIGURATION HIS 66/20,192K MOS MEM
ENV CTY     020722,020723
ENV PERCAT TYPE=CONSOLE,PREFIX=/1
ENV PERCAT TYPE=READER,PREFIX=/3
ENV PERCAT TYPE=PRINTER,PREFIX=/120063
ENV PERCAT TYPE=MPC,PREFIX=MP00,ACCESS=457777
ENV PERCAT TYPE=TAPE9,PREFIX=/2
ENV PERCAT TYPE H716,PREFIX=/6
ENV PERCAT TYPE MSU451, ACCESS=457777,MAP=MSU451
ENV PERCAT TYPE MSU451,ACCESS=457777
ENV PERCAT TYPE=MSU451
ENV DEV01  TYPE=CONSOLE,PDA=0037,UNIT=00
ENV DEV04  TYPE=H716,PDA=0122,UNIT=02
ENV DEV06  TYPE=READER,PDA=0131, UNIT=00
ENV DEV07  TYPE=PRINTER,PDA=0130,UNIT=00
ENV DEV10  TYPE=TAPE9,PDA=0120,CODE=4,UNIT=01
ENV DEV11  TYPE=TAPE9,PDA=0220,CODE=4,UNIT=02
ENV XBAR01 10 14 11 15 12 16 13 17
ENV DEV30  TYPE=MSU451F,PDA=1X01,PREF=6
ENV DEV31  TYPE=MSU451F,PDA=02X01,PREF=6
ENV DEV32  TYPE=MSU451F,PDA=03X01,PREF=6
ENV DEV33  TYPE=MSU451,PDA=04X01
ENV DEV40  TYPE=MPC,PDA=0020,FIRM=MTP601,UNIT=20
ENV DEV41  TYPE=MPC,PDA=0010,FIRM=MSU451,UNIT=10
ENV DEV44   TYPE=MPC,PDA=0030,FIRM=URC002,PORTS=U203,UCRP,UNIT=30
ENV DEV70   TYPE=MSU451C,PDA=770030,PREF=4,SUFFIX=C0
ENV PHYSDEV  TYPE=MSU451,RANGE=16646
ENV SYSDEV    SECOND=30
ENV SCHEDULE SWEEP=0
ENV SCHEDULE MAXFORCE=1,MAXSWAPS=2
ENV SCHEDULE DECAY=375341217200,NCQRATIO=200,CRUNCHER=41
ENV CHARGES  CPU=200000,CORE=600000,I/O=310000,IOTIM=0
ENV SCHEDULE CPU=20000,CORE=400000,I/O=60000
***EOF
```

The CTY card indicates the physical lines of the control terminals. The PERCAT cards are used to generate entries in PERCAT, the peripherals catalog. The name of each peripheral as it appears in PERCAT is a compound of the PREFIX, UNIT and SUFFIX fields. The ACCESS field is used to generate the ACCESS field in PERCAT. The DEV cards describe the type, location on the IOM and other aspects of the devices. The PDA field (peripheral device address) is of the form xxyy where xx is the device number on the channel and yy is the channel number on the IOM, or is of the form 7700dd where dd is a pointer to device card DEVdd, or is of the form xxXbb where xx is the device number and bb is a pointer to cross bar card XBARbb. Thus, DEV10 is a 9-track tape on channel 20 on the IOM and is device 1 on that channel. DEV41 is a controller, is device 0 on channel 10, its firmware is in the file OPCAT:MSU451 (MSU451 is the Honeywell product number for a moving head disc pack system). DEV30-DEV33 are four MSU451 disc drives. DTSS regards disc drives as entire packs (MSU451 as in DEV33), as file storage (MSU451F as in DEV30), or as center tracks (MSU451C as in DEV70). The PHYSDEV card defines the range regarded as center tracks (forty tracks starting at the octal address specified). The PREF field indicates the preference class (see section 4.3). Thus DEV70 is the catalog (or center) tracks of all the MSU451's starting at DEV30 (indicated by PDA 770030) and going in sequence (that is, DEV30, DEV31, DEV32, DEV33) and that it will be used for preference 0-4 files.

The XBAR card specifies logical channel numbers. The MPC provides a number of logical channels over a single physical channel. The configuration described in the above ENV deck is a dual channel system as shown in Fig. 8.1. In this, channel 10 is addressable as 4 logical channels 10-13, and channel 14 as 14-17. Thus DEV30 has a number of legal peripheral device addresses, namely 0110, 0111, 0112, . . . , 0117, that is, device 1 on any of channels 10-17.

The SYSDEV card indicates where the Executive will write the reserved area (see section 8.1.3 above – TSTART), in this case on device 30 (DEV30). The SCHEDULE cards are used by the installation to tune the scheduler. The CHARGES card specifies the ratios for the changing algorithm.

If tape drive number 2 became inoperable, the required changes to the ENV deck would simply be to delete card DEV11. If the MPC which provides

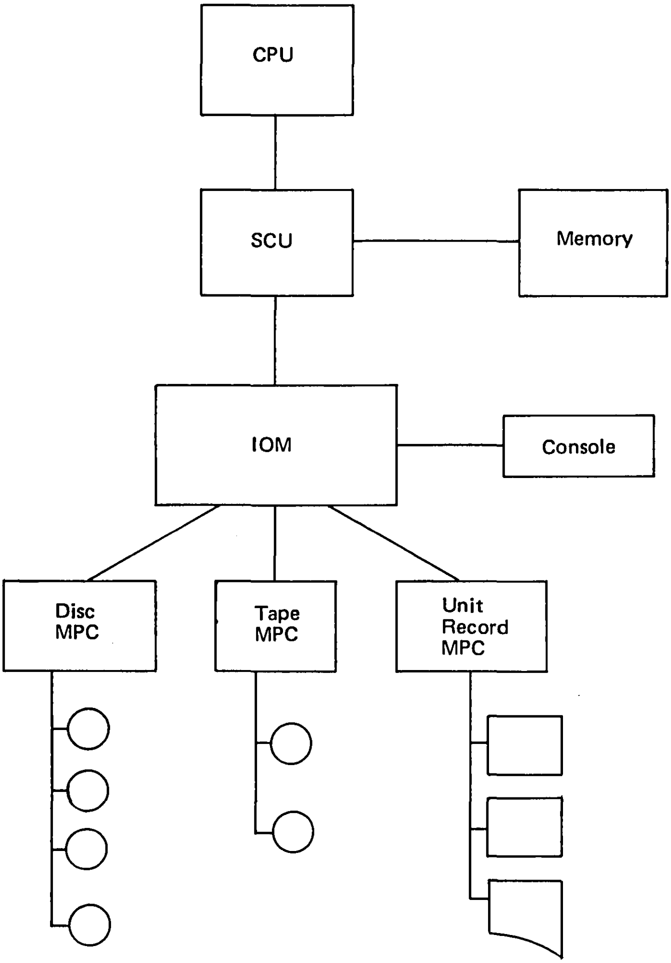


Fig. 8.1 – ENV deck configuration.

channel 10 and its associated logical channels became inoperable, the changes required would be to substitute a new XBAR card of the form:

```
ENV XBAR01      14 15 16 17
```

If at some point, four more MSU451 drives are added, and that the file tracks are to be preference 5 and the catalog tracks are to be preference 2, with the fourth drive a second spare. The changes to the ENV deck would be the addition of the following cards:

```
ENV DEV34  TYPE=MSU451F,PDA=05X01,PREF=5
ENV DEV35  TYPE=MSU451F,PDA=06X01,PREF=5
ENV DEV36  TYPE=MSU451F,PDA=07X01,PREF=5
ENV DEV37  TYPE=MSU451,PDA=10X01
ENV DEV71  TYPE=MSU451C,PDA=770034,PREF=2,SUFFIX=C1
```

8.2 SYSTEM SHUTDOWN

A typical sequence of events for system shutdown is as follows. The message and warnings system is run and users are informed that the system is going down at some time in the future. At the appropriate time, the operator types DEDICATE to OPMON, to restrict use of the system to users who are under OPMON, and the following sequence of events:

1. Operations Monitor writes a DED command to Login.
2. Operations Monitor writes a DIS ALL command to TCFACE.
3. Operations Monitor writes an ENA command to TCFACE specifying the two current control lines.
4. Login responds to Operations Monitor's DED command by writing a DED command to each monitor that is running.
5. When a given monitor receives Login's DED command, it writes accumulated billing data for each user that it is servicing to Login and prints SYSTEM CLOSED at each user's terminal. It then closes all user files and terminates. Operations Monitor ignores Login's DED command.
6. Login writes 'OFF AT XX:XX' to each user's terminal communication file as it is returned from a terminating monitor, and closes the slave end of each such terminal communication file.
7. TCFACE, in response to Operations Monitor's ENA and DIS commands, ignores all ringing phones except on the two control lines specified by Operations Monitor. TCFACE closes the master end of each terminal communication file which is returned to it, and directs the TCC Executive to disconnect each terminal involved.
8. As a result of the dedication procedure, users not connected with Operations Monitor receive:

   ```
   SYSTEM CLOSED

   OFF AT XX:XX
   ```

   messages on their terminals and are disconnected from the system. Unless a user is on one of the two current control lines, he is subsequently unable to connect with the system.

At this point the operator runs an incremental Logical Dump to dump all files modified since the last dump, and if the schedule calls for it, a full Logical Dump to dump the entire DTSS file structure to tape.

Next a system called DAY is run to produce monthly and/or daily statistics on the printer. A system called SHUFFLE may be run next to reorganise disc storage in a more efficient manner.

At this point the operator types PROTECT to OPMON which initiates the shutdown of DTSS. The following sequence of events occurs:

1. Operations Monitor writes a DIS ALL command to TCFACE, causing TCFACE to ignore all ringing phones. Thus no additional users may sign on to the system.
2. Operations Monitor writes a PROTECT command to Login.
3. Login writes OFF commands to all monitors and sets a one-minute timer.
4. In response to Login's OFF command, all monitors, including Operations Monitor, write accumulated billing data to Login, close all user files, and terminate.
5. After one minute, Login aborts any monitors still running and sets a thirty-second timer.
6. After thirty seconds Login terminates, with termination access set to zero. TCFACE then terminates, signalling the DTSS Executive to shut down. The Executive writes its storage allocation tables, core catalogs, and other relevant information out to permanent storage and prints 'SYSTEM PROTECTED' on the machine room logging device.

All of the above shutdown steps, from the initial message to users telling them the system is going down in, say fifteen minutes, through to the PROTECT command can be carried out by a PERFORM file without operator intervention other than loading dump tapes.

The operator them communicates with the Executive through the system console. The next action is to perform a consistency check on the filing system and reconstitute the free-space tables. This is done using the master-mode PURGE command. Following this the operator may type OFF if the system is to be used for anything other than DTSS, in which case a disc boot will be required some time in the future, or ONN which is equivalent to OFF followed by a disc boot.

Following OFF or a boot the operator, and other users, can sign-on. The first job is to type OPURGE to OPMON. The Purge module (so called since before the archiving system one of it's functions was to unsave user files which had not been used since a given date) creates a file of storage information used by the billing program to charge users for the file storage they use.

Operations Monitor runs the Purge module using the Simple Monitor-Edit Interface. Operations Monitor sets the following special run parameters for the Purge module:

1. Infinite catalogued and scratch storage quotas.
2. Maximum run-time.

3. Job access to all files and catalogs in the catalog tree.
4. Large state vector and load/dump job type.
5. Maximum core size if the system is dedicated.

Finally the operator runs the billing program to produce the billing files for the day.

8.3 SOFTWARE RELEASES

The Product Release System is used for two purposes:

1. To export the current package of software products for updates at running DTSS installations.
2. To install the operating system on new customer computers.

Both updates and installations may be accomplished using the same Software Tape Set. Each Software Tape Set contains a fully integrated and tested version of the Dartmouth Time-Sharing System.

DTSS releases are numbered sequentially as Release 01, Release 02, Release 03, etc. A particular release is referred to as RELXX where XX is the release number. This scheme is used to classify software in the Product Release Catalogs which are saved in the Master User Directory.

The Software Tape Set included with each release package consists of four different types of magnetic tapes:

Warm Boot Tape
Slave Loader Input Tapes
Product Release Tape
Total System Tape

The Warm Boot Tape contains the latest version of IOMTBOOT, the Executive and Slave Loader. It may be used to perform either a cold boot or a warm boot of the system. Patches to be applied to Executive or Slave Loader may also be on this tape.

Slave Loader Input Tapes contain working copies of the rest of the system software. Slave Loader Input Tape I (SLT I) loads the DTSS operating system, while SLT II loads :DLIBRARY and its sublibraries. Slave Loader Input Tape (SLT) I will be loaded during the initial boot sequence using the Slave Loader LOAD Hn command. Slave Loader will also read patches from SLT I and apply them to the specified system software. The system will then be protected and re-booted via the Executive PROTECT and ONN commands. At this time the next Slave Loader Input Tape will be loaded. This procedure is repeated until all Slave Loader Input Tapes have been loaded.

The Product Release Tape contains only those files necessary to create Warm Boot and Slave Loader Input Tapes. It is a Logical Dump of user group *RELXX (where XX is the current release number). The user catalogs on this tape contain just the binaries and sources that will be catalogued in the MFD by Slave Loader. Product Detail Files in each catalog specify all files necessary to create the Warm Boot and slave Loader Input tapes. The Product Release Tape is loaded using Logical Load after the unmodified system is up and running. It will be used to create a modified version of DTSS.

The Total System Tape contains all official DTSS software products – binaries, sources, assemblies, alter files, symbol table files, detail files, acceptable tests, etc. This tape is created using LUPDATE and contains the same user catalogs as the Product Release Tape, but in complete form. When re-assembly or listing of a module is required, the module's Product Release Catalog must be loaded from the Total System Tape using Logical Load. Some site-specific modifications must be handled this way. Other modifications can be applied to the binaries loaded from the Product Release Tape by using the DTSS Patch File, which contains all system patches to be applied by Slave Loader at boot time. Unless a module must be re-assembled or a listing produced to re-locate patches, it should not be necessary to load its catalog from the Total System Tape; the bulk of the software binaries comprising the Warm Boot and Slave Loader Input Tapes will be loaded from the Product Release Tape.

8.3.1 Product Release Catalogs

The Product Release Catalogs on both the Product Release Tape and the Total System Tape are classified as standard DTSS user catalogs. The REL supergroup in :MUD contains all releases of DTSS software, thus serving as both a loading location for the new version and backup facility for previous versions. :MUD:REL contains a user group catalog for each release of DTSS: user group REL01 contains all Software for Release 01 and user group REL02 contains all software for Release 01, and so on.

Within each group catalog, user catalogs serve as export mailboxes for individual modules. The last three digits of the user catalog denote the module code. Thus user catalog *REL01056 contains the version of SIMON (module code 056) shipped in Release 01 and user catalog *REL02056 contains the version of SIMON shipped in Release 02. Module codes are recorded in SYSLIB*** :SOFTWARE.

Each Release Catalog on the Total System Tape contains all files necessary to produce a binary copy of its module. Also included are information and documentation files concerning the use, requirements and history of the module. File types are coded by a prefix letter followed by a dot (.) followed by as much of the module name as will fit in the remaining six characters. For example, B.EXEC is the binary file for the Executive; S.LUPDAT is the symbol-table file for LUPDATE. The source file for a module is the only file whose name does not follow the dot convention: source files bear the module name. So *REL01005:BAKMON is the source treename for the Background Monitor shipped in Release 01. Prefix letters have the meanings given in Table 8.1.

Table 8.1.

Module name	Source file
A.<module name>	Alter File
B.<module name>	Binary File
D.<module name>	Documentation File
E.<module name>	Error File
I. <module name>	Information File
J. <module name>	Background Job File
L.<module name>	Listing File
P.<module name>	Perform File
R.<module name>	Product Detail File
S.<module name>	Symbol Table File

In general, Release Catalogs loaded from the Product Release Tape contain only B. and R. files (there are exceptions such as MSTAT and :DLIBRARY which require that sources be written to the Slave Loader Input Tape). Only the Total System Tape contains the complete set of files for each Product Release Catalog.

8.3.2 Product Detail Files

The 'R.' file in each Release Catalog is the Product Detail File for the module. It provides information used to create Slave Loader Input Tapes. Each line of the detail file begins with a command followed by a list of arguments. These commands will be translated into Slave Loader commands in the process of creating the STAPEMKR input file. STAPEMKR will write the translated commands and the specified files to the Slave Loader Input Tapes. Legal commands for Product Detail Files are:

CATALOG	Catalog a catalog
UPDATE	Catalog (or replace) a file
INSTALL	Catalog (but do not replace) a file
ERASE	Erase a file or catalog
PATCH	Patch a file
FILE	Insert a Product Detail File
NOTE	Comment follows

The following command formats apply to Product Deatail Files. Arguments enclosed in <angle brackets> are required; arguments enclosed in [square brackets] are optional:

1. CATALOG<treename>;<accesses>;<max>[;<entry count>]

   Creates a catalog with the specified <treename>,<accesses> and catalog storage <max>. If an <entry count> is specified, it will be used in allocating storage for catalog entries. Otherwise, a default value of ten will be used. If the catalog already exists, its accesses will be changed to those specified and, if the present catalog storage max is less than that specified in the <max> field, it will be raised accordingly. Example:

   ```
   CAT :DLIBRARY:DATALIB;41100001100;100000;200 'SYSTEM
   SUBLIBRARY
   ```

2. UPDATE <source location>; [<desination treename>]; [<accesses>] [;<pref> [;<aloc>]]

   Causes the file named in the <source location> to be catalogued with the <desination treename> and <accesses> specified. If the file is already catalogued, it will be replaced. The <pref> field denotes the file preference to be used and the <aloc> field denotes the preallocation length. If these fields are not supplied, the default preference will be six and the default preallocation length will be zero. The <source location> field usually refers to a user catalog in the current release group. A <source location> of *056:B.SIMON is valid since the '*' will be replaced by *RELXX when the translator program (DETAIL) is run for Release XX. Example:

   ```
   UPD *006:B.BASIC;:SMSYSCAT:BASIC;010100411161 'BASIC
   COMPILER
   ```

   Some update commands may have no <desination treename> or <accesses>. Such lines are used to flag files which must be catalogued on the Product Release Tape. For example, the lines:

   ```
   UPD *026: B.EXEC;;'EXECUTIVE BINARY
   UPD *026:R.EXEC;;'EXECUTIVE PRODUCT DETAIL FILE
   ```

   appear in the Product Detail File for the Executive and specify that the files names should be placed on the Product Release Tape.

3. INSTALL [<source location>];<desination treename>; <accesses> [;<pref>[;<aloc>]]

   Causes the file named in the source location to be catalogued with the specified <desination treename> and <accesses>. If the file already exists, it will not be replaced (however, the accesses will be changed to those specified). If no <source location> is specified, a zero-length file will be catalogued. <pref> and <aloc> are the same as for the UPDATE command. Examples:

```
INS *235:DEDFILE;:BAKCAT:MUXCAT:DEDFILE
;0410000447020 'DEDICATE FILE

INS ;:DUMPCAT:ALGOL;006001407001 'DUMP FILE
```

4. ERASE <treename>

   Causes the file or catalog with the specified treename to be erased. Example:

   ```
   ERA :OPCAT:OPMOVE 'DELETE OBSOLETE SOFTWARE
   ```

5. PATCH <treename>

   Causes the file with the specified treename to be patched. Location and contents of the patch or patches should follow the command line. If no patches are included, the module will not be patched. Example:

   ```
   PAT  :MON:BAK        ´Patch out GCOS simulator requirement
   007241 000000011000 ´(NOP 0) OK if GCOS source can´t be erased
   025261 025264710000 ´(TRA 35264) OK if no GCOS
   CAT:SOURCECT
   ```

   Temporary patches or site-specific modification patches are put in the DTSS Patch file only, so that the changes made to the system can be identified.

6. FILE <treename>

   Causes the specified file to be used as an insert file. This command is used in the Master Product File to name the Product Detail Files to be used for creating the Slave Loader Input Tapes. Example:

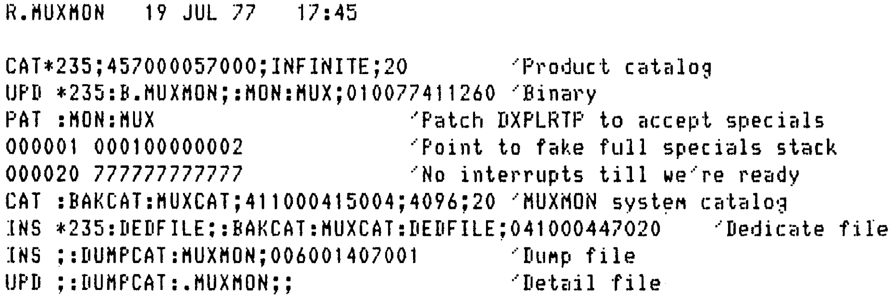
   ```
   FILE *001:R.ALGOL  ´Insert Algol detail file
   ```

7. NOTE <comment>

   Used to place a comment on a line by itself. Example:

   ```
   NOTE THIS IS A COMMENT
   ```

A sample product detail is shown below:



```
R.MUXMON   19 JUL 77   17:45

CAT*235;457000057000;INFINITE;20        'Product catalog
UPD *235:B.MUXMON;:MON:MUX;010077411260 'Binary
PAT :MON:MUX                    'Patch DXPLRTP to accept specials
000001 000100000002             'Point to fake full specials stack
000020 777777777777             'No interrupts till we're ready
CAT :BAKCAT:MUXCAT;411000415004;4096;20 'MUXMON system catalog
INS *235:DEDFILE;:BAKCAT:MUXCAT:DEDFILE;041000447020    'Dedicate file
INS ;:DUMPCAT:MUXMON;006001407001       'Dump file
UPD ;:DUMPCAT:.MUXMON;;                 'Detail file
```

8.3.3 Product Control User Number

The Product Control User Number is *RELXX000. The Product Control User Number is validated for all permissions. It is used to create the modified Slave Loader Input Tapes. The following is a list of files in *RELXX000:

DOCUMENTATION

PRODUCT Alphabetised list of products shipped.
NPRODUCT Numerical list of products shipped.
SPECIAL List of files not on Slave Loader Input Tape.
HOLD List of products being held until next release.
OBSOLETE List of products no longer being shipped.
REPORT LUPDATE report of Total System Tape.
FORMREP Formatted LUPDATE report of Total System Tape.
PRTAPE Catalog of *RELXX as dumped to the Product Release Tape.
SOFTWARE List of DTSS software modules (produced by RELPRINT).
UPDATED List of modules modified since last release.

PRODUCTION

DETAIL DXPL program to create STAPEMKR input file from Master Product File.
B.DETAIL Binary version of DETAIL.
MASTER Software Master Product File. Inserts Product Detail Files for the Desired DTSS program products. Converted by DETAIL into SLT1.
NMASTER Same as MASTER, for NON-EIS sites. Converted by DETAIL into NSLT1.
DLIBMAS Master Product File for :DLIBRARY and sublibraries. Converted by DETAIL into SLT2.
DLIBMAS2 Same as DLIBMAS, for 7-track sites. Contains only :DLIBRARY insert call. Converted by DETAIL into NSLT2.
DLIBMAS3 Same as DLIBMAS, for 7-track sites. Contains only sublibrary insert calls. Converted by DETAIL into NSLT3.

PATCHES	DTSS Patch File. Inserted by Software Master Product File, so contents are written to Slave Loader Input Tape I. Patches apply to binaries loaded from this tape.
WBT	STAPEMKR input file for Slave Loader Input Tape I.
NWBT	Same as WBT, for IOC systems.
SLT1	STAPEMKR input file for Slave Loader Input Tape II.
NSLT2	Same as :DLIBRARY portion of SLT2, for 7-track sites.
NSLT3	Same as sublibrary portion of SLT2, for 7-track sites.
SMCORCAT	The :SMCORCAT list to be catalogued automatically in :SMSYSCAT.

UTILITY

FSTATFIX	Used to initialise :CATBV:FSTAT in an emergency.
.1CLEAN	Perform file reduces a Product Release Catalog to only files specified by the Product Detail File.
.5CLEAN	Same as .1CLEAN, handles five catalogs at once.
CLEAN	DXPL program executed by above perform files.
B.CLEAN	Binary version of CLEAN.
.SCL	Perform file used in CLEAN procedure.
XCL	Program executed by .SCL.

8.3.4 Procedures for updating DTSS

The procedure is divided into three phases:

1. Loading the unmodified system.
2. Creating and loading the modified system.
3. Integrating the modified and official systems.

Loading the unmodified system

The system is first protected using OPMON's PROTECT command. A Cold Boot Patch Card and the ENV deck are placed in the card reader. The DTSS Warm Boot Tape and a scratch tape with write ring are mounted, and the INITIALIZE and BOOT buttons on the console are pressed. The EXECUTIVE and Slave Loader will be loaded from the Load Tape.

The Slave Loader Input Tapes are mounted and Slave Loader is run, and the software will be loaded and finally DTSS should then begin operating.

Turning on a control-line terminal will cause the DTSS greeting to be printed on the terminal. Since a cold boot has been performed and no :MUD was loaded from the Slave Loader Input Tapes, the operator will be automatically signed-on to OPMON with the MFD as his current catalog. Running LOAD and typing AC *RELXX where XX is the current release number, mounting the Product Release Tape when Logical Load asks and finally protecting the system and typing ONN to initialise :MUD will produce a running Unmodified System.

Creating and loading the modified system

All site-specific alters and patches are consolidated. Product Release Catalogs from the Total System Tape for modules that require re-assembly or relocation of patches are loaded and re-assembled and/or listed using the 'J.' files in the Product Release Catalogs. All site-specific patches are put in the Patch File *RELXX000:PATCHES. The Master Product File is edited to delete any modules not being updated. The program XRELXX000:B.DETAIL is run and the output file will contain a STAPEMKR file. STAPEMKR is run to create Slave Loader Input Tape I.

DTSS is then cold booted using the new Warm Boot Tape. All modified Slave Loader Input Tapes are loaded together with all those not requiring modification. Logical Load is then run from a control-line terminal and the Product Release Tape is loaded. This is followed by a disc boot using ONN and :MUD is initialised.

Integrating the modified and official systems

There are two recommended ways of integrating the Modified and Official Systems to complete the update. If the system has a large number of site-maintained programs and files scattered throughout the MFD, the Modified System is warm booted over the Official System. This forfeits one major advantage of the release, namely the cleanup of unnecessary files. Such files must be unsaved by hand. But if the system has few or no site-maintained programs and files, the Modified System is cold booted, then :MUD is loaded from the current Dump Tapes.

8.3.5 Procedures for installing DTSS

One critical difference between updates and installations is that often the Front-End Processor on new systems is not working. This necessitates installation under the Front-End Simulator (FRESIM). To install DTSS under FRESIM, the steps outlined earlier for loading the unmodified system are followed. When the last Slave Loader Input Tape has been loaded, the following steps are taken:

The system is protected using the Executive PROTECT command. Typing ONN will bring the EXEC up again. Typing LOAD T will force Slave Loader to take commands from the console.

Entering the following records on the console:

```
UPD  :D-30FACE:TCFACE;010000411777 (EOM)
WITH :D-30FACE:D-30FACE            (EOM)
REF  :D-30FACE:D-30FACE            (EOM)
WITH :D-30FACE:FRESIM              (EOM)
***EOF                             (EOM)
```

will bring up the Front-End Simulator using the console as a terminal.

Typing RUN :D-30FACE:FCFACE#ATT will bring DTSS up. Since no :MUD exists, the operator will be signed in to OPMON with the MFD as his current catalog.

LOAD is run to load the Product Release Tape. When running under FRESIM, LOGIN uses :XSYSCAT as the MFD in which it searches for :MUD user catalogs. When the load is finished, FILEMOVE is run to move :MFD:MUD to :XSYSCAT:MUD.

8.4 OPERATOR INTERFACE TO BAKMON

BAKMON runs jobs which are not connected to time-sharing terminals, and allocates processor time and peripherals to these jobs. Normally, BAKMON does this automatically, and operator intervention is required only for the handling of input and output materials. BAKMON does, however, accept one control terminal to provide an interface with the system operator. Via this interface the operator may check the status of jobs being processed, relinquish and seize control of peripherals, and regulate the flow of jobs. This section describes BAKMON's operator interface.

To establish communication with BAKMON, the system operator signs-on on a control line and types:

```
MON BAK
```

The ABORT command causes the Background Monitor to abort the specified job. The following message is sent to the user execution report:

```
JOB ABORTED BY OPERATOR
```

BAKMON automatically checks the background submission catalog every fifteen minutes for new jobs to be processed. It also checks the submission catalog every time a user submits a job from a time-sharing terminal.

In addition, BAKMON can be manually instructed to check the submission catalog via the AWAKE command. This is useful when the oeprator wishes BAKMON to begin processing immediately those jobs that have been submitted.

The maximum number of non-BMC activities that BAKMON will run at any one time is determined by an internal job slot counter. BAKMON re-adjusts this job slot counter every fifteen minutes according to the number of users connected to the time-sharing system; as more users sign-on to the system, the counter is lowered and fewer background jobs are run. BMC jobs are exempted from this constraint because they require a very small amount of memory, and at any given time only one BMC job may be in execution for each peripheral; thus they place a very small load on the system.

The JOB command causes BAKMON to reset the job slot counter to the specified value. For example, if the operator types:

```
JOB 5
```

the job slot counter is set to five and the Background Monitor will run a maximum of five non-BMC activities simultaneously. Note that the core count set by the operator is not permanent; it will be reset automatically in fifteen minutes or less.

The GET command causes BAKMON to take control of the specified peripheral. If the peripheral is busy in foreground, the monitor will respond:

```
PERIPHERAL <peripheral abbreviation> BUSY
```

The KLIMIT command resets the limit on memory size for jobs submitted by users with super-large-core permission. (All other jobs have a fixed memory size limit set by the user's validation permissions.) Currently, BAKMON will not set the super-large-core limit below 32K or above 128K, if no KLIMIT command has been given, the memory size limit for both types of job is 32K.

If a job submitted by a user with super-large-core permission requests a memory size greater than 128K, BAKMON will abort it; if the job requests a memory size less than 128K but greater than the current super-large-memory size limit, the job will be suspended until the operator specifies a large enough memory size limit via the KLIMIT command.

The operator may determine of there are any unfulfilled requests for large memory size via the RLIST command.

The OFF command causes BAKMON to terminate. Upon receiving this command, the monitor will cease to accept new jobs from the SUBMIT catalog and will halt execution of all running jobs. It will also send a message to the console typewriter stating which jobs were halted; these jobs will be restarted automatically when the monitor next comes to life.

The PSTATUS command causes BAKMON to print the status of the specified peripheral(s). The argument for this command is either a specific peripheral abbreviation or the argument ALL, which causes BAKMON to print the status of all peripherals currently under its control. The various possible peripheral statuses are as follows:

FREE
BAKMON has control of the peripheral, but no job is currently using it.

ALLC OPERATOR (RELEASED)
BAKMON does not have control of the specified peripheral, either because the monitor never opened it, or because the operator has released it.

ALLC JOB XX, USER: <user number>, PROGRAM: <program name>
The peripheral is currently reserved for or being used by the stated job.

In addition to these statuses, there are two special variants of the FREE status, which are supplied only for tape handlers:

FREE WITH UNUSED SCRATCH TAPE BAKSx

The named background scratch tape is mounted at load point, and will be assigned to the next job that requires a scratch tape. The tape may be dismounted by releasing the handler via the RELEASE command.

FREE WITH SAVED TAPE FOR JOB XX, USER: <user number>,
PROGRAM: <program name>

Tape handlers will maintain this status only for brief periods of time, so it is unlikely that this message will be seen frequently. It indicates that the handler is available for use by another job but that the specified job will soon again perform operations on the tape.

The RELEASE command causes BAKMON to relinquish control of the specified peripheral. If the peripheral is not in use, it is released immediately; if a job is using the peripheral, it will not be released until the current activity of that job has been completed. When the specified peripheral has been released, BAKMON logs the following message to the console typewriter:

```
PERIPHERAL RELEASED: <peripheral abbreviation>
```

The RLIST command causes BAKMON to output a list of the resources for which the specified job is waiting. This is printed in the format:

```
JOB XX: <number> <resource 1> (<device list>)
        <number> <resource 2>
        CORE SIZE <size>
        RUN TIME <time>
```

For example:

```
JOB 03:  1 PRINTER
         2 TAPE HANDLER(S)  (0,3,4,5)
         1 CORE UNIT(S)
         CORE SIZE 144000
```

It should be noted that a job never seizes any resources until it can get all of those for which it is waiting. Thus, some of the resources specified in response to the RLIST command may already be available. In the above example, two of the specified tape handlers and the printer may already be free, and the core size limit for the given type of job may be above 144000; but because no core units are available, the job must wait. It will wait until all five of the resources are simultaneously available.

The RLIST command specifies all resources for which the job is waiting. The RWAIT command however specifies only those resources for which the job is waiting which are not presently under the control of BAKMON.

The RTIME command causes BAKMON to print the amount of run time that the current activity in the specified job has accumulated. Run time is printed in the format:

```
JOB XX:  T.TT SEC. RUN TIME
```

If the job is not running (that is, no activities are in progress), the monitor will respond:

```
JOB XX:  NOT RUNNING
```

RUN <special activity name>

The RUN command causes BAKMON to initiate a special activity. A special activity is a predefined background job which is run whenever the operator requests it. The background job description for a special activity is permanently catalogued in the catalog :BAKCAT:RUNNING.

When the special activity has been initiated, BAKMON logs the following message to the console typewriter:

```
SPECIAL ACTIVITY <special activity name> STARTED
```

On completion, the monitor logs:

```
SPECIAL ACTIVITY <special activity name> COMPLETED
```

A typical special activity is DUMP, and it runs the system Dump Module. The DUMP special activity is also run by BAKMON every fifteen minutes without operator intervention.

The RWAIT command causes BAKMON to list all jobs which are suspended because they require the use of peripherals which are not under BAKMON's control. The format of each entry in the monitor's reponse to the RWAIT command is:

RESOURCE REQUEST FOR JOB XX, USER: <user number>, PROGRAM:
<program name> N <peripheral abbreviation> (<device list>)

For example:

```
RESOURCE REQUEST FOR JOB 05, USER: HDC51514,
    PROGRAM: BACK5
        1 TAPE HANDLER (0,3,5)
```

This message should be interpreted to mean that job 5 needs one tape handler: either handler 0, handler 3, or handler 5. The operator should therefore respond by typing GET HAN0, or GET HAN3, or GET HAN5.

Note that this same RESOURCE REQUEST message also appears on the console typewriter (without operator intervention) when BAKMON first realises that a job requires resources which are not available. The device list contains a list of devices which would satisfy the request.

The STATUS command causes BAKMON to print the status of the specified job. This status report takes the form:

```
JOB: XXXXXX, USER: <user number>, PROGRAM: <program
name><activity type> ACTIVITY <status>
```

where activity type is one of the following:

FOREGROUND	=	foreground simulation
RUN	=	general run
BMC	=	background media conversion
GCOS	=	GCOS simulation
SYSOUT	=	SYSOUT run
ABORT	=	abort initiated by user

and status is one of the following:

STARTING. The job is just starting up. No activity has yet been initiated.

RUNNING AT LINE LL. The activity at line LL in the user's background program is running.

WAITING AT LINE LL. The activity at line LL in the user's background program is waiting for resources.

ENDING. All activities have been completed and the job is nearly finished.

ABORTING. An error has occurred during the execution of the job, or the user or operator has aborted it. The job will soon be removed from the job stream.

The TELL command allows the operator to write a message to a user's background job.

The TLIMIT command has the same format and options as the KLIMIT command, but is used to limit the amount of time jobs are allowed to run.

The following commands are debugging aids and are not used during normal operation:

DIE	terminate with a memory dump
DIE NDUMP	terminate without a dump
DMPURE	declare BAKMON impure
DUMP	dump memory location(s)
PATCH	patch memory locations(s)
RCHECK	check the resource waiting list
SNAP	produce memory dump without terminating

8.5 REFERENCES

[1] Harlowe, C. (April 1974), 'DTSS Operator's Manual', DTSS Technical Memorandum 066.

[2] Pedersen, L. C. (November 1977), 'System Start-Up', DTSS Inc. Systems Programmer's Training Course notes.

[3] Green, R. L. (July 1977), 'Loading DTSS Software', DTSS System Memorandum 299.

Index

Gordon Bull came into computing by accident. After graduating in Mathematics from the University of London in 1961, he joined an insurance company as a trainee actuary. After one month he concluded that working in an office was not for him and resigned. Needing a job, he accepted a post as a research assistant in the Mathematics Department at Letchworth College, only to find when he took up the appointment that it wasn't mathematics at all, but that he was expected to design and implement a programming language for a computer being designed and built by the college. In 1964 he moved to the Hatfield Polytechnic where he has been ever since. He was involved in the setting-up of the degree in computer science and the selection of the first major time-sharing system for education in the UK at Hatfield.

In 1967, in order to gain experience of time-sharing, he spent six months at Dartmouth College, New Hampshire, U.S.A., working as part of the team writing DTSS, which marked the beginning of a long association with the College. Sandwich students from Hatfield have spent industrial periods at Dartmouth, contributing to the system; and Gordon Bull has returned to Dartmouth many times for brief visits. He recently spent a year there as a Research Associate Professor working for DTSS Inc. as an educational consultant, and for the college helping with the design of a Masters Program in Computer Science and with the specification of the seventh version of BASIC.

He has written four previous books in the computing field, numerous technical papers, and the chapter on BASIC for *Programming Language Standardisation* (edited by I. D. Hill and B. L. Meek, Ellis Horwood Limited, 1980). Since 1971 he has been actively engaged in the standardisation of BASIC, initially promoting the need for a standard and, since 1974, as a member of ANSI, ECMA and BSI committees and as a member of the European Workshop on Industrial Computer Systems (EWICS) committee, developing real-time BASIC. He also edits the EWICS newsletter.

He plays squash to keep fit and has been known to cycle to work (weather permitting)!

Cover: Dartmouth College, New Hampshire, U.S.A.

INTERACTIVE COMPUTER GRAPHICS IN SCIENCE TEACHING
Edited by J. McKENZIE, University College London, R. LEWIS, Chelsea College, University of London, and L. ELTON, University of Surrey

"well organised volume of particular value to educators interested in computer applications" – G. R. Brubaker, Illinois Institute of Technology, in *Journal of Chemical Education.*

INTRODUCTORY ALGOL 68 PROGRAMMING
D. F. BRAILSFORD and A. N. WALKER, University of Nottingham

This powerful, concise and simple programming language is explained by two pioneers in its teaching, leading from fundamental concepts, to practical applications by case studies and examples. Many advanced features are discussed.

GUIDE TO GOOD PROGRAMMING PRACTICE
Edited by B. L. MEEK, Queen Elizabeth College, University of London and P. M. HEATH, Programming Manager, Computer Centre, Plymouth Polytechnic

Brings together, in compact and assimilable form, guidance on every aspect of the programmer's job, including analysis, design, testing, debugging, 'tuning' for greater efficiency, coping with the limitations of a system, selection of language and the use of language standards. A summary of good practice for the trainee or the experienced programmer.

CLUSTER ANALYSIS ALGORITHMS for data reduction and classification of objects
H. SPATH, University of Oldenburg

Presents the current state of the art in practical cluster analysis, introducing and comparing a wide variety of algorithms used to group together objects of similar characteristics as an aid to classification and data reduction. Theoretical background included.

DYNAMIC REGRESSION: Theory and Algorithms
M. H. PESARAN, Fellow of Trinity College, Cambridge and L. J. SLATER, Department of Applied Economics, University of Cambridge

The first complete and thorough published treatment of the practical algorithms based on dynamic regression theory. Algorithms in the form of four rigorously-tested Fortran programs suitable for implementation on a wide range of computers are thus now made available to the financial, statistical, industrial and academic worlds.

FOUNDATIONS OF PROGRAMMING WITH PASCAL
LAWRIE MOORE, Birkbeck College, University of London

First-course text in computer programming in a refreshing straightforward style, blows the dust off the outmoded notion that programming languages such as Pascal, used here as being the leading introductory language, are formidable. A sound practical and theoretical pathway, avoiding unfamiliar mathematics, explains concepts, design and structure. Teaches us how to enjoy computer programming with the skill and pleasure of a craftsman.

PROGRAMMING LANGUAGE STANDARDISATION
Edited by I. D. HILL, Division of Computing and Statistics, Clinical Research Centre, Harrow, and B. L. MEEK, Director, Computer Unit, Queen Elizabeth College, University of London

A clear and thorough explanation of the aims and effects of standardisation at national and international levels. Reviews the entire field, and includes lively and entertaining argument, conveying the informal atmosphere of such discussions. Provides information and provokes thought. For computer users, managers and programmers working in commercial, administration, industry and education.

published by
ELLIS HORWOOD LIMITED
Publishers Chichester

distributed by
HALSTED PRESS a division of
JOHN WILEY & SONS
New York Chichester Brisbane Toronto

Library Edition: ISBN 0-85312-253-9